

Kubernetes 101

Introduction à l'orchestration de conteneurs

Université Claude Bernard Lyon 1

Hugo Blanc
CC BY-SA 4.0

2026 - 2027

1. Introduction à Kubernetes
2. Briques de bases
3. Premiers pas avec K8s
4. Déployer de manière déclarative
5. Persistance et données
6. Sécurité

0.1 À propos

Ce cours est un cours d'introduction à l'orchestration de conteneurs avec Kubernetes.

Il est prévu pour un volume horaire d'approximativement 10 heures. Ainsi, il ne couvrira qu'en surface le fonctionnement de Kubernetes et son éco-systèmes, et pour la bonne compréhension du cours, certains abus seront fait avec des notes de bas de page.

0.2 Prérequis

- Avoir une machine GNU/Linux en état de fonctionnement
- Avoir un utilisateur différent de root appartenant au groupe sudo
- Avoir une connexion à Internet
- Savoir naviguer dans un terminal UNIX
- Avoir des notions de conteneurisation (OCI)
- Connaitre les bases des réseaux informatiques TCP/IP

Attention

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

0.3 Où trouver le cours ?

Tout est disponible sur mon site web: <https://syscall.cafe/t/>

0.4 Évaluation

Ce cours sera évalué par :

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

À certains endroits du cours, généralement en fin de chapitre, vous trouverez des blocs :



Aide aux révisions

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua quaerat.

Ils contiennent des exercices-types des évaluations.

- 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.
- Toute aide durant les contrôles sur table sera **considérée comme de la triche**, et mènera à une note de zéro.

1. Introduction à Kubernetes
2. Briques de bases
3. Premiers pas avec K8s
4. Déployer de manière déclarative
5. Persistance et données
6. Sécurité

1.1 Qu'est-ce que Kubernetes ?

Kubernetes (souvent abrégé K8s) est une plateforme open-source qui permet **l'orchestration d'applications conteneurisées**.

Le terme « orchestration » comprend le déploiement, la mise à l'échelle et la gestion du cycle de vie des conteneurs.

1.1 Qu'est-ce que Kubernetes ?

Sans orchestration :

- déploiement manuel (ou scripté dans le meilleur des cas) de chaque application sur les bons serveurs;
- surveillance leur état de santé;
- redémarrage manuelle (ou scripté) si elles plantent;
- gestion des mises à jour;
- équilibrage de la charge entre les instances manuel
- ...

1.1 Qu'est-ce que Kubernetes ?

Sans orchestration :

- déploiement manuel (ou scripté dans le meilleur des cas) de chaque application sur les bons serveurs;
- surveillance leur état de santé;
- redémarrage manuelle (ou scripté) si elles plantent;
- gestion des mises à jour;
- équilibrage de la charge entre les instances manuel
- ...

Kubernetes fait tout cela (presque) automatiquement !

1.1 Qu'est-ce que Kubernetes ?

- **Orchestration automatisée** : placement intelligent des conteneurs sur les noeuds.
- **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
- **Configuration déclarative** : vous décrivez l'état souhaité, Kubernetes s'occupe du reste

1.2 Architecture d'un cluster Kubernetes

K8s s'utilise sous forme de **cluster** : un regroupement de plusieurs services qui ensemble, forment une solution fonctionnelle cohérente et unifiée.

1.2 Architecture d'un cluster Kubernetes

Un cluster Kubernetes fonctionne avec deux types de composants principaux :

- le **plan de contrôle** (*control plane*);
- les **noeuds** de travail (*worker nodes*).

1.2 Architecture d'un cluster Kubernetes

1.2.1 Control plane

Le Control Plane est le cerveau du cluster. Il est composé des services suivants :

- **API Server** (kube-apiserver)

Point d'entrée unique pour toutes les instructions externes. Valide et stocke les configurations. Interface REST/JSON pour tous les clients (kubectl, ...).

- **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.

1.2 Architecture d'un cluster Kubernetes

- **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.

1.2 Architecture d'un cluster Kubernetes

1.2.2 Worker nodes

Chaque node est chargé d'accueillir les applications, et quelques services :

- **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: Docker¹, containerd, CRI-O...

¹<https://kubernetes.io/blog/2020/12/08/kubernetes-1-20-release-announcement/#dockershim-deprecation>

1.2 Architecture d'un cluster Kubernetes

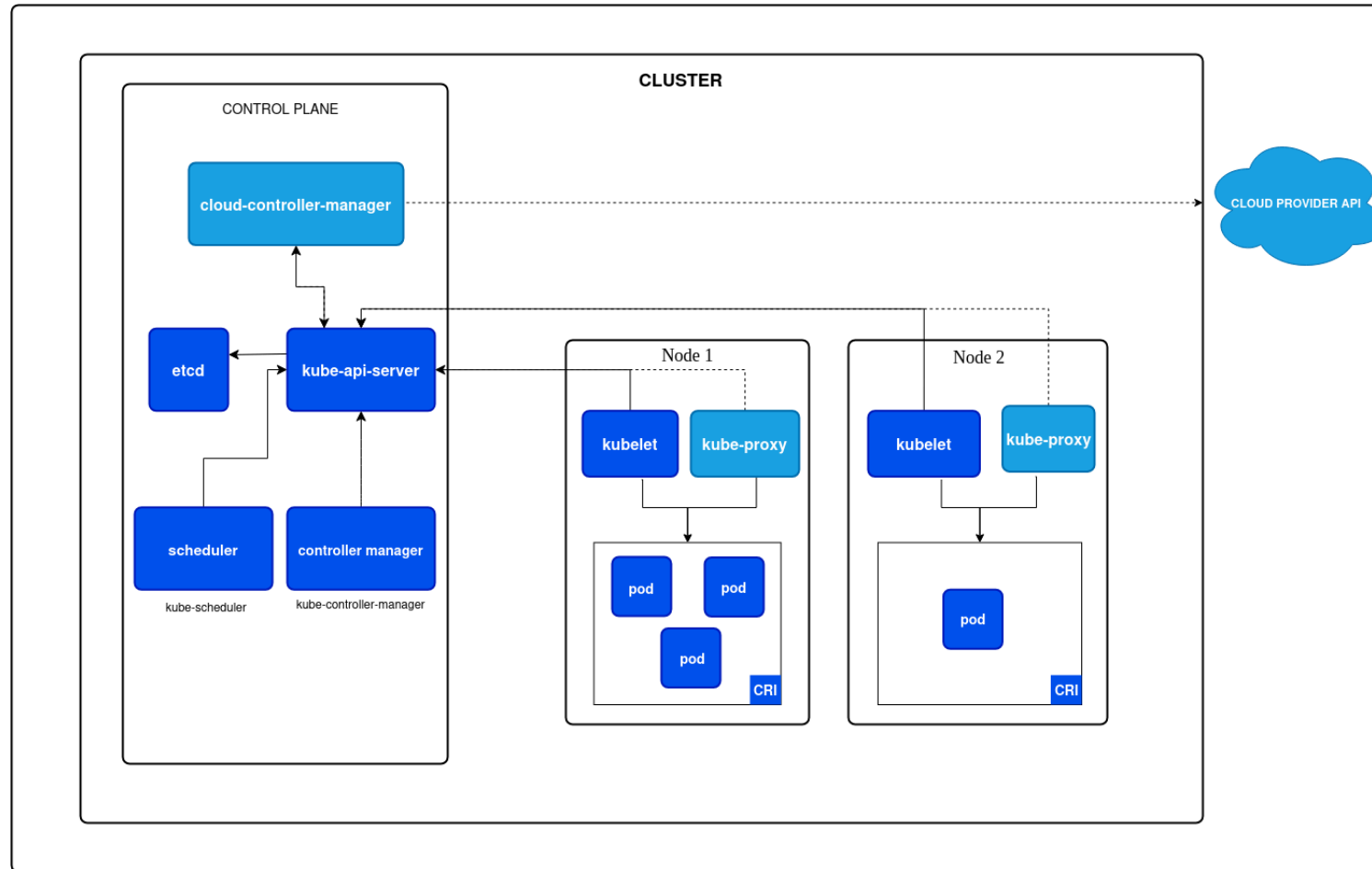


Fig. 1. – Kubernetes cluster components, kubernetes.io

1.2 Architecture d'un cluster Kubernetes

Lorsque vous parlez à un cluster K8s, tout passe par l'**API Server** :

- les utilisateur.rices y communiquent via `kubectl` ou l'API REST;
- les nodes s'y enregistrent et récupèrent leurs instructions;
- tous les composants du **Control Plane** y parlent;
- ...

1.2 Architecture d'un cluster Kubernetes

Lorsque vous parlez à un cluster K8s, tout passe par l'**API Server** :

- les utilisateur.rices y communiquent via `kubectl` ou l'API REST;
- les nodes s'y enregistrent et récupèrent leurs instructions;
- tous les composants du **Control Plane** y parlent;
- ...

Seul l'API Server lit/écrit dans etcd.

1.3 Modèle déclaratif

Traditionnellement, l'administration système se fait via une approche **impérative** :

```
$ ssh server
$ docker run --name nginx-1 nginx:1.31.1
$ docker run --name nginx-2 nginx:1.31.1

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

1.3 Modèle déclaratif

K8s favorise une approche **déclarative**,
où nous déclarons l'état souhaité :

```
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:1.31.1
```

1.3 Modèle déclaratif

Ce fichier au format YAML peut ensuite être appliqué en utilisant `kubectl` :

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

1.3 Modèle déclaratif

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.



Aide aux révisions

1. Quelle est la différence entre le Control Plane et les Worker Nodes ?
2. Quels sont les composants du Control Plane ?
3. Quelles sont les différences entre le modèle impératif et le modèle déclaratif ?

1. Introduction à Kubernetes
- 2. Briques de bases**
3. Premiers pas avec K8s
4. Déployer de manière déclarative
5. Persistance et données
6. Sécurité

2. Briques de bases

Dans l'univers Kubernetes, une infrastructure applicative se compose de plusieurs briques fonctionnelles, ayant chacune un rôle bien défini.

2.1 Pods

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

Il contient **un ou plusieurs conteneurs** qui travaillent ensemble et partagent des ressources.

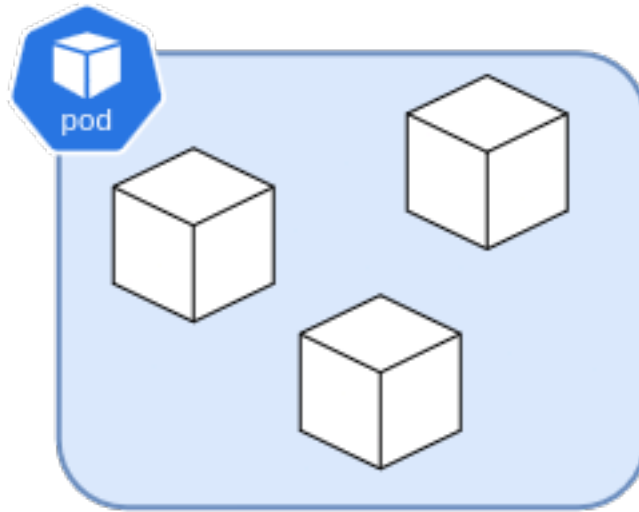


Fig. 2. – Un Pod avec 3 conteneurs.

2.1 Pods

Dans un Pod, les conteneurs...

- partagent la même adresse IP;
- partagent les mêmes volumes de stockage;
- partagent le même cycle de vie;
- communiquent entre-eux via localhost.

2.1 Pods

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: nginx
      image: nginx:1.31.1
      ports:
        - containerPort: 80
    - name: sidecar
      image: busybox
      command: ["sleep", "3600"]
```

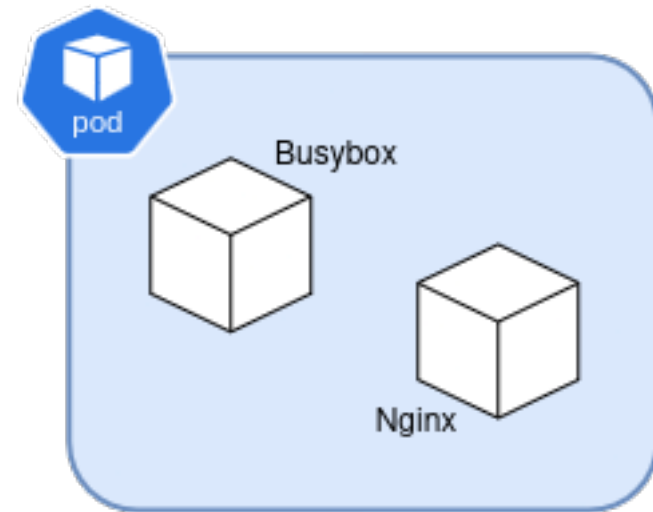


Fig. 3. – Un Pod avec 2 conteneurs : Busybox et Nginx.

2.1 Pods

En pratique, vous déployez rarement des Pods directement. Vous utilisez plutôt des objets comme les Deployments qui gèrent les Pods pour vous, voir Chapitre 2.3.

2.2 ReplicaSet

Un ReplicaSet est une ressource K8s qui permet de **garantir q'un nombre précis de Pods identiques** tourne en permanence.

2.2 ReplicaSet

Un ReplicaSet est une ressource K8s qui permet de **garantir q'un nombre précis de Pods identiques** tourne 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 répliquas, il ajuste automatiquement le nombre de Pods.

2.2 ReplicaSet

```
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:1.31.1
```

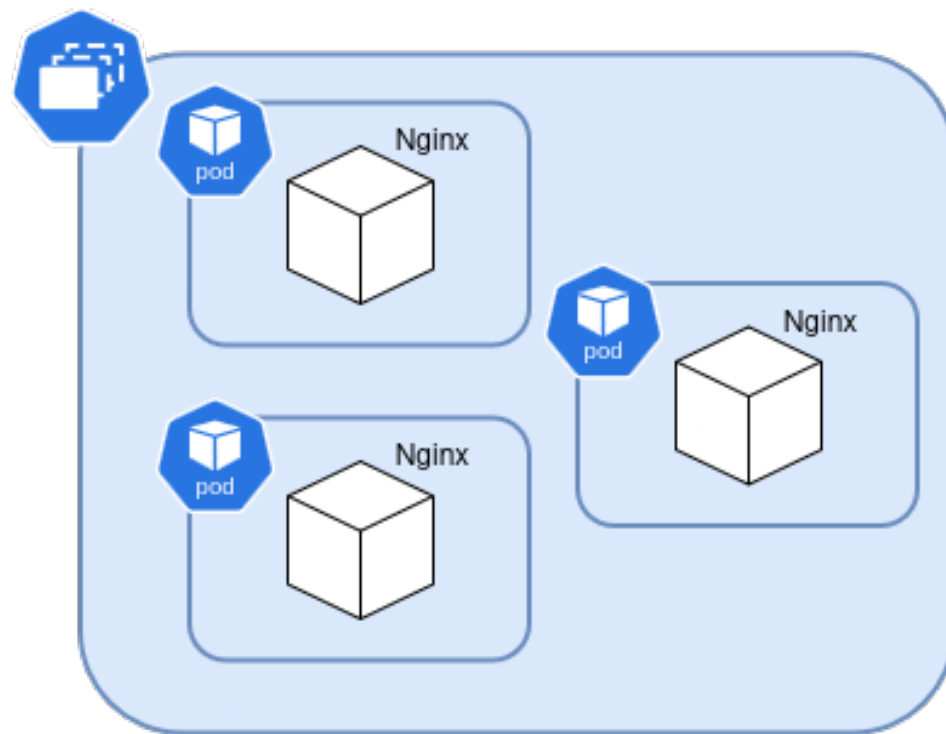


Fig. 4. – Un ReplicaSet avec 3 répliquas d'un même Pod.

2.3 Deployment

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

2.3 Deployment

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

Le Deployment est l'outil principal pour déployer des applications. Il permet, entre autres, de :

- faire des mises à jour progressives (*rolling updates*);
- revenir en arrière si quelque chose échoue ou ne va pas;
- garder un historique des versions déployées;
- mettre en pause et reprendre des déploiements en cours de route.

2.3 Deployment

```
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:1.31.1
          ports:
            - containerPort: 80
```

2.4 Service

Un Service expose les Pods sous une **identité réseau fixe**, avec équilibrage de charge automatique.

2.4 Service

Un Service expose les Pods sous une **identité réseau fixe**, avec équilibrage de charge automatique.

Les Services permettent de répondre à la problématique :

1. les Pods sont éphémères (peuvent être créés/détruits);
2. leur adresse IP change à chaque fois (voir Chapitre 2.1);
3. **comment les clients peuvent-ils accéder aux applications ?**

2.4 Service

Le Service résout ce problème en fournissant :

- une adresse IP stable (ClusterIP);
- un nom DNS stable;
- un équilibrage de charge entre les Pods.

Il existe plusieurs 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.

2.4 Service

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

2.5 DaemonSet

Les DaemonSets permettent de s'assurer qu'une copie d'un **Pod** s'exécute **sur chaque¹ noeud du cluster**. À mesure que le cluster grandit ou se réduit, le DaemonSet répartit ces Pods sur l'ensemble des noeuds.

¹Les DaemonSets peuvent être appliqués conditionnellement avec des mécanismes de nodeAffinity, taint et toleration qui ne seront pas détaillés dans ce cours. Voir [la documentation officielle](#) pour plus de détails.

2.5 DaemonSet

Utiliser un DaemonSet plutôt qu'un Deployment fait sens lorsque l'on veut s'assurer d'un conteneur tourne sur chaque noeud. Par exemple :

- probe eBPF pour surveiller les noeuds;
- logstash/fluentbit pour agréger les logs des conteneurs tournant sur le noeud;
- ...

2.5 DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: logstash
spec:
  selector:
    matchLabels:
      app: logstash
  template:
    metadata:
      labels:
        app: logstash
    spec:
      containers:
        - name: logstash
          image: docker.elastic.co/logstash/logstash:8.15.0
```

2.6 Namespaces

Les Namespaces permettent de diviser et d'organiser de manière logique les ressources pour mieux les gérer. Ils permettent notamment :

- L'isolation : séparer les environnements (dev, test, prod) ou les projets de différentes équipes pour éviter les conflits. Il est possible d'avoir deux ressources avec le même nom dans des Namespaces différents.
- La gestion des ressources : attribuer des limites de consommation (CPU, RAM) spécifiques à chaque espace via des Resource Quotas, pour éviter qu'une seule application ne monopolise tout le cluster.
- La sécurité (RBAC) : restreindre les droits d'accès, voir Chapitre 6.2.

2.6 Namespaces

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
  labels:
    env: developpement
    team: backend
```



Aide aux révisions

1. Quelle différence entre un Pod et un Deployment ?
2. Qu'arrive-t-il si vous supprimez un Pod géré par 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 ?

1. Introduction à Kubernetes
2. Briques de bases
- 3. Premiers pas avec K8s**
4. Déployer de manière déclarative
5. Persistance et données
6. Sécurité

3.1 Installation et configuration

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

3.1 Installation et configuration

3.1.1 Installer kubectl

kubectl est l'outil en ligne de commande officiel pour interagir avec des clusters Kubernetes. Ce sera votre principale interface.

```
$ 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/  
$ kubectl version --client
```

(Se référer au [guide d'installation officiel](#) si nécessaire.)

3.1 Installation et configuration

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

Je vous recommande de faire un alias `k=kubectl` dans votre shell, car vous allez beaucoup le taper :)

3.1 Installation et configuration

3.1.2 Installer minikube

Minikube permet de créer un cluster Kubernetes local à des fins de développement et l'apprentissage.

```
$ curl -LO "https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64"
```

```
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

(Se référer au [guide d'installation officiel](#) si nécessaire.)

Attention aux architectures CPU !

3.1 Installation et configuration

Et enfin...

3.1.3 Créer un cluster K8s local

```
$ minikube start
```

```
$ kubectl cluster-info
```

```
Kubernetes control plane is running
```

```
at https://127.0.0.1:32776
```

```
CoreDNS is running at
```

```
https://127.0.0.1:32776/api/v1/
```



Fig. 5. – Great success!

3.1 Installation et configuration

Pour supprimer le cluster Minikube (et tout ce qu'il contient) de manière **définitive** :

```
$ minikube delete
```

- * Deleting "minikube" in docker ...
- * Deleting container "minikube" ...
- * Removing /home/user/.minikube/machines/minikube ...
- * Removed all traces of the "minikube" cluster.

3.1 Installation et configuration

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

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

3.2 Explorer le cluster

La syntaxe la plus utilisée de kubectl est :

```
kubectl <action> <ressource> [name]
```

Par exemple :

```
$ kubectl get pods
```

```
$ kubectl get nodes
```

```
$ kubectl delete deployment nginx
```

etc.

3.2 Explorer le cluster

Une fois le cluster démarré (voir Chapitre 3.1.3), il est possible de voir ses noeuds :

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	control-plane	11s	v1.34.0

3.2 Explorer le cluster

Une fois le cluster démarré (voir Chapitre 3.1.3), il est possible de voir ses noeuds :

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	control-plane	11s	v1.34.0

Nous avons donc un seul node dans notre cluster local : le **Control Plane**. Pour du test local, pas besoin de plus.

3.2 Explorer le cluster

Nous pouvons avoir plus d'informations sur le noeud avec la commande describe :

```
$ kubectl describe node minikube
```

```
Name:                minikube
Roles:               control-plane
Labels:              beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=minikube
                    kubernetes.io/os=linux
```

```
...
```

3.2 Explorer le cluster

Regardons les Namespaces qui ont été créé :

```
$ kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	6m43s
kube-node-lease	Active	6m43s
kube-public	Active	6m43s
kube-system	Active	6m43s

3.2 Explorer le cluster

Pour voir les Pods qui sont créés sur le Control Plane, nous pouvons regarder ceux du Namespace kube-system :

```
$ kubectl get pods --namespace kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-66bc5c9577-gsh7v	1/1	Running	0	7m55s
etcd-minikube	1/1	Running	0	8m1s
kube-apiserver-minikube	1/1	Running	0	8m1s
kube-controller-manager-minikube	1/1	Running	0	8m1s
kube-proxy-z6rwh	1/1	Running	0	7m55s
kube-scheduler-minikube	1/1	Running	0	8m1s
storage-provisioner	1/1	Running	1 (7m43s ago)	8m

3.2 Explorer le cluster

Un Pod peut être dans différents états, indiqués dans la colonne STATUS :

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

3.2 Explorer le cluster



Exercices

1. Combien de volumes sont montés dans le Pod etcd-minikube?
2. Quelle est l'adresse IP assignée au Pod kube-proxy ?
3. Que se passe-t-il si l'on supprime le Pod coredns ? Pourquoi ?

3.3 Créer des Pods

Il est possible de créer des ressources K8s, comme des Pods, de manière impérative avec `kubectl run <pod name> [options]` :

```
$ kubectl run nginx --image nginx
```

```
pod/nginx created
```

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	8s

3.3 Créer des Pods



Exercices

1. Créez un nouveau Namespace nommé `playground`. Quels labels y sont automatiquement assignés ?
2. Créez un Pod `nginx` dans ce nouveau Namespace.
3. Regardez les logs de ce nouveau Pod avec `kubectl logs`.

3.4 Créer des Deployments

Comme pour les Pods, il est possible de créer des Deployments de manière impérative :

```
$ kubectl create deployment nginx --image nginx
```

```
deployment.apps/nginx created
```

```
$ kubectl get deployments nginx
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	1/1	1	1	47s

3.4 Créer des Deployments

Quand un Pod est créé par le biais d'un Deployment, son nom est suffixé par 2 valeurs : l'ID du ReplicaSet suivi l'ID du Pod unique parmi le ReplicaSet.

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-566d6954b7-bjl4w	1/1	Running	0	78s

3.4 Créer des Deployments



Exercices

1. Modifiez votre Deployment pour qu'il y ai 3 répliquas avec la commande `kubectl scale`, et vérifiez que l'ID du ReplicaSet reste inchangé alors qu'un nouvel ID de Pod est généré.
2. Testez la résilience du Deployment en supprimant un Pod. Que se passe-t-il alors ?

3.5 Exposer les conteneurs avec un Service

Pour exposer un Deployment (et ses Pods) de manière impérative, il existe la commande `kubectl expose`.

Il est ensuite possible d'accéder à l'application avec la commande `kubectl port-forward`.

3.5 Exposer les conteneurs avec un Service



Exercices

1. Créez un nouveau Deployment, utilisant cette fois l'image `ghcr.io/eze-kiel/nginx-k8s`.
2. Exposez de Deployment avec un Service de type `NodePort` sur le port 80. Attention, l'image de base écoute sur le port 8080, il faut donc utiliser l'option `--target-port` aussi.
3. Utilisez la commande `kubectl port-forward` pour forwarder le port 80 d'un des Pods vers le port 8080 de votre ordinateur.
4. Allez sur <http://localhost:8080> dans votre navigateur.

Minikube met à disposition un dashboard pour visualiser l'état du cluster :

```
$ minikube addons enable dashboard  
$ minikube dashboard
```

Cette fonctionnalité imite celle native à Kubernetes : <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>

Attention ! Bien qu'utile dans les phases de développement, exposer le dashboard K8s sur Internet amène des risques de sécurité.



Aide aux révisions

TODO

1. Introduction à Kubernetes
2. Briques de bases
3. Premiers pas avec K8s
- 4. Déployer de manière déclarative**
5. Persistance et données
6. Sécurité

4.1 Les raisons du déclaratif

Jusqu'à présent, nous avons utilisé la CLI `kubectl` pour créer/supprimer/modifier des ressources.

```
$ kubectl create deployment webapp --image=nginx:1.31.1
$ 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
```

4.1 Les raisons du déclaratif

Jusqu'à présent, nous avons utilisé la CLI `kubectl` pour créer/supprimer/modifier des ressources.

```
$ kubectl create deployment webapp --image=nginx:1.31.1
$ 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 ?

4.1 Les raisons du déclaratif

Avantages de l'approche déclarative :

- **Gestion de version** : 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 $\Leftrightarrow$ Cluster

4.2 Format

Tous les objets Kubernetes sous forme de YAML suivent la même structure de base :

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...)

4.2 Format

Tous les objets Kubernetes sous forme de YAML suivent la même structure de base :

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...)

4.2 Format

Tous les objets Kubernetes sous forme de YAML suivent la même structure de base :

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
 - `annotations` : métadonnées

4.2 Format

Tous les objets Kubernetes sous forme de YAML suivent la même structure de base :

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
 - `annotations` : métadonnées
4. `spec` : Spécification de l'état désiré
 - Structure différente selon le kind, décrit comment l'objet doit être

4.2 Format

4.2.1 Labels et sélecteurs

Les labels sont des paires clé-valeur attachées aux objets Kubernetes qui permettent de les organiser de manière logique :

```
metadata:  
  labels:  
    app: nginx  
    version: "1.31.1"  
    environment: production  
    team: frontend  
    component: webserver  
    security-scan: "passed"
```

4.2 Format

Il est ensuite généralement possible de filtrer par label avec l'option `-l` :

```
$ kubectl get pods -l app=nginx
$ kubectl get pods -l environment=production
$ kubectl get pods -l app=nginx,environment=production
$ kubectl get pods -l 'version!=1.20'
```

4.2 Format

Les labels sont également utilisés par K8s via les sélecteurs : ils permettent par exemple à Kubernetes de savoir vers quels Pods le Service doit rediriger le trafic¹ :

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      [...]
```

¹En réalité, un Service ne «redirige» pas de trafic en lui-même, il s'agit simplement d'une configuration réseau dans kube-proxy permettant le routage.

4.3 Utiliser des manifestes YAML

Pour appliquer un fichier YAML déclarant une ressource Kubernetes, il existe la commande `kubectl apply -f <file>` :

```
$ cat ns.yml
apiVersion: v1
kind: Namespace
metadata:
  name: foobar
  labels:
    name: foobar
```

4.3 Utiliser des manifestes YAML

Pour appliquer un fichier YAML déclarant une ressource Kubernetes, il existe la commande `kubectl apply -f <file>` :

```
$ cat ns.yml
```

```
apiVersion: v1
```

```
kind: Namespace
```

```
metadata:
```

```
  name: foobar
```

```
  labels:
```

```
    name: foobar
```

```
$ kubectl apply -f ns.yml
```

```
namespace/foobar created
```

4.3 Utiliser des manifestes YAML

Pour appliquer un fichier YAML déclarant une ressource Kubernetes, il existe la commande `kubectl apply -f <file>` :

```
$ cat ns.yml
apiVersion: v1
kind: Namespace
metadata:
  name: foobar
  labels:
    name: foobar
```

```
$ kubectl apply -f ns.yml
namespace/foobar created
```

```
$ kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	7m31s
foobar	Active	28s
kube-node-lease	Active	7m31s
kube-public	Active	7m31s
kube-system	Active	7m31s

4.3 Utiliser des manifestes YAML

Une fois une ressource K8s déployée, il est possible de voir à quoi ressemble sa définition finale sous forme déclarative (en YAML), avec la combinaison de `kubectl get` et l'option `--output yaml` :

```
$ kubectl get deployments nginx-k8s --output yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: "2026-08-15T07:18:39Z"
  generation: 1
  labels:
    app: nginx-k8s
  name: nginx-k8s
# ...
```

4.3 Utiliser des manifestes YAML



Exercices

1. Avec des primitives du Shell (pas de copier-coller !), récupérez le YAML du Deployment `nginx-k8s` dans le fichier `deployment.yml`.
2. Augmentez le nombre de répliquas à 3, puis appliquez avec `kubectl apply`.
Un nouveau Deployment est-il créé ? Pourquoi ?

1. Introduction à Kubernetes
2. Briques de bases
3. Premiers pas avec K8s
4. Déployer de manière déclarative
- 5. Persistance et données**
6. Sécurité

5. Persistance et données

Kubernetes propose 2 grandes abstractions concernant le stockage qui dépendent du cycle de vie des Pods ou non : les **volumes** et les **volumes persistants**.

5.1 Volumes

Par défaut, le système de fichiers d'un conteneur est **éphémère** : si le conteneur redémarre, tout ce qu'il y a écrit est perdu.

De plus, chaque conteneur a son propre système de fichiers, isolé des autres.

¹<https://kubernetes.io/docs/concepts/storage/volumes/>

5.1 Volumes

Par défaut, le système de fichiers d'un conteneur est **éphémère** : si le conteneur redémarre, tout ce qu'il y a écrit est perdu.

De plus, chaque conteneur a son propre système de fichiers, isolé des autres.

Les **volumes**¹ répondent à ces deux problématiques : un volume est un **répertoire monté dans les conteneurs d'un Pod**, dont le contenu survit aux redémarrages des conteneurs.

¹<https://kubernetes.io/docs/concepts/storage/volumes/>

5.1 Volumes

Un volume « simple » suit le **cycle de vie de son Pod** :

- il survit aux redémarrages des conteneurs;
- il est détruit avec le Pod.

On parle de volume **éphémère**. Pour des données qui doivent survivre au Pod, il faudra un volume **persistant** (voir Chapitre 5.2).

5.1 Volumes

Un volume se déclare en 2 temps dans un manifeste :

1. `spec.volumes` : la liste des volumes du Pod, chacun avec un nom et un type;
2. `volumeMounts` : le point de montage du volume dans chaque conteneur qui l'utilise.

5.1 Volumes

5.1.1 emptyDir

Un volume emptyDir est un répertoire **initialement vide**, créé quand le Pod est assigné à un noeud, et supprimé **définitivement** avec le Pod.

Usages typiques : fichiers temporaires, cache, partage de fichiers entre les conteneurs d'un même Pod.

5.1 Volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: nginx
      image: nginx:1.31.1
      volumeMounts:
        - name: share
          mountPath: /share
    - name: sidecar
      image: busybox
      command: ["sleep", "3600"]
      volumeMounts:
        - name: share
```

```
      mountPath: /share
volumes:
  - name: share
    emptyDir: {}
```

5.1 Volumes

Ici, les deux conteneurs voient le même répertoire /share : un fichier écrit par l'un est immédiatement visible par l'autre.

5.1 Volumes

Deux options sont à connaître :

- `medium: Memory` : le volume est monté en RAM (`tmpfs`). Plus rapide, mais volatile, et l'espace utilisé compte dans la limite mémoire des conteneurs;
- `sizeLimit` : taille maximale du volume.

`volumes:`

- `name: cache`

- `emptyDir:`

- `medium: Memory`

- `sizeLimit: 500Mi`

5.1 Volumes

5.1.2 hostPath

Un volume hostPath monte un fichier ou un répertoire **du noeud hôte** dans le Pod :

5.1 Volumes

5.1.2 hostPath

Un volume hostPath monte un fichier ou un répertoire **du noeud hôte** dans le Pod :

- le Pod devient dépendant du noeud sur lequel il tourne;
- le Pod accède au système de fichiers de l'hôte.

Attention

hostPath présente des risques de sécurité importants : un Pod compromis peut lire/modifier des fichiers sensibles du noeud. À n'utiliser que si nécessaire¹, typiquement pour des composants système (comme les DaemonSets d'agrégation de logs).

¹la [documentation officielle](#) recommande de le restreindre au maximum.

5.1 Volumes

Le champ `type` permet de contrôler ce qui est attendu sur l'hôte : `Directory` (doit exister), `DirectoryOrCreate` (créé si absent), `File`, `FileOrCreate`, `Socket`...

`volumes:`

- `name: logs-noeud`
 `hostPath:`
 `path: /var/log`
 `type: Directory`

5.1 Volumes

5.1.3 configMap

Une ConfigMap est un objet K8s qui stocke de la **configuration non sensible** sous forme de paires clé-valeur, séparément de l'image du conteneur.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  LOG_LEVEL: debug
  app.conf: |
    listen 8080;
    workers 4;
```

5.1 Volumes

Une ConfigMap peut être montée dans un Pod comme un volume : **chaque clé devient un fichier** dans le répertoire de montage, toujours en **lecture seule**.

5.1 Volumes

```
spec:
  containers:
    - name: app
      image: nginx:1.31.1
      volumeMounts:
        - name: config
          mountPath: /etc/config
  volumes:
    - name: config
      configMap:
        name: app-config
```

Le conteneur voit alors les fichiers `/etc/config/LOG_LEVEL` et `/etc/config/app.conf`.

5.1 Volumes

Les ConfigMaps peuvent aussi être injectées en variables d'environnement, mais contrairement au montage en volume, celles-ci ne sont pas mises à jour si la ConfigMap change.



Exercices

1. Écrivez un manifeste `pod.yml` décrivant un Pod avec 2 conteneurs `busybox` (`command: ["sleep", "3600"]`) partageant un volume `emptyDir` monté dans `/` partage.
2. Avec `kubectl exec`, écrivez un fichier depuis le premier conteneur, puis lisez-le depuis le second.
3. Supprimez le Pod et recréez-le. Le fichier existe-t-il toujours ? Pourquoi ?

5.2 Volumes persistants

Une base de données dans un Pod avec un `emptyDir` perdrait toutes ses données à la moindre suppression du Pod...

Kubernetes découple donc le stockage du cycle de vie des Pods avec deux objets :

- **PersistentVolume (PV)** : un morceau de stockage du cluster (disque cloud, NFS, ...), avec un cycle de vie **indépendant de tout Pod**;
- **PersistentVolumeClaim (PVC)** : une **demande** de stockage faite par un.e utilisateur.rice («je veux 1 Gio en lecture-écriture»).

5.2 Volumes persistants

L'analogie est la même qu'entre noeuds et Pods : les Pods consomment les ressources des noeuds, les PVC consomment les ressources des PV.

- le PV est une ressource **à l'échelle du cluster**;
- le PVC est une ressource **dans un Namespace**;
- K8s lie chaque PVC à un PV compatible (**binding**), de manière exclusive : un PV ne peut être lié qu'à un seul PVC.

5.2 Volumes persistants

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

```
apiVersion: v1
kind: Pod
metadata:
  name: db
spec:
  containers:
    - name: db
      image: postgres:17
      volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: data-pvc
```

5.2 Volumes persistants

Les **modes d'accès** (`accessModes`) déclarent comment le volume peut être monté :

- `ReadWriteOnce (RWO)` : lecture-écriture par **un seul noeud**;
- `ReadOnlyMany (ROX)` : lecture seule par plusieurs noeuds;
- `ReadWriteMany (RWX)` : lecture-écriture par plusieurs noeuds;
- `ReadWriteOncePod (RWOP)` : lecture-écriture par **un seul Pod**.

5.2 Volumes persistants

Les **modes d'accès** (accessModes) déclarent comment le volume peut être monté :

- ReadWriteOnce (RWO) : lecture-écriture par **un seul noeud**;
- ReadOnlyMany (ROX) : lecture seule par plusieurs noeuds;
- ReadWriteMany (RWX) : lecture-écriture par plusieurs noeuds;
- ReadWriteOncePod (RWOP) : lecture-écriture par **un seul Pod**.

Tous les types de stockage ne supportent pas tous les modes : un disque cloud classique est RWO, un NFS peut être RWX.

5.2 Volumes persistants

5.2.1 Provisionnement

Deux façons de créer des PV :

- **statique** : un.e admin crée les PV à la main, en avance;
- **dynamique** : le cluster crée automatiquement un PV à la demande, quand un PVC apparaît.

5.2 Volumes persistants

Le provisionnement dynamique repose sur les **StorageClasses** : le PVC référence une classe (via `storageClassName`, ou la classe par défaut du cluster), et le provisionneur associé crée le PV correspondant.

C'est ce que fait Minikube avec sa StorageClass standard et son Pod storage-provisioner (aperçu dans `kube-system`) !

5.2 Volumes persistants

Que devient un PV quand son PVC est supprimé ? C'est la politique de récupération (`persistentVolumeReclaimPolicy`) qui décide :

- **Retain** : le PV et ses données sont conservés, un.e admin doit faire le ménage manuellement;
- **Delete** : le PV et le stockage sous-jacent sont supprimés (défaut pour le provisionnement dynamique).

5.2 Volumes persistants



Exercices

1. Créez un PVC demandant 1 Gio en ReadWriteOnce. Quel est son statut ?
2. Un PV a été créé automatiquement (`kubectl get pv`). Qui l'a créé ?
3. Quelle est la politique de récupération de ce PV ? Que se passera-t-il si vous supprimez le PVC ? Vérifiez.

5.2 Volumes persistants

5.2.2 StatefulSets

Un Deployment considère ses Pods comme **interchangeables** : noms aléatoires, pas d'ordre, stockage partagé... Problématique pour une base de données répliquée, où chaque instance a son identité et **ses propres données**.

5.2 Volumes persistants

Un StatefulSet gère des Pods avec une **identité stable et unique** :

- des noms prévisibles : db-0, db-1, db-2...;
- un nom DNS stable par Pod, via un Service dit *headless*;
- **un PVC dédié par Pod**, qui suit le Pod même s'il est replanifié ailleurs;
- un déploiement et une mise à l'échelle **ordonnés** : db-1 n'est créé que quand db-0 est prêt.

5.2 Volumes persistants

```
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  clusterIP: None # Service headless
  selector:
    app: db
  ports:
    - port: 5432
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: db
spec:
  serviceName: db
  replicas: 3
  selector:
    matchLabels:
      app: db
```

```
template:
  metadata:
    labels:
      app: db
  spec:
    containers:
      - name: db
        image: postgres:17
        volumeMounts:
          - name: data
            mountPath: /var/lib/postgresql
volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
```

5.2 Volumes persistants

Le champ `volumeClaimTemplates` remplace la déclaration d'un PVC : K8s crée **un PVC par Pod** à partir du modèle (`data-db-0`, `data-db-1`, ...).

Si `db-1` est supprimé, un nouveau Pod nommé `db-1` est recréé et **retrouve son PVC**, donc ses données.

¹[la documentation officielle](#)

5.2 Volumes persistants

Le champ `volumeClaimTemplates` remplace la déclaration d'un PVC : K8s crée **un PVC par Pod** à partir du modèle (`data-db-0`, `data-db-1`, ...).

Si `db-1` est supprimé, un nouveau Pod nommé `db-1` est recréé et **retrouve son PVC**, donc ses données.

Attention

Supprimer ou réduire un StatefulSet **ne supprime pas** les PVC associés : c'est un garde-fou contre la perte de données¹, mais le ménage est à votre charge.

¹[la documentation officielle](#)

5.2 Volumes persistants



Exercices

1. Déployez le StatefulSet db ci-dessus (Service compris) et observez la création des Pods avec `kubectl get pods --watch`. Que remarquez-vous sur les noms et l'ordre ?
2. Supprimez le Pod db-1. Quel est le nom de son remplaçant ?
3. Listez les PVC. Supprimez le StatefulSet : que deviennent-ils ?



Aide aux révisions

1. Quelle est la durée de vie d'un volume `emptyDir` ?
2. Pourquoi l'usage de `hostPath` est-il déconseillé ?
3. Quelle est la différence entre un PV et un PVC ? Qui crée quoi ?
4. Qu'est-ce que le provisionnement dynamique ?
5. Dans quels cas utiliser un `StatefulSet` plutôt qu'un `Deployment` ?
6. Que deviennent les PVC d'un `StatefulSet` supprimé ?

1. Introduction à Kubernetes
2. Briques de bases
3. Premiers pas avec K8s
4. Déployer de manière déclarative
5. Persistance et données
- 6. Sécurité**

6. Sécurité

Nous avons déjà croisé des questions de sécurité dans ce cours : le dashboard à ne pas exposer sur Internet, les volumes `hostPath` à éviter...

¹<https://kubernetes.io/docs/concepts/security/cloud-native-security/>

6. Sécurité

Nous avons déjà croisé des questions de sécurité dans ce cours : le dashboard à ne pas exposer sur Internet, les volumes `hostPath` à éviter...

La sécurité d'un cluster se pense **en couches**, c'est le modèle des $4C^1$:

1. **Cloud** (ou datacenter) : l'infrastructure qui héberge le cluster;
2. **Cluster** : les composants K8s et leur configuration;
3. **Container** : les images et leurs privilèges;
4. **Code** : l'application elle-même.

Chaque couche s'appuie sur la précédente : un code parfait ne protège pas d'un cluster mal configuré.

¹<https://kubernetes.io/docs/concepts/security/cloud-native-security/>

Ce chapitre répond à quatre questions :

- où stocker les données sensibles ? → les **Secrets**;
- qui a le droit de faire quoi sur l'API ? → le **RBAC**;
- que peuvent faire les conteneurs sur leur noeud ? → les **securityContexts**;
- qui peut parler à qui sur le réseau ? → les **NetworkPolicies**.

6.1 Secrets

Au chapitre précédent, les ConfigMaps stockaient de la configuration **non sensible**. Mais où mettre un mot de passe de base de données, un token d'API, un certificat TLS ?

6.1 Secrets

Au chapitre précédent, les ConfigMaps stockaient de la configuration **non sensible**. Mais où mettre un mot de passe de base de données, un token d'API, un certificat TLS ?

Pas dans l'image (lisible par quiconque peut la télécharger), pas dans une ConfigMap. K8s propose un objet dédié : le **Secret**.

6.1 Secrets

```
apiVersion: v1
kind: Secret
metadata:
  name: db-creds
type: Opaque
data:
  password: UzNjcmV0IQ== # encodé en base64
stringData:
  username: admin          # texte brut
```

Deux champs pour les valeurs :

- data : valeurs encodées en base64;
- stringData : valeurs en clair, encodées à la création.

Attention

Le base64 est un **encodage**, pas du chiffrement : il se décode trivialement. Et par défaut, les Secrets sont stockés **en clair** dans etcd : quiconque a accès à l'API ou à etcd peut les lire¹. Un Secret est une déclaration d'intention, pas une protection en soi.

¹Les parades : chiffrement au repos d'etcd, RBAC strict, ou gestionnaire externe (HashiCorp Vault, ...). Voir [la documentation officielle](#).

6.1 Secrets

Un Pod consomme un Secret comme une ConfigMap : en variable d'environnement ou en volume.

```
containers:
- name: app
  image: postgres:17
  env:
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-creds
          key: password
```

```
containers:
- name: app
  volumeMounts:
    - name: creds
      mountPath: /etc/creds
      readOnly: true
volumes:
- name: creds
  secret:
    secretName: db-creds
```

Monté en volume, le Secret est placé dans un tmpfs (en RAM) : il n'est jamais écrit sur le disque du noeud.

6.1 Secrets

En impératif :

```
$ kubectl create secret generic db-creds --from-literal=password='S3cret!'
```

Le champ type indique l'usage du Secret : Opaque (défaut, données arbitraires),
kubernetes.io/tls (certificat + clé privée), kubernetes.io/dockerconfigjson
(authentification aux registres privés)...



Exercices

1. Créez un Secret db-creds de manière impérative, avec une clé password.
2. Affichez-le en YAML. Le mot de passe est-il protégé ?
3. Décodez-le en une seule ligne de shell.
4. Créez un Pod busybox qui expose ce Secret dans la variable d'environnement DB_PASSWORD, et vérifiez avec `kubectl exec`.

Toute requête au cluster passe par l'API Server (voir chapitre 1), qui applique deux étapes :

1. **l'authentification** : qui êtes-vous ?
2. **l'autorisation** : avez-vous le droit de faire ça ?

Toute requête au cluster passe par l'API Server (voir chapitre 1), qui applique deux étapes :

1. **l'authentification** : qui êtes-vous ?
2. **l'autorisation** : avez-vous le droit de faire ça ?

RBAC (*Role-Based Access Control*) est le mécanisme d'autorisation standard de K8s : **tout est interdit par défaut**, et des rôles ouvrent des droits.

Deux catégories d'identités¹ :

- les **utilisateur.rices** : les humains. K8s n'a **pas d'objet** « **User** » : ils sont gérés à l'extérieur du cluster (certificats clients, OIDC/SSO...);
- les **ServiceAccounts** : l'identité des applications. Ce sont des objets K8s, propres à un Namespace (chacun contient un ServiceAccount default).

¹<https://kubernetes.io/docs/reference/access-authn-authz/authentication/>

Deux catégories d'identités¹ :

- les **utilisateur.rices** : les humains. K8s n'a **pas d'objet** « User » : ils sont gérés à l'extérieur du cluster (certificats clients, OIDC/SSO...);
- les **ServiceAccounts** : l'identité des applications. Ce sont des objets K8s, propres à un Namespace (chacun contient un ServiceAccount default).

Chaque Pod est associé à un ServiceAccount, dont un token est automatiquement monté dans ses conteneurs : c'est avec cette identité qu'il parle à l'API Server.

¹<https://kubernetes.io/docs/reference/access-authn-authz/authentication/>

RBAC repose sur 4 objets :

- **Role** : un ensemble de permissions, dans un Namespace;
- **ClusterRole** : idem, mais à l'échelle du cluster (noeuds, PV, ou toutes les instances d'une ressource);
- **RoleBinding** : attribue un Role (ou un ClusterRole) à des sujets, dans un Namespace;
- **ClusterRoleBinding** : attribue un ClusterRole sur tout le cluster.

Une permission se décrit par trois listes : les groupes d'API, les ressources, et les **verbes** autorisés.

rules:

```
- apiGroups: ["" ]    # "" = API core
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

Les verbes standards : get, list, watch, create, update, patch, delete, deletecollection.

6.2 RBAC

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

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

Deux propriétés importantes :

- RBAC est **purement additif** : il n'existe pas de règle «deny». Un droit non accordé est refusé;
- des ClusterRoles sont fournis d'office : `cluster-admin` (tout), `admin` et `edit` (dans un Namespace), `view` (lecture seule).

Pour vérifier des droits, `kubectl auth can-i :`

```
$ kubectl auth can-i delete pods
yes
$ kubectl auth can-i list secrets \
  --as=system:serviceaccount:default:stagiaire
no
```

L'option `--as` permet de tester avec l'identité d'un autre sujet, ici le ServiceAccount stagiaire du Namespace default.



Exercices

1. Créez un ServiceAccount stagiaire. Peut-il lister les Pods ?
2. Créez un Role pod-reader (get, list, watch sur les Pods) et attribuez-le à stagiaire, le tout en impératif.
3. Re-testez. Et peut-il supprimer des Pods ? Pourquoi ?

6.3 securityContext

Un conteneur s'exécute avec l'utilisateur défini dans son image... très souvent root.

En cas de compromission de l'application, l'attaquant est donc root **dans** le conteneur, et l'isolation avec le noeud ne tient plus qu'au noyau Linux.

Le **securityContext** permet de restreindre les privilèges d'exécution.

6.3 securityContext

Il se déclare à deux niveaux :

- `spec.securityContext` : pour tous les conteneurs du Pod;
- `spec.containers[].securityContext` : pour un conteneur, **prioritaire** en cas de conflit.

6.3 securityContext

Les champs à connaître¹ :

- runAsUser, runAsGroup, runAsNonRoot : UID/GID des processus, et refus de démarrer en root (Pod et conteneur);
- fsGroup : GID propriétaire des volumes (Pod seulement);
- allowPrivilegeEscalation : interdit de gagner des privilèges (ex: binaires *setuid*);
- readOnlyRootFilesystem : système de fichiers racine en lecture seule;
- capabilities : ajout/retrait de *capabilities* Linux;
- privileged : accès quasi complet à l'hôte (conteneur seulement).

¹<https://kubernetes.io/docs/tasks/configure-pod-container/security-context/>

6.3 securityContext

Un Pod durci ressemble à ceci :

```
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
  containers:
    - name: app
      image: mon-app:1.2.3
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
        capabilities:
          drop: ["ALL"]
```



Attention

`privileged: true` désactive l'essentiel de l'isolation : le conteneur a un accès quasi équivalent à root sur le noeud (périphériques, modules noyau...). À réserver aux composants système qui en ont réellement besoin.

Pour ne pas vérifier ces réglages Pod par Pod, K8s définit 3 profils, les **Pod Security Standards**¹ :

- privileged : aucune restriction;
- baseline : bloque les escalades de privilèges connues (privileged, hostPath...);
- restricted : impose les bonnes pratiques de durcissement (runAsNonRoot, allowPrivilegeEscalation: false, drop: ["ALL"]...).

¹<https://kubernetes.io/docs/concepts/security/pod-security-standards/>

6.3 securityContext

Ils s'appliquent par Namespace, via la *Pod Security Admission* :

```
metadata:
```

```
  labels:
```

```
    pod-security.kubernetes.io/enforce: restricted
```

(Modes disponibles : enforce rejette, warn avertit, audit journalise.)



Exercices

1. Créez un Pod busybox (command: `["id"]`) avec `runAsUser: 1000` et `runAsGroup: 1000` au niveau du Pod. Que renvoie `kubectl logs` ?
2. Créez un Pod nginx avec `runAsNonRoot: true`. Que se passe-t-il ? Pourquoi ? (indice : `kubectl describe`)

6.4 NetworkPolicies

Par défaut, **tous les Pods peuvent communiquer avec tous les Pods**, sans restriction, y compris entre Namespaces.

6.4 NetworkPolicies

Par défaut, **tous les Pods peuvent communiquer avec tous les Pods**, sans restriction, y compris entre Namespaces.

Autrement dit : n'importe quel Pod compromis peut tenter sa chance directement sur la base de données.

Une **NetworkPolicy** est un pare-feu déclaratif :

- c'est une ressource de Namespace;
- elle sélectionne des Pods par **labels**;
- elle décrit le trafic autorisé, entrant (Ingress) et/ou sortant (Egress).

Attention

Les NetworkPolicies sont implémentées par le plugin réseau (**CNI**) du cluster. Sans plugin compatible (Calico, Cilium...), elles sont acceptées par l'API mais **n'ont aucun effet** ! C'est le cas du CNI par défaut de Minikube¹.

¹la [documentation Minikube](#) recommande `minikube start --cni calico`.

6.4 NetworkPolicies

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-allow-web
spec:
  podSelector:
    matchLabels:
      app: db
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: web
      ports:
        - protocol: TCP
          port: 5432
```

Ici : seuls les Pods portant le label `app: web` peuvent se connecter aux Pods `app: db`, et uniquement sur le port 5432.

Les sources (`from`) et destinations (`to`) peuvent être des `podSelector`, des `namespaceSelector`, ou des plages d'adresses IP (`ipBlock`).

6.4 NetworkPolicies

Dès qu'une NetworkPolicy sélectionne un Pod pour une direction, ce Pod devient **isolé** dans cette direction : seul le trafic explicitement autorisé passe.

D'où le grand classique, le *default deny* :

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {} # tous les Pods du Namespace
  policyTypes:
    - Ingress
```

Tout trafic entrant est bloqué; on rouvre ensuite au cas par cas.

Deux règles de composition :

- les politiques **s'additionnent** : le trafic autorisé est l'union de ce que chacune permet, il n'y a pas de règle «deny»;
- une connexion doit être autorisée **des deux côtés** : en Egress chez l'émetteur **et** en Ingress chez le destinataire.



Exercices

1. Recréez le cluster avec un CNI compatible : `minikube delete && minikube start --cni calico`. (Attention : tout est supprimé !)
2. Lancez un Pod web (image nginx), exposez-le sur le port 80, et vérifiez qu'un Pod test peut le joindre :

```
kubectl run test --rm -it --image=busybox -- wget -q0- --timeout=2 http://web
```
3. Appliquez la politique `default-deny-ingress` ci-dessus. Que donne le même test ?
4. Écrivez une NetworkPolicy qui rouvre l'accès au port 80 des Pods `run: web`, seulement depuis les Pods `run: test`. Re-testez.

6.5 Et les images ?

La sécurité commence avant le cluster, dès la construction des images :

- des images **minimales** (Alpine, *distroless*) : moins de binaires, moins de surface d'attaque;
- des tags **précis** (nginx:1.31.1), jamais latest : vous savez ce qui tourne;
- une instruction USER non-root dans le Dockerfile;
- un **scanner de vulnérabilités** (Trivy, Gype...) dans la CI;
- des images tirées de registres de confiance uniquement.



Aide aux révisions

1. Que protège réellement l'encodage base64 d'un Secret ?
2. Quelle est la différence entre un Role et un ClusterRole ?
3. Peut-on écrire une règle RBAC qui interdit une action ?
4. Avec quelle identité un Pod parle-t-il à l'API Server ?
5. Quels champs de securityContext empêchent un conteneur de tourner en root ?
6. Que se passe-t-il si on applique une NetworkPolicy sans CNI compatible ?
7. Un Pod est sélectionné par une NetworkPolicy Ingress : quel trafic entrant reçoit-il ?

© ⓘ ♻️ Hugo Blanc, 2026

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.
- À 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.