# Conteneurisation et Orchestration

# Partie 7 : Kubernetes

## Et Kubernetes alors ?

Si Docker Swarm est simple et efficace, pourquoi tout le monde parle de **Kubernetes** ?

- **Standard de l'industrie** : Kubernetes est devenu l'orchestrateur de conteneurs dominant, soutenu par une immense communauté et tous les grands fournisseurs de cloud (Google: GKE, Aws: EKS, Azure: AKS).
- **Extensibilité et Puissance** : Il offre un niveau de configuration, d'automatisation et de flexibilité bien supérieur à Swarm.
- **Écosystème Riche** : Un nombre impressionnant d'outils sont construits autour de Kubernetes (monitoring, logging, service mesh...).

Swarm est une excellente porte d'entrée, mais Kubernetes est la destination pour les applications complexes et à grande échelle.

# L'Histoire de Kubernetes

- **Origines chez Google** : Kubernetes (du grec "gouvernail" ou "pilote") est basé sur **Borg**, le système d'orchestration interne de Google, utilisé pendant plus de 10 ans pour gérer des milliards de conteneurs.
- **Open Source (2014)** : Google a publié Kubernetes en open-source, offrant son expérience massive de la gestion de conteneurs à la communauté.
- **La CNCF (2015)** : Kubernetes a été le projet fondateur de la **Cloud Native Computing Foundation**, une fondation qui a assuré sa gouvernance neutre et a favorisé son adoption massive.

# À quoi sert Kubernetes ?

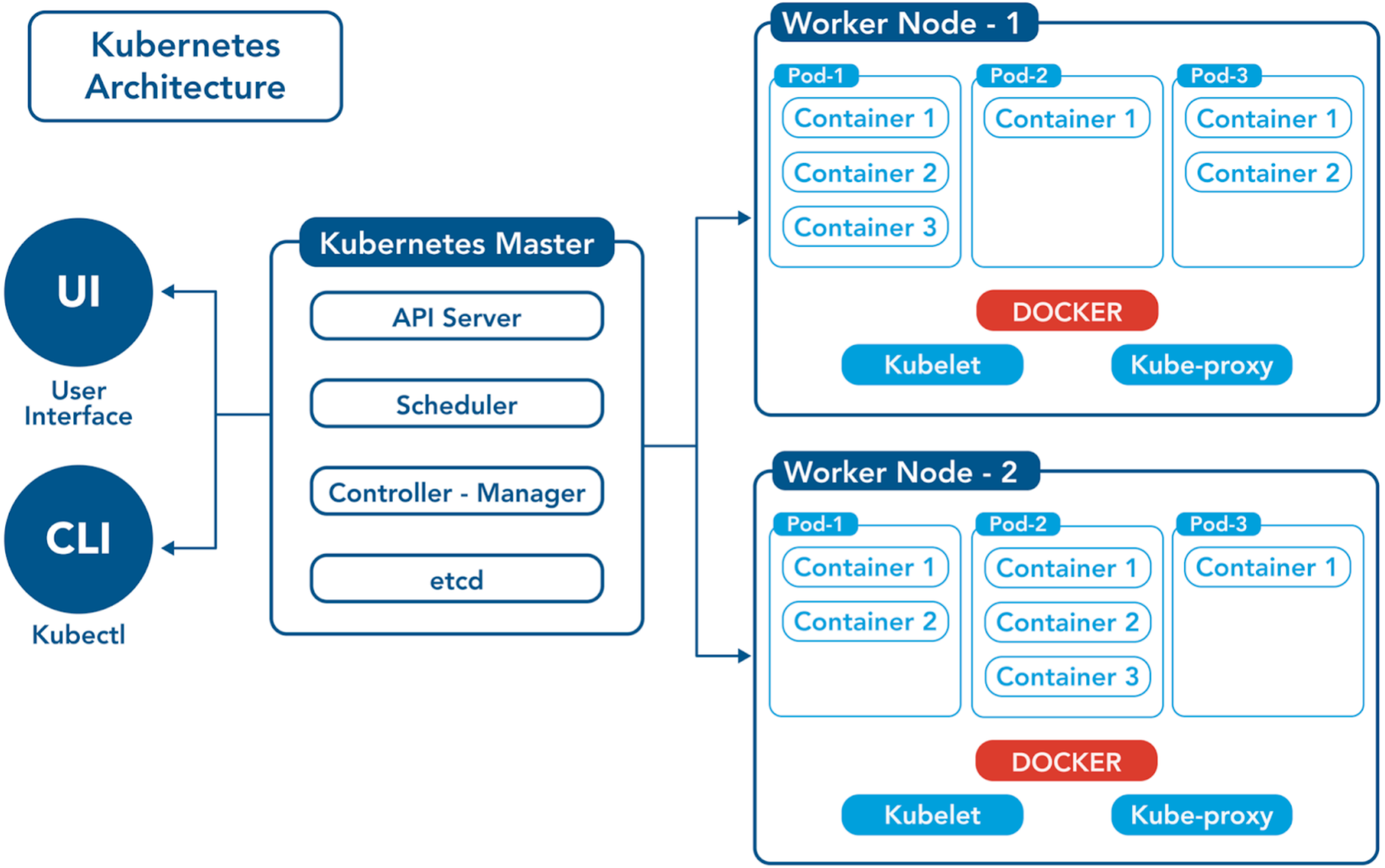
Kubernetes automatise le déploiement, la mise à l'échelle et la gestion des applications conteneurisées. Il permet de :

1. **Garantir la disponibilité** (Self-healing) : Redémarre automatiquement les conteneurs qui plantent.
2. **Simplifier la scalabilité** : Augmente ou diminue le nombre de conteneurs en fonction de la demande (manuellement ou automatiquement).
3. **Automatiser les déploiements** : Permet des mises à jour sans interruption de service (rolling updates) et des retours en arrière faciles.
4. **Optimiser les ressources** : "Case" intelligemment les conteneurs sur les nœuds du cluster pour maximiser l'utilisation des ressources.
5. **Gérer la configuration et les secrets** de manière sécurisée.

# Architecture de Kubernetes

Kubernetes fonctionne sur un modèle **client-serveur** distribué. Un cluster est composé de deux types de nœuds :

- **Control Plane (Master Nodes)** : Le **cerveau** du cluster. Il prend les décisions globales (ordonnancement, détection de pannes...).
- **Worker Nodes** : Les **muscles** du cluster. Ils exécutent les applications conteneurisées (les Pods).


Kubernetes Architecture
UI
User Interface
CLI
Kubectl
Kubernetes Master
API Server
Scheduler
Controller - Manager
etcd
Worker Node - 1
Pod-1
Container 1
Container 2
Container 3
Pod-2
Container 1
Pod-3
Container 1
Container 2
DOCKER
Kubelet
Kube-proxy
Worker Node - 2
Pod-1
Container 1
Container 2
Pod-2
Container 1
Container 2
Container 3
Pod-3
Container 1
DOCKER
Kubelet
Kube-proxy

## Le Control Plane : Le Cerveau du Cluster

Le Control Plane est responsable de la gestion de l'état du cluster. Il est composé de plusieurs services clés :

- **API Server** : Le **point d'entrée** de Kubernetes. Toutes les communications (de `kubectl`, des autres composants) passent par lui.
- **Scheduler** : **Décide** sur quel Worker Node un nouveau Pod doit être exécuté, en fonction des ressources disponibles et des contraintes.
- **Controller Manager** : Exécute des **boucles de contrôle** pour s'assurer que l'état actuel du cluster correspond à l'état désiré (ex: redémarrer un Pod en cas de crash).
- **etcd** : La **base de données** clé-valeur du cluster. Elle stocke l'état désiré et l'état actuel de toutes les ressources.

## Les Worker Nodes : Là où le travail se fait

Chaque Worker Node exécute les charges de travail applicatives. Il contient trois composants principaux :

- **Kubelet** : L'**agent** de Kubernetes sur chaque nœud. Il communique avec l'API Server et s'assure que les conteneurs décrits dans les Pods sont en cours d'exécution et en bonne santé.
- **Kube-proxy** : Gère le **réseau** sur chaque nœud. Il maintient les règles réseau qui permettent la communication vers les Pods depuis l'intérieur et l'extérieur du cluster.
- **Container Runtime** : Le **moteur** qui exécute les conteneurs. Le plus courant est `containerd`, mais Docker est aussi supporté.

## Le Flux de Déploiement : Que se passe-t-il ?

1. Un utilisateur envoie un fichier YAML à l'**API Server** via `kubectl apply`.
2. L'API Server valide et stocke la configuration dans **etcd**.
3. Le **Scheduler** voit qu'un nouveau Pod a été créé mais n'est pas assigné. Il choisit le meilleur Worker Node et met à jour l'information dans etcd via l'API Server.
4. Le **Kubelet** du nœud choisi est notifié. Il lit les informations du Pod et demande au **Container Runtime** de télécharger l'image et de démarrer le conteneur.
5. Le **Controller Manager** surveille en permanence. Si un Pod s'arrête, il prend des mesures pour le remplacer et maintenir l'état désiré.

# Les Ressources Kubernetes

# Les Namespaces : Isoler les Environnements

Un **Namespace** est un "cluster virtuel" à l'intérieur de votre cluster Kubernetes.

- **Isolation** : Permet de séparer les ressources (Pods, Services...) par projet, équipe ou environnement (ex: `dev`, `staging`, `prod`).
- **Organisation** : Évite les conflits de noms entre les différentes équipes.
- **Gestion des Accès** : On peut définir des droits spécifiques (RBAC) pour chaque Namespace.
- **Quotas de ressources** : Limiter la consommation de CPU et de RAM par Namespace.

## Les Namespaces par Défaut

Un cluster Kubernetes est livré avec trois namespaces initiaux :

- `default` : Le namespace par défaut pour les objets sans autre namespace spécifié.
- `kube-system` : Utilisé pour les objets créés par le système Kubernetes lui-même (Control Plane, CoreDNS, etc.). **Ne pas y toucher !**
- `kube-public` : Ce namespace est lisible par tous les utilisateurs (y compris non authentifiés) il est principalement réservé à un usage interne au cluster.

# Gestion des Namespaces

## Créer un Namespace (déclaratif) :

```
apiVersion: v1 # Version de l'API
kind: Namespace # Type de ressource
metadata:
  name: mon-namespace # Nom du Namespace
```

## Deployer un Namespace :

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

## Lister les Namespaces :

```
kubectl get namespaces
```

## Supprimer un Namespace :

```
kubectl delete namespace mon-namespace
```

# Interagir avec les Namespaces

### Créer une ressource dans un Namespace :

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

### Lister les Pods dans un Namespace :

```
kubectl get pods -n mon-namespace
```

### Changer le contexte par défaut :

```
kubectl config set-context --current --namespace=mon-namespace
```

## Les Pods : La Plus Petite Unité

Un **Pod** est la plus petite unité de déploiement dans Kubernetes. C'est une enveloppe logique pour un ou plusieurs conteneurs.

- **Un ou Plusieurs Conteneurs** : Un Pod encapsule un ou plusieurs conteneurs (ex: une application et son "sidecar" de logging).
- **Réseau et Stockage Partagés** : Les conteneurs d'un même Pod partagent la même adresse IP et peuvent communiquer via `localhost`. Ils peuvent aussi partager des volumes de stockage.
- **Éphémère** : Les Pods sont considérés comme jetables. On ne les gère jamais directement, mais via des contrôleurs comme les Deployments.

# Gestion des Pods : Création

## Créer un Pod (déclaratif) :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mon-pod
spec: # Spécification du Pod
  containers: # Liste des conteneurs
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80 # Port du conteneur
```

# Gestion des Pods : Inspection

## Deployer un Pod :

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

## Lister les Pods :

```
kubectl get pods
```

## Obtenir des informations détaillées :

```
kubectl describe pod mon-pod
```

## Voir les logs :

```
kubectl logs mon-pod
```

## Exécuter une commande dans un Pod :

```
kubectl exec -it mon-pod -- bash
```

## Les Services : Exposer les Pods

Un **Service** expose un groupe de Pods via une adresse IP et un nom DNS stables. Il sert de **load balancer** interne.

- **Découplage** : Les Pods peuvent être créés ou détruits, leur IP change, mais l'IP du Service reste stable.
- **Sélection par Labels** : Un Service utilise des `labels` pour identifier les Pods qu'il doit cibler.

## Les Types de Services

- **`ClusterIP`** : Expose le Service sur une `IP interne` au cluster. C'est le type `par défaut`, idéal pour la communication entre microservices.
- **`NodePort`** : Expose le Service sur un port statique sur chaque nœud du cluster (`<IP_Nœud>:<NodePort>`). Utile pour le développement ou pour exposer un service sans load balancer cloud.
- **`LoadBalancer`** : Crée un load balancer externe chez un fournisseur cloud (AWS, GCP, Azure...). C'est la méthode standard pour exposer une application sur Internet.
- **`ExternalName`** : Mappe le Service à un nom DNS externe (ex: `my.database.example.com`), agissant comme un alias.

# Gestion des Services

## Créer un Service (déclaratif) :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mon-service
spec:
  selector: # Cible les Pods avec ce label
    app: mon-app
  ports:
    - protocol: TCP
      port: 80 # Port exposé par le Service
      targetPort: 8080 # Port cible sur les Pods
  type: ClusterIP
```

# Gestion des Services

## Deployer un Service :

```
kubectl apply -f service.yaml
```

## Lister et inspecter les Services :

```
kubectl get services
```

```
kubectl describe service mon-service
```

## Supprimer un Service :

```
kubectl delete service mon-service
```

## Les ReplicaSets : Garantir le Nombre de Pods

Un **ReplicaSet** s'assure qu'un nombre spécifié de réplicas d'un Pod sont toujours en cours d'exécution.

- **Haute Disponibilité** : Si un Pod tombe en panne, le ReplicaSet en crée un nouveau automatiquement.
- **Scalabilité** : Permet de mettre à l'échelle manuellement le nombre de Pods.

**Note :** On utilise rarement les ReplicaSets directement. On préfère les **Deployments**, qui les gèrent pour nous de manière plus évoluée (gestion des mises à jour, etc.).

# Gestion des ReplicaSets

## Créer un ReplicaSet (déclaratif) :

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mon-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mon-app
  template:
    # ... template de Pod ...
```

## Le Template de Pod

Le `template` est la section la plus importante d'un contrôleur. C'est le **modèle** qui sera utilisé pour créer chaque Pod. Il contient la `metadata` (avec les `labels`) et la `spec` (avec les `containers`) du Pod.

**Lien Clé :** Le `selector` du contrôleur doit correspondre aux `labels` définis dans le `template`.

# Le Template de Pod

```yaml
spec:
  replicas: 3 # On crée 3 pods basés sur le template
  selector: # 1. Le contrôleur cherche les Pods avec ce label
    matchLabels:
      app: mon-app
  template:   # 2. Il utilise ce template pour les créer
    metadata:
      labels: # 3. Le label du Pod DOIT correspondre au sélecteur
        app: mon-app
    spec:     # 4. C'est la spécification exacte du Pod (conteneurs, etc.)
      containers:
      - name: nginx
        image: nginx
```

# Gestion des ReplicaSets : Création

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mon-replicaset
spec:
  replicas: 3 # Nombre de Pods souhaités
  selector:
    matchLabels:
      app: mon-app # Le label pour trouver les Pods à gérer
  template:
    metadata:
      labels:
        app: mon-app # Le label appliqué aux Pods créés
    spec:
      containers:
      - name: nginx # Nom du conteneur
        image: nginx # Image docker utilisé
        ports:
        - containerPort: 80
```

# Gestion des ReplicaSets : Création

## Deployer un ReplicaSet :

```
kubectl apply -f replicaset.yaml
```

## Lister et inspecter les ReplicaSets :

```
kubectl get replicasets
```

```
kubectl describe replicasets mon-replicaset
```

## Scaler un ReplicaSet :

```
kubectl scale replicaset mon-replicaset --replicas=5
```

## Supprimer un ReplicaSet :

```
kubectl delete replicasets mon-replicaset
```

## Les Deployments : Gérer le Cycle de Vie

Un **Deployment** est un contrôleur de haut niveau qui gère les **ReplicaSets** et les **Pods**. C'est la ressource la plus courante pour déployer des applications stateless.

- **Mises à Jour sans Interruption (Rolling Updates)** : Met à jour les Pods progressivement, en remplaçant les anciennes versions par les nouvelles sans couper le service.
- **Rollbacks** : Permet de revenir facilement à une version précédente en cas de problème.
- **Scalabilité Déclarative** : On définit l'état désiré (ex: "je veux 5 réplicas de mon app"), et le Deployment s'occupe du reste.

# Gestion des Deployments : Création

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mon-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mon-app
  template:
    metadata:
      labels:
        app: mon-app
    spec:
      containers:
      - name: nginx
        image: nginx:1.21.6
        ports:
        - containerPort: 80
```

# Gestion des Deployments : Création

## Deployer un Deployment :

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

## Scaler un Deployment :

```
kubectl scale deployment mon-deployment --replicas=5
```

## Lister et inspecter les Deployments :

```
kubectl get deployments
```

```
kubectl describe deployments mon-deployment
```

## Supprimer un Deployment :

```
kubectl delete deployments mon-deployment
```

# Gestion des Deployments : Mises à Jour

## Mettre à jour l'image (Rolling Update) :

```
kubectl set image deployment/mon-deployment nginx=nginx:1.22.0
```

## Suivre le statut du déploiement :

```
kubectl rollout status deployment/mon-deployment
```

## Voir l'historique et annuler un déploiement :

```
kubectl rollout history deployment/mon-deployment
```

```
kubectl rollout undo deployment/mon-deployment
```

## Redémarrer un déploiement :

```
kubectl rollout restart deployment/mon-deployment
```

## ConfigMaps : Gérer la Configuration

Un **ConfigMap** permet de stocker des données de configuration `non sensibles` (variables d'environnement, arguments de ligne de commande, fichiers de configuration) en dehors de l'image de votre conteneur.

- **Découplage** : Sépare la configuration du code de l'application.
- **Portabilité** : Facilite l'utilisation de la même image d'application dans différents environnements (dev, staging, prod).

# Gestion des ConfigMaps : Création

## Créer un ConfigMap (déclaratif) :

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mon-configmap
data: # Données de configuration
  MA_CLE: "ma_valeur"
  NODE_ENV: "production"
```

# Gestion des ConfigMaps : Création

## Deployer un ConfigMap :

```
kubectl apply -f configmap.yaml
```

## Lister et inspecter les ConfigMaps :

```
kubectl get configmaps
```

```
kubectl describe configmap mon-configmap
```

## Supprimer un ConfigMap :

```
kubectl delete configmaps mon-configmap
```

# Utiliser un ConfigMap

## Comme variable d'environnement (dans un Pod) :

```yaml
# ... spec.containers ...
env:
  - name: MA_VARIABLE
    valueFrom:
      configMapKeyRef:
        name: mon-configmap
        key: MA_CLE
```

## Comme fichier dans un volume (dans un Pod) :

```yaml
# ... spec.volumes ...
volumes:
  - name: config-volume
    configMap:
      name: mon-configmap
```

## Secrets : Gérer les Données Sensibles

Un **Secret** est similaire à un ConfigMap, mais il est conçu pour stocker des données sensibles.

- **Données sensibles** : Mots de passe, clés API, certificats TLS.
- **Stockage** : Les données sont stockées en base64 dans `etcd`. **Attention, base64 n'est pas un chiffrement !** Il est recommandé d'utiliser des mécanismes de chiffrement au repos pour `etcd` et des solutions comme HashiCorp Vault ou un secret manager (fournit par le provider) pour une sécurité renforcée.

# Gestion des Secrets : Création

**Créer un Secret :**

Les valeurs doivent être encodées en base64.

`echo -n 'mon_mot_de_passe_secret' | base64`

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mon-secret
type: Opaque # Type de secret par défaut
data: # Données encodées en base64
  DB_PASSWORD: bW9uX21vdF9kZV9wYXNzZV9zZWNyZXQ=
```

## Gestion des Secrets : Création (Impératif)

On peut aussi créer des Secrets en ligne de commande de manière imperative, ce qui est très pratique pour intégrer des fichiers.

**Depuis un fichier `.env` :**

```
# Fichier .env:
# DB_USER=admin
# DB_PASSWORD=supersecret
```

```
kubectl create secret generic mon-secret-env --from-env-file=.env
```

# Impératif vs. Déclaratif : Deux Approches

Il y a deux manières principales de gérer les ressources Kubernetes :

- **Approche Impérative** : Vous donnez des **ordres directs** à Kubernetes.
  - `kubectl run nginx --image=nginx` ("Exécute un conteneur nginx").
  - C'est simple et rapide pour des tests, mais difficile à reproduire et à suivre.
- **Approche Déclarative** : Vous décrivez l'**état final souhaité** dans un fichier YAML.
  - `kubectl apply -f nginx-pod.yaml` ("Assure-toi que l'état du cluster correspond à ce fichier").
  - Kubernetes se charge de faire correspondre la réalité à votre déclaration.

## Pourquoi l'Approche Déclarative est Préférée

L'approche déclarative est le **standard de l'industrie** pour la gestion de Kubernetes en production.

- **Infrastructure as Code (IaC)** : Les fichiers YAML peuvent être stockés dans **Git**, ce qui permet le versioning, les revues de code et le suivi des modifications.
- **Reproductibilité** : Garantit que les déploiements sont cohérents entre les environnements (dev, staging, prod).
- **Automatisation (GitOps)** : S'intègre parfaitement dans les pipelines CI/CD. Une modification dans Git peut déclencher automatiquement un `kubectl apply`.
- **Source de Vérité** : Le dépôt Git devient la source de vérité unique de l'état de votre infrastructure.
- **Auto-réparation** : Si quelqu'un supprime une ressource, la boucle de réconciliation de Kubernetes la ramènera à l'état déclaré dans le fichier YAML (voir aussi ArgoCD).

# Gestion des Secrets : Création

## Deployer un Secret :

```
kubectl apply -f secret.yaml
```

## Lister et inspecter les Secrets :

```
kubectl get secrets
```

```
kubectl describe secret mon-secret
```

## Supprimer un Secret :

```
kubectl delete secrets mon-secret
```

# Utiliser un Secret

## Comme variable d'environnement (dans un Pod) :

```yaml
# ... spec.containers ...
env:
  - name: DATABASE_PASSWORD
    valueFrom:
      secretKeyRef:
        name: mon-secret
        key: DB_PASSWORD
```

## Comme fichier dans un volume (dans un Pod) :

```yaml
# ... spec.volumes ...
volumes:
- name: secret-volume
  secret:
    secretName: mon-secret
```

## Jobs & CronJobs : Exécuter des Tâches

- **Job** : Crée un ou plusieurs Pods pour exécuter une **tâche ponctuelle** qui se termine (ex: une migration de base de données, un traitement batch). Le Job s'assure que la tâche se termine avec succès.
- **CronJob** : Permet de planifier l'exécution d'un **Job à intervalles réguliers**, en utilisant la syntaxe `cron` (ex: une sauvegarde toutes les nuits à 2h du matin).

# Gestion des Jobs

## Créer un Job (déclaratif) :

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: mon-job
spec:
  template: # Template de Pod pour le Job
    spec:
      containers:
      - name: etl-processor
        image: python:3.9-slim
        command: ["python", "-c", "import time; print('Run script...'); time.sleep(15); print('Script success.')"]
      restartPolicy: Never # Ne jamais redémarrer le conteneur, le Job créera un nouveau Pod
  backoffLimit: 4 # 4 tentatives avant de marquer le Job comme échoué
```

# Gestion des Jobs

## Deployer un Job :

```
kubectl apply -f job.yaml
```

## Lister et inspecter les Jobs :

```
kubectl get jobs
```

```
kubectl describe job mon-job
```

## Supprimer un Job :

```
kubectl delete jobs mon-job
```

# Gestion des CronJobs

## Créer un CronJob (toutes les minutes) :

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: mon-cronjob
spec:
  schedule: "*/1 * * * *" # Syntaxe Cron: toutes les minutes
  jobTemplate: # Modèle du Job à créer
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            command: ["/bin/sh", "-c", "date; echo 'Hello!'"]
          restartPolicy: OnFailure
```

## Conclusion

Kubernetes est l'orchestrateur de conteneurs **standard de l'industrie**, offrant une puissance et une flexibilité inégalées pour les applications modernes.

- **Architecture Robuste** : La séparation entre le **Control Plane** (le cerveau) et les **Worker Nodes** (les muscles) garantit la résilience.
- **Approche Déclarative** : En décrivant l'état souhaité dans des fichiers **YAML**, vous bénéficiez de l'**Infrastructure as Code**, de l'automatisation et de l'auto-réparation.
- **Écosystème Riche** : La maîtrise de ses objets fondamentaux (Pods, Services, Deployments...) est la clé pour déployer, scaler et gérer des applications complexes en production.

Bien que sa courbe d'apprentissage soit raide, sa puissance en fait un outil incontournable pour le Cloud Native.