paul.millet@lameduse.fr

LaMeDuSe

UBE : Kubernetes, mise en œuvre

Version : 10/06/2024

LaMeDuSe

Prérequis

- Connaissances dans l’administration de système Linux
- Connaissances générales en conteneurisation

Objectifs

- Comprendre le positionnement de Kubernetes et la notion d’orchestration
- Installer Kubernetes et ses différents composants
- Utiliser les fichiers descriptifs YAML
- Définir les bonnes pratiques pour travailler avec Kubernetes

Tour de table

- Présentation des membres
- Vos attentes vis-à-vis de la formation

Programme de la formation

1. Introduction à Kubernetes
2. Fichiers descriptifs
3. Architecture de Kubernetes
4. Exploiter Kubernetes
5. Kubernetes en production
6. Déploiment d’un cluster Kubernetes
7. Ressources des TP
8. Annexe & Ressources connexes

paul.millet@lameduse.fr

LaMeDuSe

UBE : Kubernetes, mise en œuvre

Introduction à Kubernetes

1. De la virtualisation à la conteneurisation - Docker & Kubernetes.
2. Installation de Kubernetes
3. Accéder au cluster Kubernetes
4. Déploiement et publication manuelle
5. Détail et introspection du déploiement
6. TP : Déploiement d’une plateforme de test

Un orchestrateur c’est quoi ?

- Coordination, gestion et automatisation
  - de la configuration (pré-requis, paramètres...)
  - du déploiement (allocation, provisionnement...)
  - de la maintenance (mise à jour, pannes...)

=> Augmentation de l'efficacité opérationnelle

\+ réduction des erreurs

Etude de cas : “Borg, L’orchestrateur de Google”

- Google utilisait Borg depuis les années 2000
- Problèmes de Borg
  - Très adapté aux besoins internes / difficile à adapter pour d’autres cas d’usage => Pas d’utilisateur extérieur.
  - Complexité de gestion : Exploitation difficile par des équipes non formée et courbe d’apprentissage raide.
  - Pas / peu standardisée.
  - Conception monolithique : Une base de code pour plusieurs fonctions métier.

Problème : Complexité d’utilisation, de maintenance, et d’évolution.

Kubernetes : La solution de Google

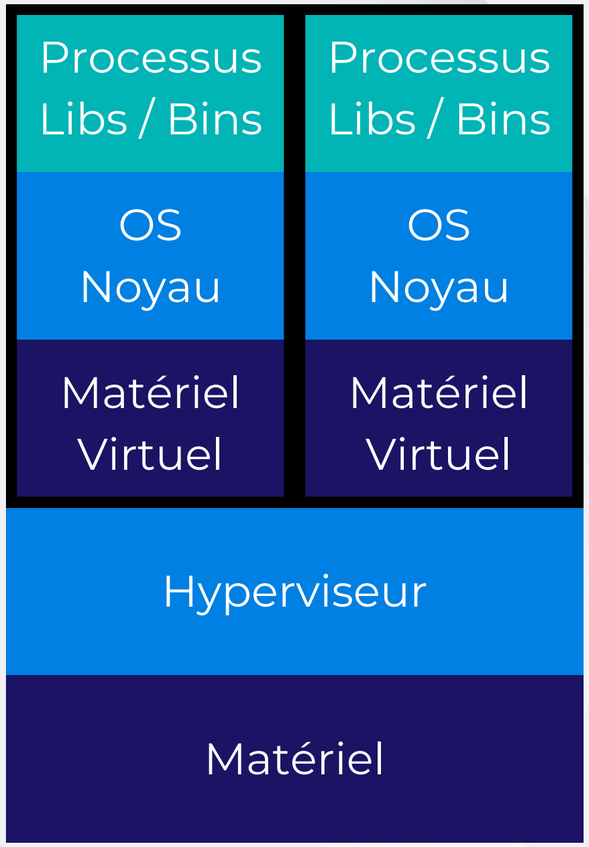
- Automatisation
- Standardisation des interfaces
- Standardisation des composants
- Open-Source = Utilisation par des entreprises autres que Google
- Faible courbe d’apprentissage
- Bonne documentation
- Séparation des composants : Conception en micro-service

=> Facile à mettre en place et à maintenir

Qui développe Kubernetes

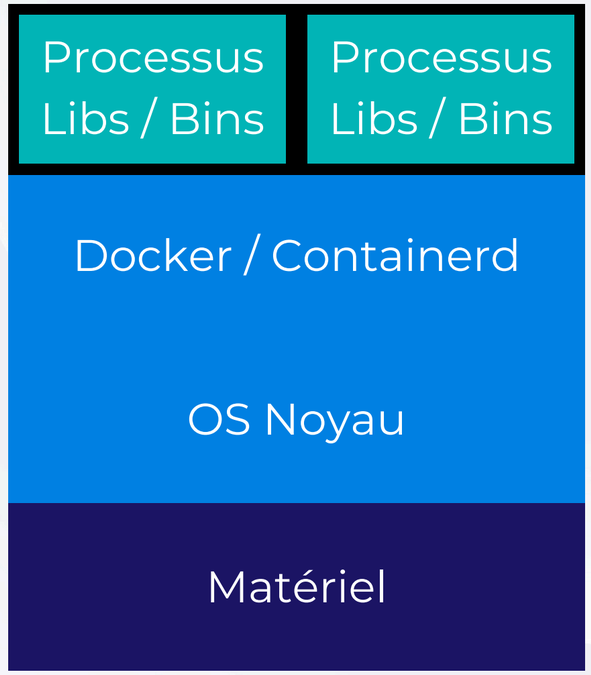
Google + Red Hat + CoreOS

Virtualisation et Conteneurisation



Virtualisation

Stockage persistant (Statefull)	Stockage non persistant par défaut (Stateless)
Séparation des fichiers et librairie	Séparation des fichiers et librairie
Système d'exploitation indépendant	Utilise les ressources du système d'exploitation (kernel)
Matériel virtualisé	Pas de virtualisation matérielle
Consommation importante Isolation totale	Consommation réduite Isolation partielle



Conteneurisation

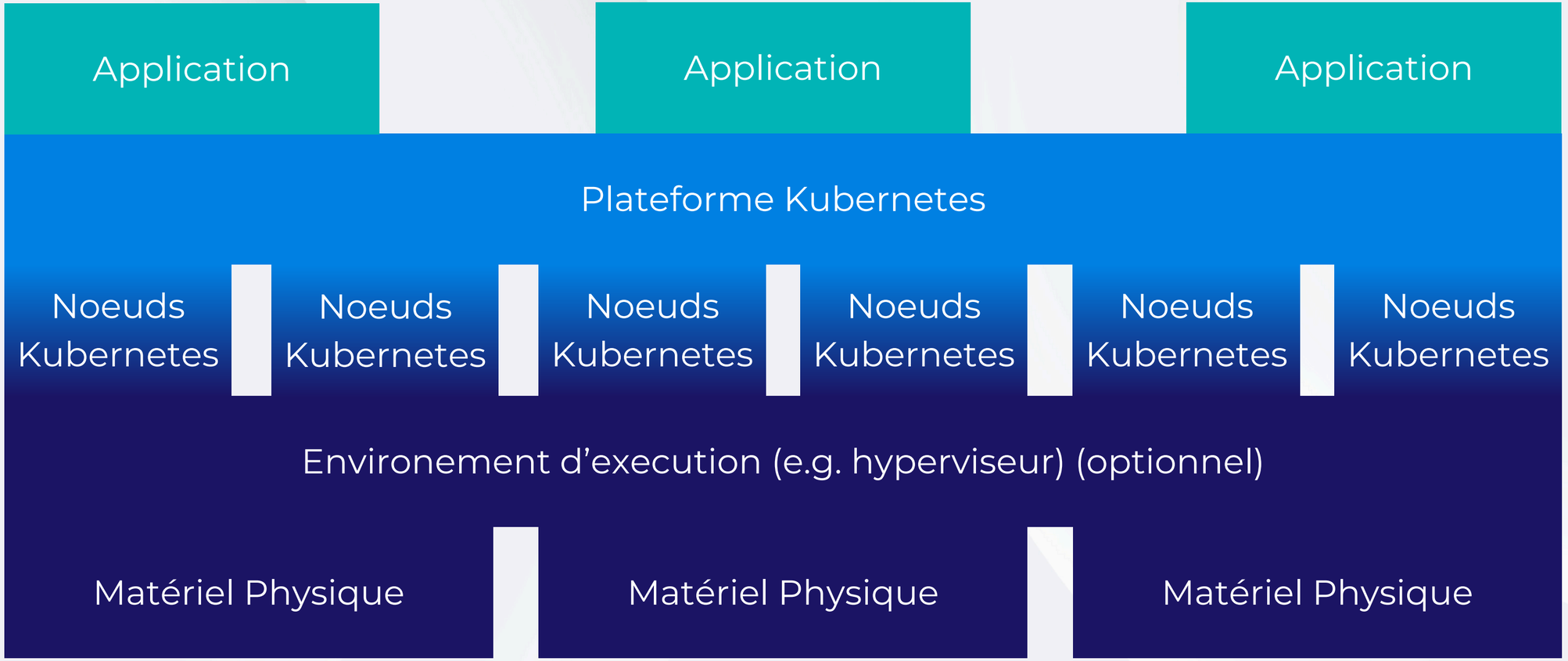
Solutions d'installation de kubernetes

	Développement Local	**Production Cloud Public**	**Production Cloud Privé (3rd party)**	**Production Cloud Privé (Sans 3rd party)**
Solutions	MiniKube	Google Kubernetes Engine (AWS) Elastic Kubernetes Engine Azure Kubernetes Service	Rancher Kubernetes Engine 2 K3s Open-Shift	KubeAdm
Avantages	Simple d'utilisation	Simple d'utilisation Support commercial	Simple d'utilisation Support commercial	Customisable No vendor lock-in
Inconvénient	Non utilisable en production	Onéreux Vendor Lock-In	Vendor Lock-in	Complexité d'utilisation Support communautaire

Installation de Docker (debian)

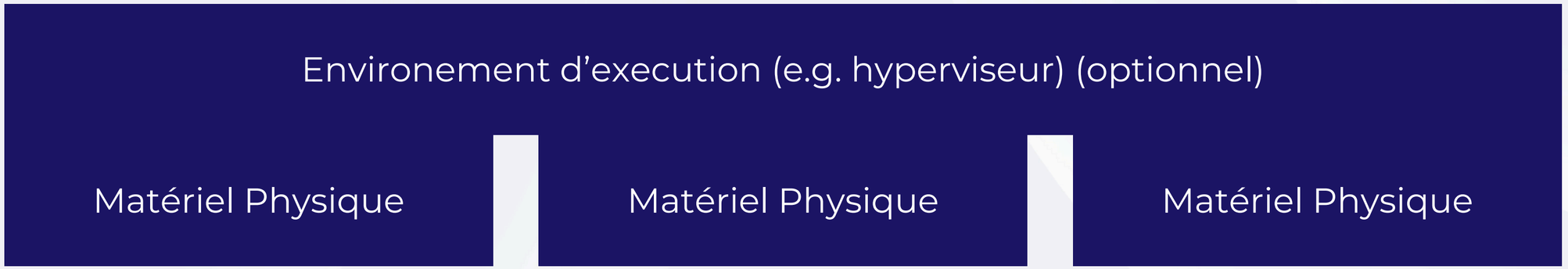
```
# Téléchargement du script d'installation de docker
$ curl -fsSL https://get.docker.com -o install-docker.sh
# Vérification du script
$ cat install-docker.sh
# Lancement du script
$ sudo sh install-docker.sh
# Vérification de l'installation
$ systemctl status docker
# Activation du service au démarage
$ systemctl enable docker
```

Niveau d’abstraction vue globale



1.ajout_1

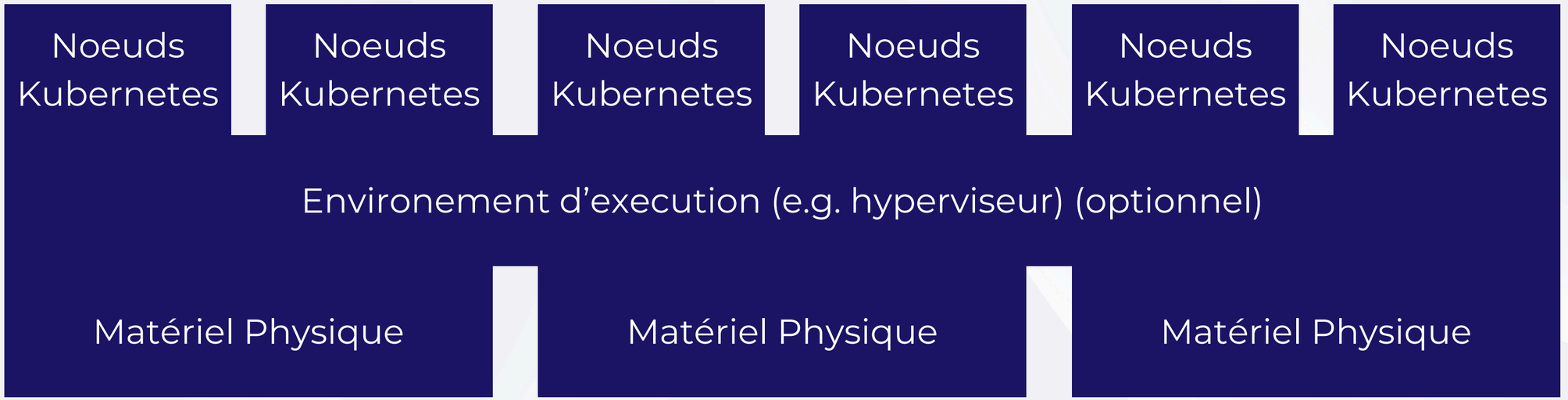
Niveau d’abstraction vue physique



1.ajout_1

LaMeDuSe

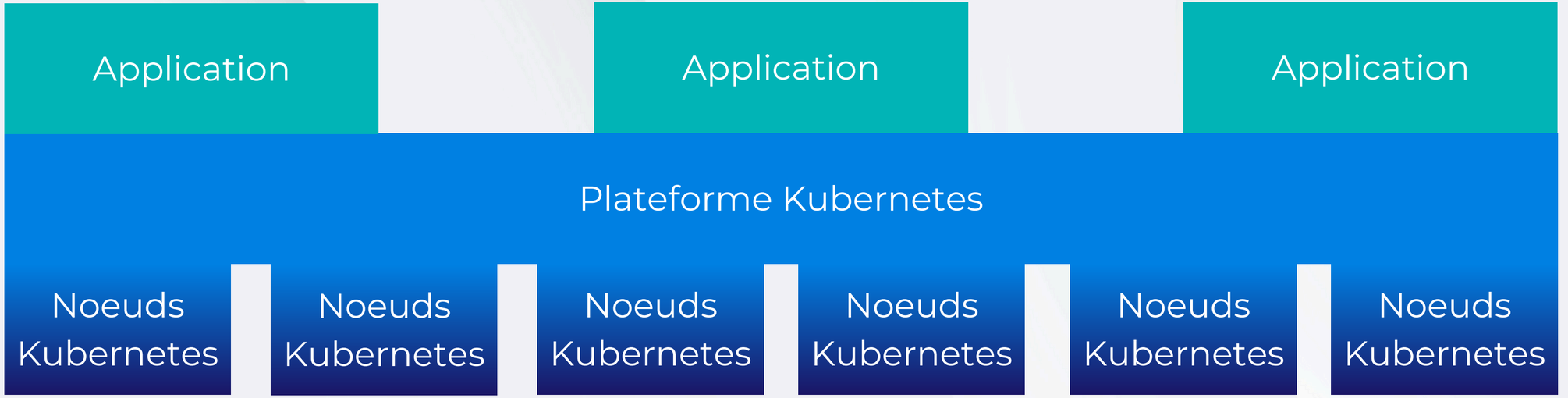
Niveau d’abstraction vue plateforme d’excution



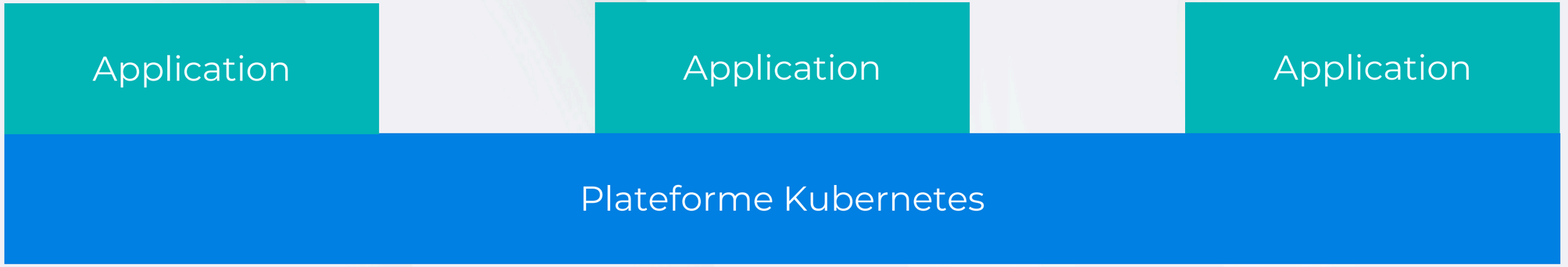
1.ajout_1

LaMeDuSe

Niveau d’abstraction vue cloud



Niveau d’abstraction vue applicative



Accès au cluster kubernetes : CLI (Kubectl)

- kubectl est l'outil en ligne de commande pour interagir avec les clusters Kubernetes.
- Il vous permet de déployer et de gérer des applications sur Kubernetes ainsi que de visualiser et diagnostiquer les ressources et les clusters.

Accès au cluster kubernetes : CLI (Kubectl)

- kubectl get <type_ressource> [nom_ressource] :
  - Affiche les informations sur les ressources Kubernetes. Si aucun nom n'est spécifié, toutes les ressources de ce type seront affichées.
  - Exemples :
    - kubectl get pods - Liste tous les pods.
    - kubectl get services - Liste tous les services.

Accès au cluster kubernetes : CLI (Kubectl)

- kubectl describe <type_ressource> <nom_ressource> :
  - Affiche des informations détaillées sur une ressource spécifique, y compris ses événements et son état actuel.
  - Exemple :
    - kubectl describe pod my-pod - Affiche les détails du pod nommé "my-pod".

Accès au cluster kubernetes : CLI (Kubectl)

- kubectl create -f <fichier_yaml> :
  - Crée une ressource Kubernetes à partir d'un fichier de configuration YAML.
    - Exemple :
      - kubectl create -f deployment.yaml - Crée une ressource à partir du fichier deployment.yaml.

Accès au cluster kubernetes : CLI (Kubectl)

- kubectl apply -f <fichier_yaml> :
  - Applique les modifications définies dans un fichier YAML pour mettre à jour les ressources existantes ou en créer de nouvelles.
  - Exemple :
    - kubectl apply -f service.yaml - Applique les configurations du fichier service.yaml.

Accès au cluster kubernetes : CLI (Kubectl)

- kubectl delete <type_ressource> <nom_ressource> :
  - Supprime une ressource spécifique. Vous pouvez également utiliser un fichier YAML pour supprimer des ressources.
  - Exemples :
    - kubectl delete pod my-pod - Supprime le pod nommé "my-pod".
    - kubectl delete -f deployment.yaml - Supprime les ressources définies dans le fichier deployment.yaml.

Accès au cluster kubernetes : CLI (Kubectl)

- kubectl logs <nom_pod> [-c <nom_container>] :
  - Affiche les logs du pod spécifié. Vous pouvez également spécifier un conteneur particulier dans un pod multi-containers.
  - Exemple :
    - kubectl logs my-pod - Affiche les logs du pod nommé "my-pod".
    - kubectl logs my-pod -c my-container - Affiche les logs du conteneur nommé "my-container" dans le pod "my-pod".

Accès au cluster kubernetes : CLI (Kubectl)

- kubectl exec -it <nom_pod> -- <commande> :
  - Exécute une commande dans un conteneur en cours d'exécution. Utilisez -it pour obtenir un terminal interactif.
  - Exemple :
    - kubectl exec -it my-pod -- /bin/sh - Ouvre un shell interactif dans le pod nommé "my-pod".
  - Note :
    - Option --container pour spécifier le container

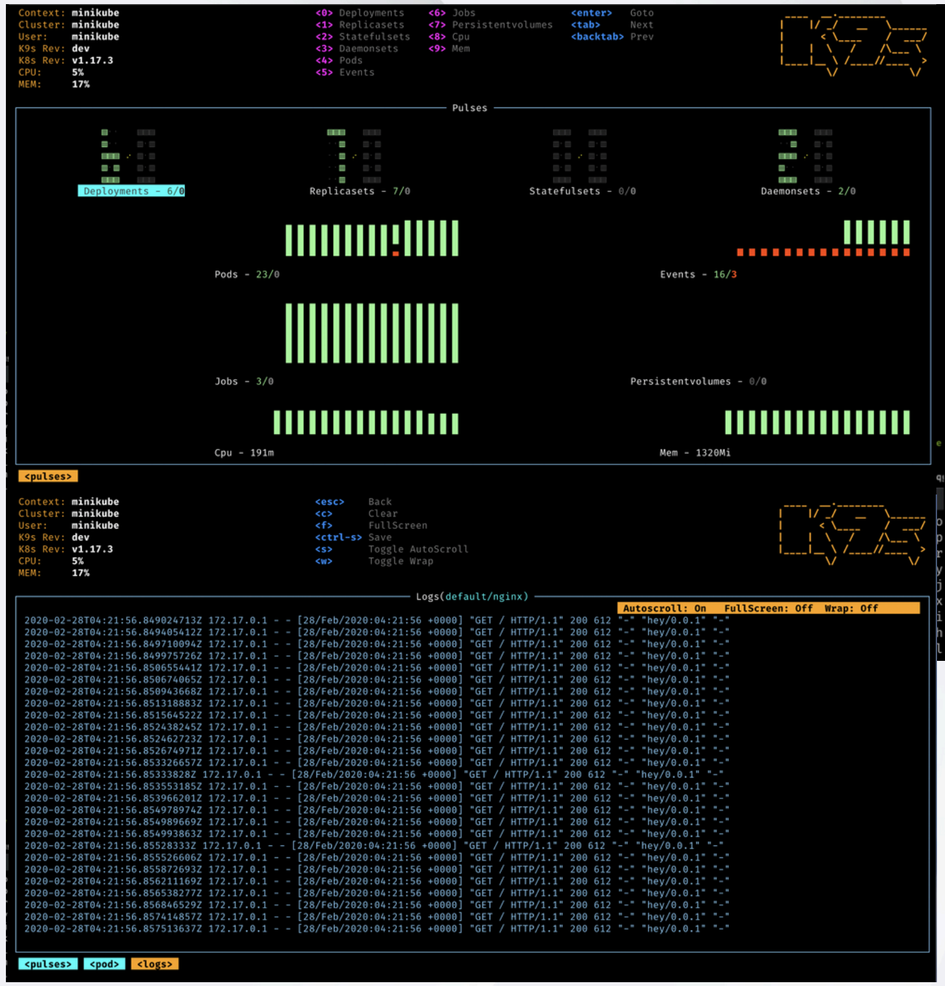
Accès au cluster kubernetes : CLI (Kubectl)

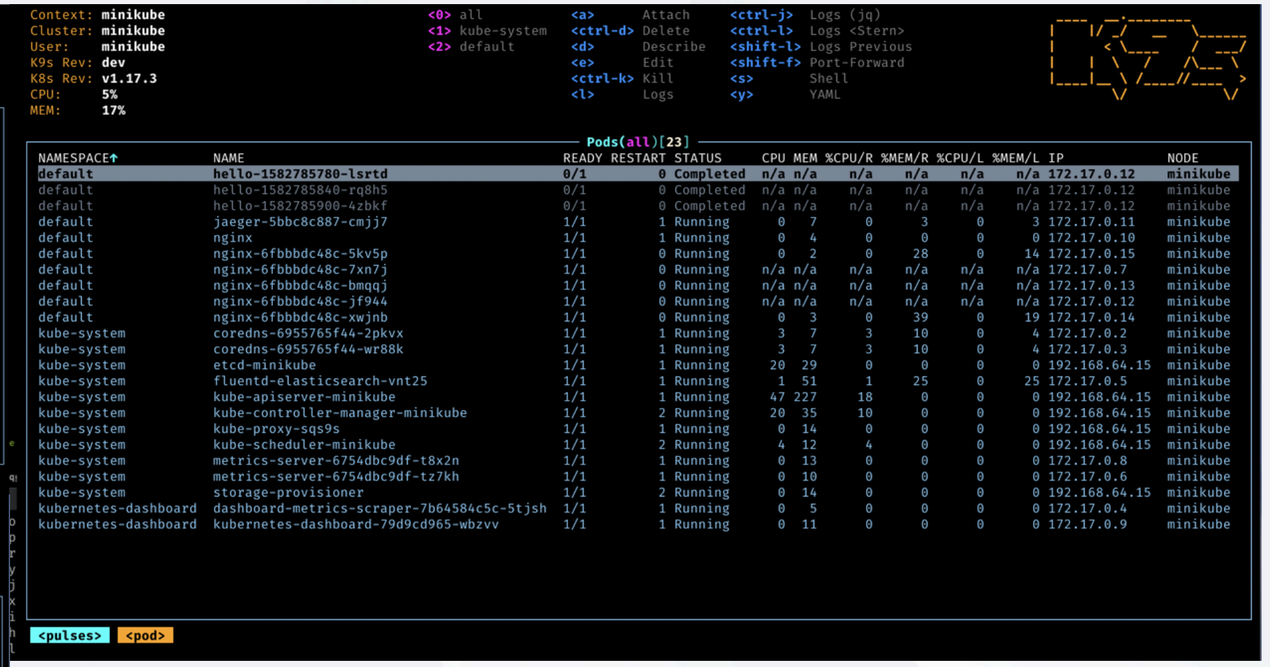
- kubectl cluster-info :
  - Affiche des informations sur les services principaux du cluster, y compris l'URL de l'API et des composants comme le dashboard.
  - Exemple :
    - kubectl cluster-info - Affiche des informations sur le cluster Kubernetes.

Accès au cluster kubernetes : CLI (Kubectl)

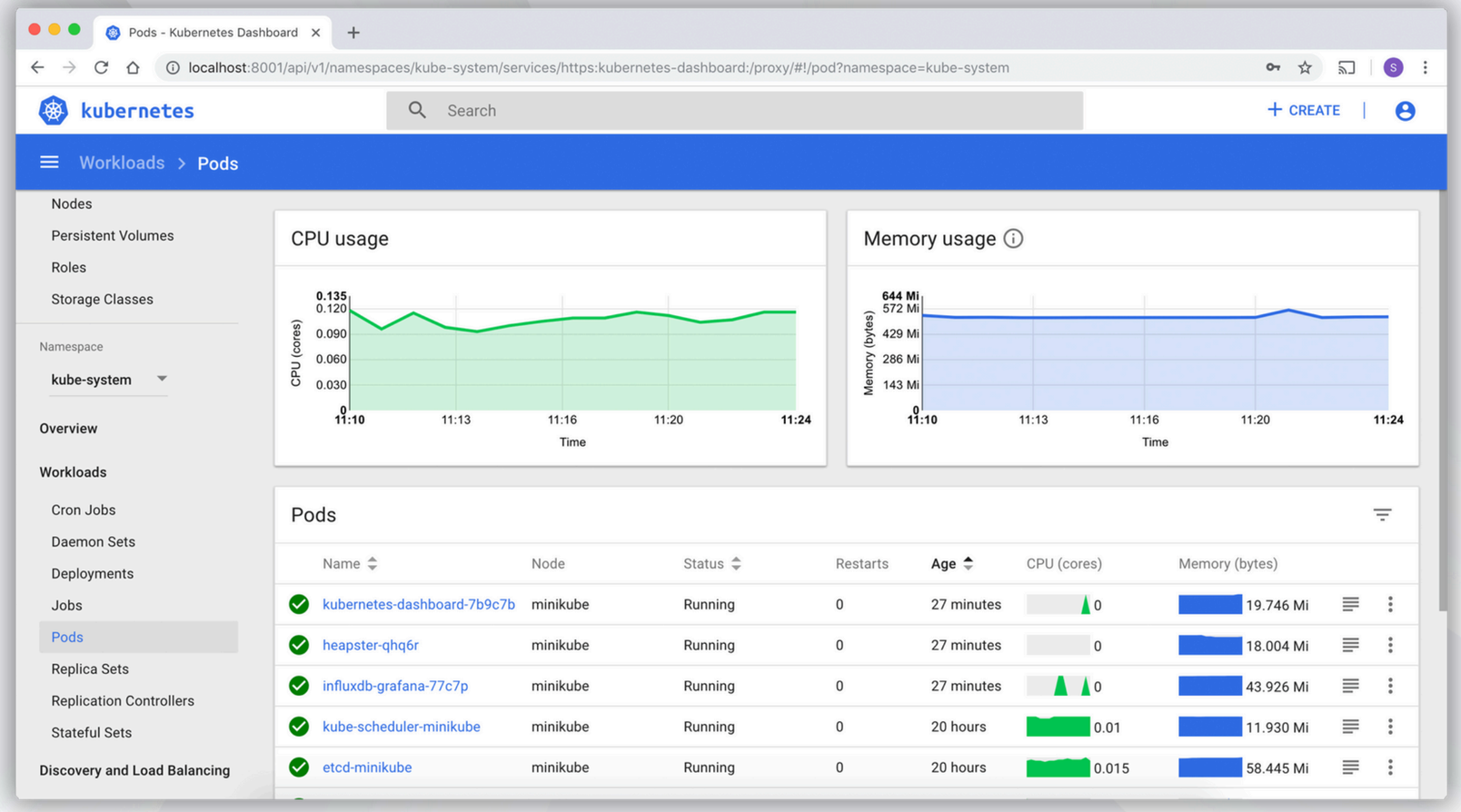
- kubectl config current-context :
  - Affiche le contexte Kubernetes actuellement utilisé. Le contexte détermine le cluster, l'utilisateur et le namespace par défaut.
  - Exemple :
    - kubectl config current-context - Affiche le contexte actuel configuré dans kubectl

Accès au cluster kubernetes : CLI (K9s)





Accès au cluster kubernetes : GUI (Dashboard)



Accès au cluster kubernetes : APIs

- L'API de Kubernetes est le cœur du système Kubernetes.
- Elle permet aux utilisateurs, applications et outils de communiquer avec le cluster.
- Toutes les interactions avec le cluster, que ce soit via kubectl, des outils tiers ou des services internes, passent par cette API.

Accès au cluster kubernetes : APIs

- L'API expose différents types de ressources qui représentent des objets dans le cluster, tels que des Pods, Services, Deployments, ConfigMaps, etc.
- Chaque ressource a une représentation en JSON ou YAML, avec des champs spécifiques pour configurer son état et son comportement.

Accès au cluster kubernetes : APIs

- L'API de Kubernetes est organisée en groupes et versions pour assurer la compatibilité et l'évolution du système.
  - core/v1 : Le groupe API de base contenant les ressources fondamentales (ex : Pods, Services).
  - apps/v1 : Contient des ressources pour les déploiements, les DaemonSets, etc.
  - batch/v1 : Utilisé pour les Jobs et CronJobs.
- Chaque groupe API peut évoluer indépendamment avec des versions comme v1, v1beta1, etc.

Accès au cluster kubernetes : APIs

L'API de Kubernetes supporte les opérations CRUD (Create, Read, Update, Delete) sur les ressources.

- POST : Créer une nouvelle ressource.
- GET : Lire les détails d'une ressource ou lister des ressources.
- PUT/PATCH : Mettre à jour une ressource existante.
- DELETE : Supprimer une ressource.

Ces opérations sont exécutées via des appels HTTP RESTful.

Accès au cluster kubernetes : APIs

Les endpoints de l'API Kubernetes sont des URL spécifiques utilisées pour interagir avec les ressources.

- /api/v1/namespaces/{namespace}/pods : Lister ou créer des pods dans un namespace donné.
- /apis/apps/v1/deployments : Gérer les déploiements.
- /api/v1/nodes : Obtenir des informations sur les nœuds du cluster.

Les requêtes peuvent être effectuées en utilisant des outils comme curl, ou à travers des SDKs dans différents langages de programmation.

Accès au cluster kubernetes : APIs

L'accès à l'API Kubernetes est protégé par des mécanismes d'authentification et d'autorisation.

- Authentification : Via des tokens, certificats client ou OAuth.
- Autorisation : Basée sur des rôles (RBAC - Role-Based Access Control) qui définissent quelles actions un utilisateur ou un service peut effectuer sur quelles ressources.

Accès au cluster kubernetes : APIs

- Les labels sont des paires clé-valeur attachées aux ressources Kubernetes.
- Ils permettent de filtrer et de sélectionner des ensembles de ressources via des sélecteurs (label selectors).
- Exemple : Sélectionner tous les pods avec le label app=frontend.
  - Endpoint : /api/v1/namespaces/{namespace}/pods?labelSelector=app=frontend.
- Les labels sont essentiels pour la gestion des déploiements, la mise en place de services, et l'organisation des ressources

Accès au cluster kubernetes : APIs

- Les annotations sont similaires aux labels, mais servent à attacher des métadonnées non modifiables par les systèmes internes de Kubernetes.
- Elles sont utilisées pour stocker des informations supplémentaires sur les ressources, comme des traces, des configurations ou des informations liées à des outils tiers.

Accès au cluster kubernetes : APIs

- Kubernetes utilise des contrôleurs qui implémentent des boucles de réconciliation pour garantir que l'état actuel des ressources correspond à l'état désiré spécifié par les utilisateurs.
  - Exemple : Le Deployment Controller s'assure que le nombre souhaité de réplicas d'un pod est toujours en cours d'exécution.
- Les contrôleurs sont des consommateurs de l'API Kubernetes qui surveillent les ressources, prennent des décisions, et effectuent les actions nécessaires.

Accès au cluster kubernetes : APIs

- Les Admission Controllers sont des composants qui interceptent les requêtes à l'API avant que les objets ne soient stockés. Ils peuvent modifier, rejeter ou valider des requêtes selon des règles définies.
  - Exemple : L'Admission Controller PodSecurityPolicy peut bloquer la création de pods non conformes aux politiques de sécurité.
- Les Admission Controllers jouent un rôle crucial dans la sécurisation et la gouvernance du cluster.

Accès au cluster kubernetes : APIs

- Kubernetes permet d'étendre l'API avec des Custom Resource Definitions (CRDs) et API Aggregation.
  - CRDs : Permettent aux utilisateurs de définir leurs propres types de ressources.
  - API Aggregation : Permet l'ajout de nouveaux groupes d'API, servant des ressources custom via des serveurs d'API supplémentaires.
- Ces mécanismes permettent aux développeurs d'étendre Kubernetes pour répondre à des besoins spécifiques ou créer des opérateurs.

Accès au cluster kubernetes : APIs

L'API de Kubernetes expose plusieurs endpoints pour le monitoring et le debugging du cluster.

- /metrics : Expose les métriques Prometheus pour les composants du cluster.
- /logs : Permet de récupérer les logs des nœuds et composants.
- /healthz : Points de vérification de la santé des composants.

Ces outils sont essentiels pour assurer la stabilité et la performance du cluster.

Déploiement et publication manuelle

Création du déploiement

1. Définir un fichier YAML pour le déploiement (deployment.yaml).
2. Spécifier l'image du conteneur, le nombre de réplicas, et les labels.
3. Appliquer le fichier avec :
    a. kubectl apply -f deployment.yaml

Exposition du déploiment

1. Définir un fichier YAML pour le service (service.yaml).
2. Associer le service au déploiement via les labels.
3. Appliquer le fichier avec :
    a. kubectl apply -f service.yaml

Détail et introspection du déploiement

1. kubectl get deployment <nom_du_deployment>
2. kubectl describe deployment <nom_du_deployment>
3. kubectl get service <nom_du_service>
4. Accéder à l'application via l'adresse IP et le port du service.

TP : Déploiement, publication et analyse d'un déploiement

paul.millet@lameduse.fr

LaMeDuSe

UBE : Kubernetes, mise en œuvre

Les fichiers descriptifs

1. Syntaxe YAML
2. Scalabilité d'un déploiement.
3. Stratégie de mise à jour sans interruption (update/rollback).
4. Suppression d'un déploiement.
5. TP : Déploiement, publication et analyse d'un déploiement.

Les fichiers descriptifs - Déploiement

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: my-app-image:v1
```

Les fichiers descriptifs - Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: LoadBalancer
```

Scalabilité d'un déploiement. (yaml)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 5 #modifier le nombre de réplicas
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: my-app-image:v1
```

Scalabilité d'un déploiement. (commande)

Sinon par commande :

- “kubectl scale deployment my-app --replicas=5"

Stratégie de mise à jour sans interruption (update/rollback).

- Mise à jour sans interruption :
  - Kubernetes permet de déployer des mises à jour d'applications sans interrompre les services.
  - Cela est réalisé à l'aide de déploiements gérés par des stratégies de mise à jour, garantissant que les utilisateurs continuent d'accéder aux applications pendant les modifications.

Stratégie de mise à jour sans interruption (update/rollback).

- Rolling Update : Cette stratégie met à jour progressivement les pods avec une nouvelle version de l'application, en remplaçant les anciens pods par de nouveaux, un ou plusieurs à la fois.
  - Avantages : Pas de temps d'arrêt, possibilité de surveiller chaque étape.
  - Fonctionnement :
    - Un nouveau pod est créé avec la mise à jour.
    - Un ancien pod est supprimé.
    - Le processus continue jusqu'à ce que tous les pods soient remplacés.

Stratégie de mise à jour sans interruption (update/rollback).

- Rollback : Si une mise à jour cause des problèmes, Kubernetes permet de revenir rapidement à une version antérieure du déploiement.
- Pourquoi faire un rollback ? : Corriger une erreur, éviter un downtime prolongé.
- Fonctionnement :
  - Kubernetes garde un historique des déploiements. Un rollback remet la configuration à une version antérieure.

Stratégie de mise à jour sans interruption (update/rollback).

- Déclencher un rollback :
  - Revenir à la version précédente :
    - kubectl rollout undo deployment my-app
  - Rollout vers une version spécifique :
    - Voir les révisions disponibles :
      - kubectl rollout history deployment my-app
    - Rollback vers une révision spécifique :
      - kubectl rollout undo deployment my-app --to-revision=<numéro_de_revision>
    - Vérifier l'état après rollback :
      - kubectl get deployment my-app
  -

Stratégie de mise à jour sans interruption (update/rollback).

Bonnes pratiques :

1. Tester les mises à jour sur un environnement de staging avant de les déployer en production.
2. Utiliser des sondes de santé (liveness et readiness probes) pour s'assurer que les pods sont prêts avant de passer à l'étape suivante du déploiement.
3. Surveiller en continu les performances et l'état des pods pendant la mise à jour.
4. Garder un historique des révisions pour faciliter les rollbacks rapides en cas de problème.

Suppression d'un déploiement.

- Etape 1 : Execution de la commande
  - kubectl delete deployment <nom_du_deployment>
- Étape 2 : Vérifier la Suppression
  - kubectl get deployments
  - kubectl get pods -l app=<label_du_deployment>
- Étape 3 : Nettoyer les Ressources Associées (optionnel)
  - kubectl delete service <nom_du_service>
- Supprimer les volumes persistants, configurations, et autres objets liés si vous ne prévoyez pas de les réutiliser :
  - kubectl delete pvc <nom_du_pvc>
  - kubectl delete configmap <nom_du_configmap>
  - kubectl delete secret <nom_du_secret>

TP : Déploiement, publication et analyse d'un déploiement.

paul.millet@lameduse.fr

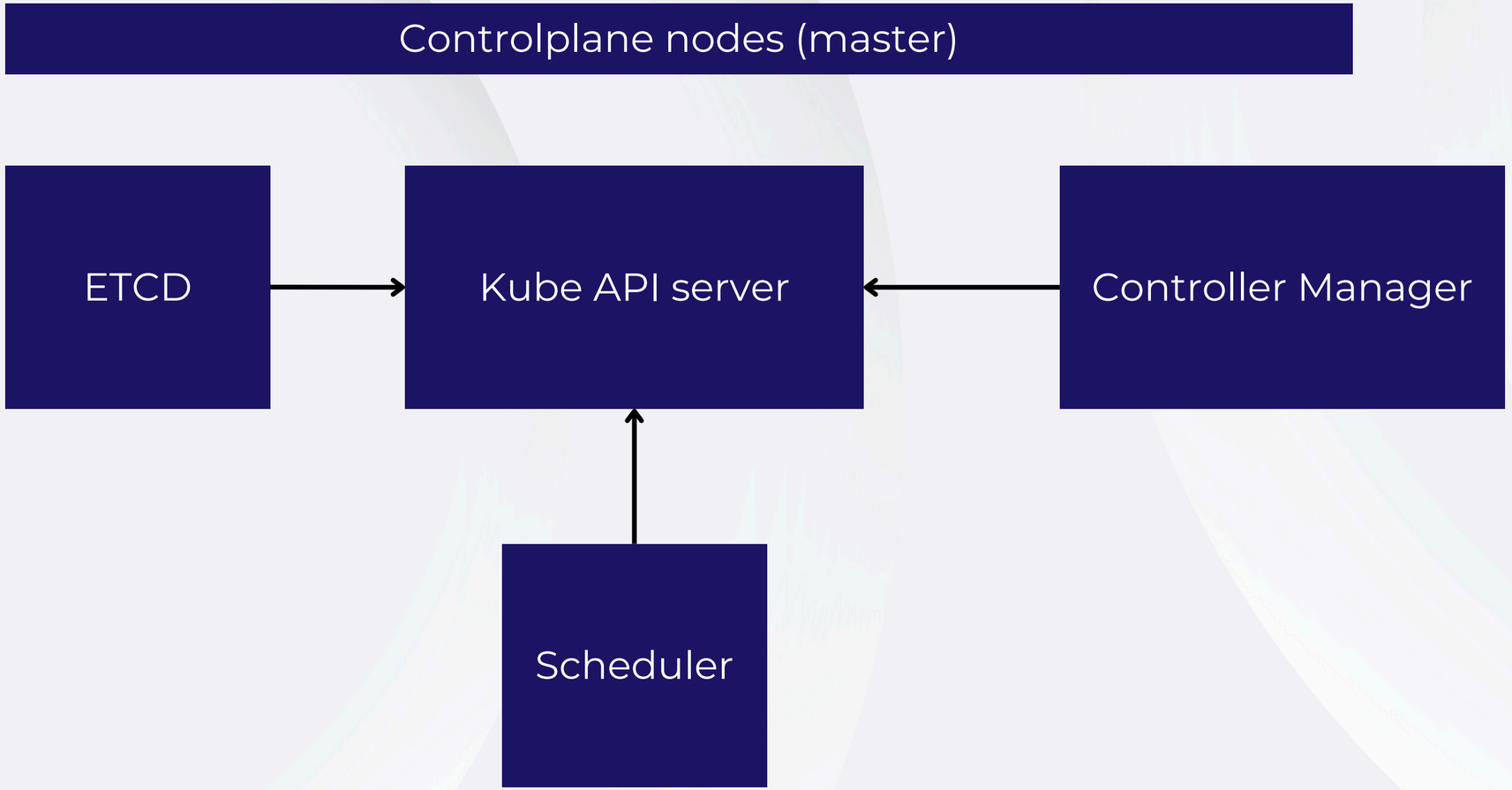
LaMeDuSe

UBE : Kubernetes, mise en œuvre

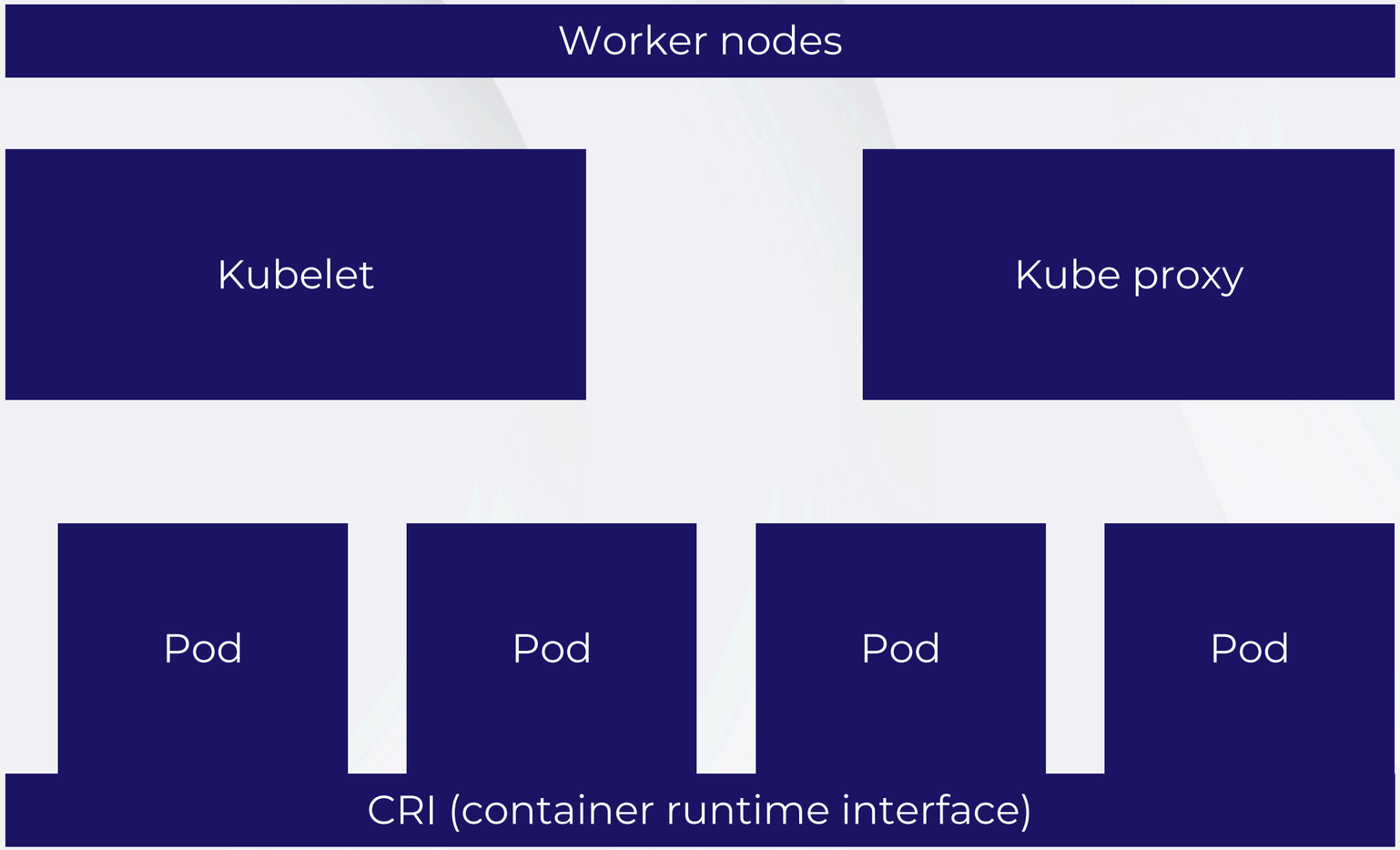
Architecture Kubernetes

1. Composants du master node : API server, scheduler, controller manager, etc.
2. Architecture d'un noeud : Kubelet, le moteur de conteneur (docker), Kube-proxy.
3. Objets Kubernetes : volume, service, pod, etc.
4. Objet statefull, objet stateless.
5. Solution du deployment.
6. TP : Utilisation de deployment.

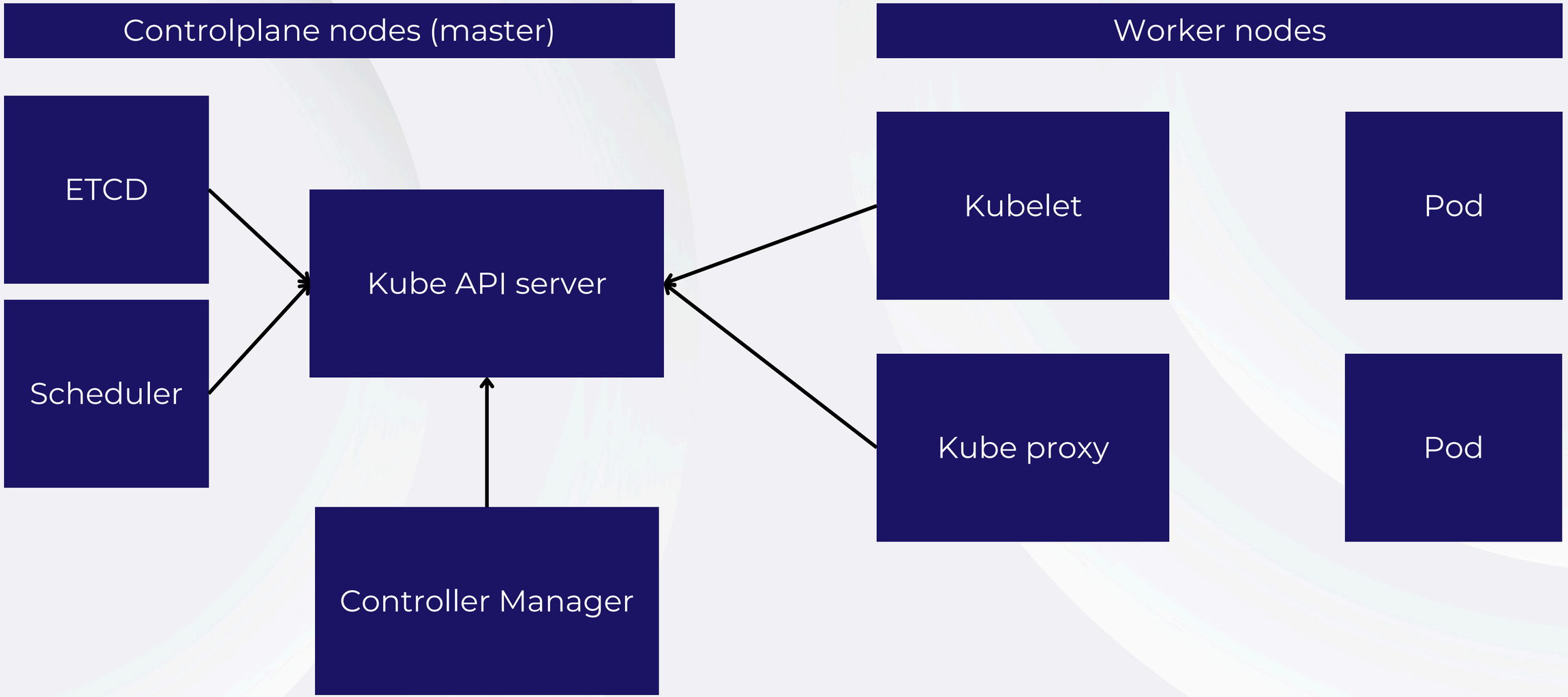
Composants du master node



Architecture d'un noeud



Architecture Kubernetes



Objets Kubernetes : volume, service, pod, etc.

- Pod
  - Unité de base dans Kubernetes, représentant un ou plusieurs conteneurs qui partagent des ressources (stockage, réseau).
  - Utilisé pour exécuter des applications conteneurisées.
- Deployment
  - Gère des ensembles de pods réplicas, permettant des mises à jour sans interruption et le scaling.
  - Fournit des fonctionnalités comme les rolling updates et les rollbacks.

Objets Kubernetes : volume, service, pod, etc.

- Service
  - Expose les pods pour permettre la communication réseau, interne ou externe.
  - Types principaux : ClusterIP, NodePort, LoadBalancer.
- Namespace
  - Permet de diviser les ressources d'un cluster en environnements logiques isolés.
  - Utilisé pour organiser et gérer des applications distinctes ou des équipes.

Objets Kubernetes : volume, service, pod, etc.

- ConfigMap
  - Stocke des configurations sous forme de paires clé-valeur, indépendamment du code d'application.
  - Injecte des configurations dans les pods via des variables d'environnement ou des volumes.
- Secret
  - Stocke des informations sensibles comme des mots de passe, des clés SSH, des certificats.
  - Similaire à ConfigMap mais sécurisé (base64 encodé).

Objets Kubernetes : volume, service, pod, etc.

- PersistentVolume (PV)
  - Représente un stockage physique dans le cluster, utilisé pour persister les données des pods.
  - Exemples : stockage local, disques réseau, NFS, etc.
- PersistentVolumeClaim (PVC)
  - Demande de stockage faite par les utilisateurs pour consommer des PersistentVolumes.
  - Les pods utilisent les PVC pour accéder au stockage.

Objets Kubernetes : volume, service, pod, etc.

- ServiceAccount
  - Fournit une identité pour les pods afin d'interagir avec l'API Kubernetes.
  - Utilisé pour limiter les permissions d'accès aux ressources du cluster.
- Role et RoleBinding
  - Role : Définit des permissions spécifiques (lecture, écriture) sur des ressources dans un namespace.
  - RoleBinding : Associe un rôle à un utilisateur ou un groupe.

Objets Kubernetes : volume, service, pod, etc.

- ClusterRole et ClusterRoleBinding
  - ClusterRole : Similaire à Role mais s'applique à l'ensemble du cluster.
  - ClusterRoleBinding : Associe un ClusterRole à des utilisateurs ou groupes à l'échelle du cluster.

Objets Kubernetes : volume, service, pod, etc.

- ReplicaSet
  - Assure un nombre spécifié de réplicas de pods en cours d'exécution à tout moment.
  - Géré principalement par des déploiements.
- DaemonSet
  - Exécute un pod sur chaque nœud du cluster (ou sous-ensemble de nœuds).
  - Utilisé pour des services de cluster comme les agents de surveillance ou de logging.

Objets Kubernetes : volume, service, pod, etc.

- StatefulSet
  - Gère des applications nécessitant un état stable ou un identifiant réseau unique.
  - Utilisé pour des bases de données, des caches, ou des applications à état.
- Job et CronJob
  - Job : Exécute une tâche jusqu'à sa complétion (pods temporaires).
  - CronJob : Planifie des Jobs à exécuter à des intervalles réguliers (similaire aux cron jobs Linux).

Objet statefull, objet stateless

- Stateful : Applications qui maintiennent un état ou des données persistantes entre les sessions ou les redémarrages.
  - Base de donnée, stockage de donnée
- Stateless : Applications qui ne conservent pas d'état entre les requêtes et peuvent être recréées ou déplacées sans impact.
  - Serveur web, serveur applicatif

Solution du deployment.

- Deployment : gère la création et la mise à jour de groupes de pods sans état (stateless).
- StatefulSet : utilisé pour les applications stateful qui nécessitent des identités stables et un stockage persistant.
- DaemonSet : garantit que tous (ou certains) nœuds exécutent une copie d’un pod donné.
- ReplicaSet : assure un nombre spécifique de pods en cours d'exécution à tout moment.
- Job : utilisé pour les tâches courtes et exécutées une seule fois (one-off).
- CronJob : planifie des tâches périodiques ou à exécution unique basées sur des expressions cron.

TP: Utilisation de deployment.

paul.millet@lameduse.fr

LaMeDuSe

UBE : Kubernetes, mise en œuvre

Exploiter Kubernetes

- Clusterisation avec replicas et deployment.
- Types de services.
- Labels et choix d'un nœud pour le déploiement.
- Affinité et anti-affinité.
- Daemons set, health check, config map et secrets.
- Persistent Volumes et Persistent Volumes Claim.
- TP: Déploiement d'une base de données et d'une application.

Clusterisation avec replicas et deployment.

- Objectif de la Clusterisation :
  - Assurer la disponibilité, la tolérance aux pannes et la scalabilité des applications en répartissant les charges sur plusieurs instances (pods) au sein du cluster.
- Rôles des Replicas et Déploiements :
  - Replicas : Instances identiques d'un pod pour équilibrer la charge et assurer la disponibilité.
  - Déploiements : Gèrent les replicas, permettent les mises à jour continues, le scaling et les rollbacks.

Clusterisation avec replicas et deployment.

- Pourquoi utiliser des Replicas ?
  - Haute Disponibilité : Répartir les pods sur plusieurs nœuds pour éviter les points de défaillance uniques.
  - Scalabilité : Augmenter ou diminuer le nombre de replicas pour s’adapter à la charge de travail.
  - Tolérance aux Pannes : Remplacer automatiquement les pods défaillants.

Clusterisation avec replicas et deployment.

- Rôle des Déploiements :
  - Gestion des Réplicas : Maintenir le nombre de pods spécifié et les redémarrer en cas de défaillance.
  - Mises à Jour Continues : Effectuer des mises à jour progressives (Rolling Updates) sans interruption de service.
  - Rollbacks : Revenir à une version précédente si une mise à jour échoue.

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1       # Pods supplémentaires pendant la mise à jour
    maxUnavailable: 1  # Pods indisponibles pendant la mise à jour
```

Types de services.

- ClusterIP (par défaut)
  - Usage : Expose le service à l'intérieur du cluster, accessible uniquement par d'autres services ou pods.
  - Utilisation courante : Communication interne entre les microservices.

```
kind: Service
metadata:
  name: my-app-clusterip
spec:
  type: ClusterIP
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
```

Types de services.

- NodePort
  - Expose le service sur une IP de chaque nœud du cluster, accessible via un port statique (port du nœud).
  - Exposer des services à l'extérieur du cluster pour le développement ou des tests.

```
kind: Service
metadata:
  name: my-app-nodeport
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30007  # Port entre 30000 et 32767
```

Types de services.

- ExternalName
  - Mappe un service à un nom DNS externe, redirige les requêtes vers des services en dehors du cluster.
  - Accéder à des services externes (bases de données, API externes).
  -

```
kind: Service
metadata:
  name: my-app-externalname
spec:
  type: ExternalName
  externalName: my.external.service.com
```

Types de services.

- Quand utiliser ClusterIP ?
  - Pour les communications internes entre microservices au sein du cluster.
- Quand utiliser NodePort ?
  - Pour des accès externes simples ou temporaires, développement, et tests.
- Quand utiliser LoadBalancer ?
  - Applications en production nécessitant une haute disponibilité et un accès depuis l'extérieur du cluster via un IP public.
- Quand utiliser ExternalName ?
  - Pour rediriger les requêtes à des services externes qui ne sont pas gérés par Kubernetes.

Labels et choix d'un nœud pour le déploiement.

- Qu'est-ce qu'un Label ?
  - Un label est une paire clé-valeur attribuée à des objets Kubernetes (pods, nœuds, services, etc.).
  - Utilisé pour organiser, sélectionner et filtrer les objets.
- Pourquoi utiliser des Labels ?
  - Pour catégoriser et identifier les objets pour des requêtes spécifiques.
  - Pour la sélection des pods dans les services, les déploiements, et d'autres configurations.
- Exemples de Labels :
  - “env: production”
  - “app: frontend”

Labels

- Sélection de Pods avec Labels :
  - Les services, déploiements, et autres ressources utilisent les labels pour cibler les pods spécifiques.

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

Labels

- Sélection de Pods avec Labels :
  - Les services, déploiements, et autres ressources utilisent les labels pour cibler les pods spécifiques.

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
```

Choix d'un nœud pour le déploiement.

- Choix d'un Nœud : Pourquoi ?
  - Pour déployer des pods sur des nœuds spécifiques en fonction de leurs caractéristiques (CPU, mémoire, emplacement, etc.).
  - Pour respecter des contraintes de sécurité, de performances, ou d'autres exigences spécifiques.
- Affinité de Nœud (Node Affinity) :
  - Déploye les pods sur les nœuds qui correspondent à certains labels.
- Types d'affinité :
  - Required (obligatoire)
  - Preferred (préférée).

Choix d'un nœud pour le déploiement.

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: my-app-container
    image: my-app-image:v1
```

Affinité et anti-affinité.

- Objectif :
  - Contrôler le placement des pods dans le cluster Kubernetes en fonction de règles prédéfinies.
  - Utilisé pour améliorer la distribution des workloads, optimiser l'utilisation des ressources, et répondre à des exigences spécifiques de performance ou de sécurité.
- Types :
  - Affinité de Pod: Pour spécifier où les pods doivent (ou ne doivent pas) être planifiés.
  - Anti-Affinité de Pod : Pour spécifier quels pods ne doivent pas être placés ensemble.

Affinité et anti-affinité.

- Affinité de Pod :
  - Encourage les pods à être placés à proximité d'autres pods pour améliorer la performance et la communication intra-cluster.
  - Utilise des règles basées sur des labels de pods.

```
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels:
              app: frontend
          topologyKey: "kubernetes.io/hostname"
  containers:
  - name: my-app-container
    image: my-app-image:v1
```

Le pod sera programmé sur le même nœud que les pods avec le label app=frontend

Affinité et anti-affinité.

- Anti-Affinité de Pod :
  - Empêche les pods d’être planifiés ensemble sur les mêmes nœuds pour répartir la charge ou pour des raisons de tolérance aux pannes.
  - Utile pour éviter les « points chauds » ou les dépendances critiques sur un seul nœud.

```
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels:
              app: frontend
          topologyKey: "kubernetes.io/hostname"
  containers:
  - name: my-app-container
    image: my-app-image:v1
```

Le pod ne sera pas programmé sur un nœud qui exécute déjà des pods avec le label app=frontend.

Daemons set, health check, config map et secrets.

- Qu'est-ce qu'un DaemonSet ?
  - Un DaemonSet assure qu'une copie d'un pod particulier tourne sur chaque nœud (ou un sous-ensemble de nœuds) du cluster.
  - Utilisé pour déployer des tâches de maintenance ou des services critiques, comme des agents de monitoring ou des collecteurs de logs.
- Utilisations Courantes :
  - Monitoring (ex. : Prometheus Node Exporter).
  - Collecte de logs (ex. : Fluentd, Filebeat).
  - Gestion des ressources (ex. : Daemons de stockage).

Daemons set, health check, config map et secrets.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-exporter
spec:
  selector:
    matchLabels:
      name: node-exporter
  template:
    metadata:
      labels:
        name: node-exporter
    spec:
      containers:
      - name: node-exporter
        image: prom/node-exporter
```

Daemons set, health check, config map et secrets.

- Qu'est-ce qu'un Health Check ?
  - Les Health Checks permettent à Kubernetes de surveiller l'état des applications via des probes.
  - Liveness Probe : Vérifie si une application fonctionne encore (et redémarre le pod en cas d'échec).
  - Readiness Probe : Vérifie si l'application est prête à recevoir du trafic (évite d'envoyer du trafic à un pod non prêt).
- Types de Probes :
  - HTTP Probe : Fait une requête HTTP sur un chemin spécifique.
  - Command Probe : Exécute une commande dans le conteneur et vérifie le code de retour.
  - TCP Probe : Tente de se connecter à un port TCP.

Daemons set, health check, config map et secrets.

```
spec:
  containers:
  - name: my-app-container
    image: my-app-image:v1
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 10
      periodSeconds: 10
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

Daemons set, health check, config map et secrets.

- Qu'est-ce qu'un ConfigMap ?
  - Un ConfigMap permet de stocker des configurations non sensibles sous forme de paires clé-valeur.
  - Permet de décorréler les paramètres de configuration du code de l'application.
- Utilisations Courantes :
  - Fournir des fichiers de configuration, des variables d'environnement.
  - Injecter des paramètres de configuration dans les pods à l'exécution.

Daemons set, health check, config map et secrets.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  DATABASE_URL: "postgres://user:password@host:5432/dbname"
  config.json: |
    {
      "setting1": "value1",
      "setting2": "value2"
    }
```

```
spec:
  containers:
  - name: my-app-container
    image: my-app-image:v1
    env:
    - name: DATABASE_URL
      valueFrom:
        configMapKeyRef:
          name: my-config
          key: DATABASE_URL
```

Daemons set, health check, config map et secrets.

- Qu'est-ce qu'un Secret ?
  - Un Secret est similaire à un ConfigMap, mais il est conçu pour stocker des données sensibles, comme des mots de passe, des clés API, ou des certificats.
  - Les données sont encodées en base64 et sont mieux sécurisées en mémoire et en transit que les ConfigMaps.
- Utilisations Courantes :
  - Stocker des mots de passe de base de données, des clés SSH, des tokens d'API.
  - Fournir des informations sensibles à des applications sans les exposer dans le code ou les configurations.

Persistent Volumes et Persistent Volumes Claim.

- Qu'est-ce qu'un Persistent Volume (PV) ?
  - Un PV est une ressource de stockage dans Kubernetes qui abstrait les détails de l'infrastructure de stockage (NFS, iSCSI, cloud storage, etc.).
  - Il est provisionné par un administrateur ou dynamiquement par Kubernetes.
- Caractéristiques :
  - Indépendant du cycle de vie des pods : les données persistent même après la suppression des pods.
  - Défini par des capacités (taille, type d'accès).

Persistent Volumes et Persistent Volumes Claim.

- Types d'accès :
  - ReadWriteOnce (RWO) : Peut être monté en lecture/écriture par un seul nœud.
  - ReadOnlyMany (ROX) : Peut être monté en lecture seule par plusieurs nœuds.
  - ReadWriteMany (RWX) : Peut être monté en lecture/écriture par plusieurs nœuds.

Persistent Volumes et Persistent Volumes Claim.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-example
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: /mnt/data
```

Persistent Volumes et Persistent Volumes Claim.

- Qu'est-ce qu'un Persistent Volume Claim (PVC) ?
  - Un PVC est une requête d'un utilisateur pour un PV avec des spécifications spécifiques (taille, modes d'accès).
  - Kubernetes associe un PVC à un PV approprié qui répond aux exigences de la demande.
- Fonctionnement :
  - Les utilisateurs créent des PVCs, et Kubernetes les associe automatiquement à des PVs disponibles.
  - PVCs permettent aux utilisateurs de demander du stockage sans avoir à connaître les détails de l'infrastructure sous-jacente.

Persistent Volumes et Persistent Volumes Claim.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-example
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

TP :Déploiement d'une base de données et d'une application.

paul.millet@lameduse.fr

LaMeDuSe

UBE : Kubernetes, mise en œuvre

LaMeDuSe

Kubernetes en production

- Frontal administrable Ingress.
- Limitation de ressources.
- Gestion des ressources et autoscaling.
- Service Discovery (env, DNS).
- Les namespaces et les quotas.
- Gestion des accès.
- Haute disponibilité et mode maintenance.
- TP: Déploiement de conteneur et gestion de la montée en charge.

Frontal administrable Ingress.

- Qu'est-ce qu'un Ingress ?
  - Ingress est un objet Kubernetes qui gère l'accès externe aux services du cluster, généralement via HTTP ou HTTPS.
  - Il offre un moyen de configurer des règles de routage pour le trafic entrant vers les services internes du cluster.
- Pourquoi utiliser Ingress ?
  - Pour exposer plusieurs services sous un seul point d'entrée (ex. : une seule adresse IP ou un seul DNS).
  - Pour gérer des règles de routage complexes, des terminaux TLS, et des équilibrages de charge.
  - Fournit des fonctionnalités de gestion de trafic avancées, comme les redirections et les réécritures d'URL.

Frontal administrable Ingress.

- Ingress Controller :
  - Composant responsable de l'implémentation des règles Ingress.
  - Différents contrôleurs disponibles (NGINX, Traefik, HAProxy, etc.).
  - Nécessaire pour faire fonctionner l'objet Ingress ; sans contrôleur, l'Ingress ne fonctionnera pas.
- Ingress Resource :
  - Définit les règles de routage pour les requêtes HTTP/HTTPS vers les services Kubernetes.
  - Utilise des annotations pour configurer des options spécifiques (ex. : authentification, redirection HTTPS).
- TLS Support :
  - Ingress peut gérer les certificats TLS pour sécuriser le trafic entre les clients et les services du cluster.

Frontal administrable Ingress.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1-service
            port:
              number: 80
  tls:
  - hosts:
    - example.com
    secretName: tls-secret
```

Limitation de ressources

- Pourquoi Limiter les Ressources ?
  - Gestion Efficace des Ressources : Assurer que les applications ne consomment pas plus de CPU ou de mémoire que nécessaire.
  - Isolation des Applications : Prévenir qu’une application gourmande en ressources n'affecte les autres applications du cluster.
  - Stabilité et Performance : Améliorer la stabilité et les performances globales du cluster en allouant les ressources de manière appropriée.

Gestion des ressources et autoscaling.

- Qu'est-ce que la Gestion des Ressources ?
  - La gestion des ressources dans Kubernetes permet de définir et de contrôler la quantité de CPU et de mémoire qu'un conteneur peut utiliser.
  - Assure une utilisation efficace des ressources du cluster et évite que des pods consomment plus que ce qui est disponible.
- Concepts Clés :
  - Requêtes de Ressources : Quantité minimale de CPU et de mémoire garantie pour un conteneur.
  - Limites de Ressources : Quantité maximale de CPU et de mémoire qu'un conteneur peut utiliser.

Gestion des ressources et autoscaling.

- Planification des Pods :
  - Kubernetes utilise les requêtes de ressources pour décider sur quel nœud placer un pod.
  - Les nœuds doivent avoir assez de ressources disponibles pour satisfaire les requêtes des pods.
- Impact des Limites :
  - Les limites préviennent qu'un conteneur consomme trop de ressources et impacte les autres conteneurs sur le même nœud.
  - Les conteneurs qui dépassent leurs limites de CPU seront throttlés (limitation de vitesse), ce qui peut affecter leur performance.
  - Les conteneurs qui dépassent leurs limites de mémoire seront tués et redémarrés par Kubernetes.

Gestion des ressources et autoscaling.

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-limits-example
spec:
  containers:
  - name: my-container
    image: my-image:v1
    resources:
      requests:
        memory: "512Mi"
        cpu: "500m"
      limits:
        memory: "1Gi"
        cpu: "1000m"
```

Gestion des ressources et autoscaling.

- Qu'est-ce que l'Autoscaling ?
  - L'autoscaling ajuste automatiquement le nombre de pods ou la taille du cluster en fonction de la demande de ressources.
  - Permet d'adapter l'infrastructure en réponse aux variations de la charge de travail.
- Types d'Autoscaling :
  - Horizontal Pod Autoscaler (HPA) : Ajuste le nombre de réplicas d'un pod basé sur l'utilisation des ressources ou des métriques personnalisées.
  - Vertical Pod Autoscaler (VPA) : Ajuste les requêtes et les limites de ressources des pods existants.
  - Cluster Autoscaler : Ajuste le nombre de nœuds dans un cluster pour répondre aux besoins en ressources des pods.

Gestion des ressources et autoscaling.

- Qu'est-ce que le HPA ?
  - Ajuste automatiquement le nombre de pods dans un déploiement ou un ensemble de réplicas en fonction de l'utilisation des ressources (CPU, mémoire) ou des métriques personnalisées.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
 name: hpa-example
spec:
 scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: my-deployment
 minReplicas: 1
 maxReplicas: 10
 metrics:
 - type: Resource
  resource:
   name: cpu
   target:
    type: Utilization
    averageUtilization: 50
```

Gestion des ressources et autoscaling.

- Qu'est-ce que le VPA ?
  - Ajuste automatiquement les requêtes et les limites de ressources des pods en fonction de leur utilisation réelle.
  - Utile pour les applications avec des besoins de ressources qui changent au fil du temps

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-example
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-deployment
  updatePolicy:
    updateMode: "Auto"
```

Service Discovery (env, DNS).

- Qu'est-ce que la Découverte de Services ?
  - La découverte de services permet aux applications de localiser et communiquer avec d'autres services dans un cluster Kubernetes sans avoir à connaître les adresses IP spécifiques.
- Mécanismes Principaux :
  - Variables d'Environnement : Kubernetes injecte des variables d'environnement dans les conteneurs pour la découverte des services.
  - DNS : Kubernetes fournit un service DNS interne pour résoudre les noms des services en adresses IP.

Service Discovery (env, DNS).

- Variables d'Environnement :
  - Kubernetes crée automatiquement des variables d'environnement pour chaque service, contenant des informations comme l'adresse IP et le port.
- Format des Variables d'Environnement :
  - Pour un service nommé my-service dans le namespace default, Kubernetes crée des variables comme :
    - MY_SERVICE_SERVICE_HOST : Adresse IP du service.
    - MY_SERVICE_SERVICE_PORT : Port du service.

Service Discovery (env, DNS).

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: example-image:v1
    env:
    - name: MY_SERVICE_SERVICE_HOST
      value: "10.0.0.1"
    - name: MY_SERVICE_SERVICE_PORT
      value: "80"
```

Service Discovery (env, DNS).

- Service DNS Kubernetes :
  - Kubernetes fournit un service DNS interne qui résout les noms de services en adresses IP.
  - Les noms de service sont résolus au format service-name.namespace.svc.cluster.local.
- Format de Résolution DNS :
  - Un service nommé my-service dans le namespace default peut être résolu en utilisant :
    - my-service.default.svc.cluster.local
    - Pour un service avec un port spécifique : my-service.default.svc.cluster.local:80

Service Discovery (env, DNS).

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: example-image:v1
    command:
    - curl
    - http://my-service.default.svc.cluster.local
```

Les namespaces et les quotas.

- Qu'est-ce qu'un Namespace ?
  - Un Namespace est une abstraction dans Kubernetes permettant de partitionner les ressources du cluster en plusieurs environnements isolés.
  - Utilisé pour organiser les ressources et les objets dans des environnements distincts, comme le développement, les tests, et la production.

Les namespaces et les quotas.

- Objectifs des Namespaces :
  - Isolation : Séparer les ressources pour différents projets ou équipes.
  - Gestion : Faciliter la gestion des ressources et des configurations spécifiques à chaque environnement.
  - Limitation : Appliquer des politiques de sécurité et des quotas spécifiques à chaque namespace.

Les namespaces et les quotas.

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

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

Les namespaces et les quotas.

- Qu'est-ce qu'un Quota de Ressources ?
  - Les quotas de ressources limitent la consommation de ressources (CPU, mémoire, etc.) dans un namespace pour éviter la surutilisation et assurer une gestion équitable des ressources.
- Objectifs des Quotas :
  - Prévention de la Surutilisation : Éviter qu'un namespace consomme toutes les ressources disponibles du cluster.
  - Gestion Équitable : Assurer que les ressources sont partagées équitablement entre différents namespaces.
  - Contrôle Budgétaire : Aider à respecter les contraintes budgétaires en contrôlant l'utilisation des ressources.

Les namespaces et les quotas.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resource-quota
  namespace: my-namespace
spec:
  hard:
    requests.cpu: "2"
    requests.memory: "4Gi"
    limits.cpu: "4"
    limits.memory: "8Gi"
    pods: "10"
    services: "5"
```

Gestion des accès.

- Qu'est-ce que la Gestion des Accès ?
  - La gestion des accès dans Kubernetes permet de contrôler qui peut interagir avec le cluster et quelles actions ils peuvent effectuer.
  - Assure la sécurité et la conformité en limitant l'accès aux ressources et aux opérations du cluster.
- Concepts Clés :
  - Authentification : Vérifie l'identité des utilisateurs ou des services.
  - Autorisation : Détermine ce qu'un utilisateur ou un service est autorisé à faire après l'authentification.
  - Contrôle d'Accès : Utilisation de rôles et de politiques pour gérer les permissions.

Gestion des accès.

- Types d'Authentification :
  - Certificats Client : Utilisation de certificats X.509 pour authentifier les utilisateurs et les services.
  - Jetons Bearer : Authentification via des jetons portés dans les requêtes HTTP.
  - API Keys : Clés API pour accéder aux services Kubernetes.
  - OAuth2/OpenID Connect : Intégration avec des fournisseurs d'identité externes pour une authentification simplifiée.

Gestion des accès.

- Contrôle d'Accès Basé sur les Rôles (RBAC) :
  - Permet de définir des rôles avec des permissions spécifiques et d'assigner ces rôles à des utilisateurs ou des groupes.
  - Les rôles définissent ce qu'un utilisateur ou un service peut faire avec les ressources Kubernetes.
- Types de Rôles :
  - ClusterRole : Définie des permissions à l'échelle du cluster.
  - Role : Définie des permissions dans un namespace spécifique.

Gestion des accès.

```
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: ClusterRole
metadata:
  name: cluster-admin
rules:
- apiGroups: [""]
  resources: ["*"]
  verbs: ["*"]
```

Gestion des accès.

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

```
apiVersion:
rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-binding
subjects:
- kind: User
  name: bob
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Gestion des accès.

- Pod Security Policies (PSP) :
  - Définissent des politiques de sécurité pour les pods, comme les capacités, les utilisateurs, et les volumes.
  - PSP est remplacé par les nouvelles fonctionnalités telles que PodSecurity Admission (PSA).
- PodSecurityAdmission (PSA) :
  - Remplace PSP et fournit une manière plus flexible et simplifiée de gérer les politiques de sécurité des pods.

Gestion des accès.

```
apiVersion: policy/v1
kind: PodSecurityPolicy
metadata:
  name: example-psp
spec:
  privileged: false
  capabilities:
    add: ["NET_ADMIN"]
  volumes:
  - "configMap"
  - "secret"
```

Gestion des accès.

- Modes de Sécurité PSA :
  - Privileged : Permet des configurations de sécurité moins restrictives, mais ce mode est généralement déconseillé pour la production.
  - Baseline : Applique des règles de sécurité de base, incluant des configurations minimales recommandées pour une sécurité raisonnable.
  - Restricted : Applique des règles de sécurité plus strictes pour des environnements plus sécurisés.

Gestion des accès.

```
apiVersion: policy/v1beta1
kind: PodSecurity
metadata:
  name: example-policy
  namespace: default
spec:
  enforce: baseline
```

Haute disponibilité et mode maintenance.

- Qu'est-ce que la Haute Disponibilité des Nœuds ?
  - Assure que les nœuds du cluster Kubernetes restent opérationnels et accessibles même en cas de défaillance de certains nœuds.
  - Évite les interruptions des services en cas de panne de nœuds ou de problèmes d'infrastructure.

Haute disponibilité et mode maintenance.

- Haute Disponibilité du Control Plane (master) :
  - Déploiement le Control Plane sur plusieurs nœuds pour éviter la défaillance du Control Plane.
  - Utilisez des solutions de stockage partagé et de synchronisation pour maintenir la cohérence entre les instances du Control Plane.
- Haute Disponibilité des Nœuds de Travail :
  - Déployez plusieurs nœuds de travail pour garantir que les applications continuent de fonctionner même si certains nœuds échouent.

Haute disponibilité et mode maintenance.

- Étape 1 : Marquer le Nœud comme Non-Disponible
  - Marquez le nœud pour qu'il ne reçoive pas de nouveaux pods.
  - “kubectl cordon <noeud>”
- Étape 2 : Déplacer les Pods Actuels
  - Évitez que les pods actuels restent sur le nœud pendant la maintenance.
  - kubectl drain <noeud> --ignore-daemonsets --delete-local-data
- Étape 3 : Effectuer la Maintenance
  - Réalisez les opérations de maintenance.
- Étape 4 : Réintégrer le Nœud dans le Cluster
  - kubectl uncordon <node-name>

TP: Déploiement de conteneur et gestion de la montée en charge.

paul.millet@lameduse.fr

LaMeDuSe

UBE : Kubernetes, mise en œuvre

LaMeDuSe

Kubernetes en production

- Préparation des nœuds.
- Déploiement : d'un master-nodeadm, d'un master-node, d'un worker-node.
- Mise en place du Dashboard et du réseau.
- TP: Déploiement d'un cluster.

Préparation des noeuds : Création des machines

- admin node
  - 2 go de ram
  - 2 coeur
- master node
  - 4 go de ram
  - 4 coeur
- worker node
  - 4 go de ram
  - 4 coeur

Master & Worker : Installation de Docker et de RKE2

```
# Téléchargement du script d'installation de docker
$ curl -fsSL https://get.docker.com -o install-docker.sh
# Vérification du script
$ cat install-docker.sh
# Lancement du script
$ sudo sh install-docker.sh
# Vérification de l'installation
$ systemctl status docker
# Activation du service au démarage
$ systemctl enable docker
# Installation de RKE2
$ curl -sfL https://get.rke2.io | sh -
```

Déploiement : d'un master-nodeadm, d'un master-node, d'un worker-node. (master)

```
# Execution du service
sudo systemctl enable rke2-server
sudo systemctl start rke2-server
# Récupération du token
sudo cat /var/lib/rancher/rke2/server/node-token
# Récupération de la configuration KubeConfig
sudo cat /etc/rancher/rke2/rke2.yaml
```

Déploiement : d'un master-nodeadm, d'un master-node, d'un worker-node. (agent)

```
# Configuration du service
sudo mkdir -p /etc/rancher/rke2
echo "server: https://<MASTER_NODE_IP>:6443" | sudo tee /etc/rancher/rke2/config.yaml
echo "token: <YOUR_CLUSTER_TOKEN>" | sudo tee -a /etc/rancher/rke2/config.yaml
# Execution du service
sudo systemctl enable rke2-agent
sudo systemctl start rke2-agent
```

Déploiement : d'un master-nodeadm, d'un master-node, d'un worker-node. (node-adm)

```
# Installation de Kubectl
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
# Copie du fichier kubeconfig
sudo cp /etc/rancher/rke2/rke2.yaml ~/.kube/config
sudo chmod 644 ~/.kube/config
```

Déploiement : d'un master-nodeadm, d'un master-node, d'un worker-node. (node-adm)

```
# Déploiement de la dashboard
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
# Vérifier le déploiement
kubectl get pods -n kubernetes-dashboard
kubectl get svc -n kubernetes-dashboard
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kubernetes-dashboard-ingress
  namespace: kubernetes-dashboard
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
  - host: dashboard.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: kubernetes-dashboard
            port:
              number: 443
```

TP: Déploiement d'un cluster.

paul.millet@lameduse.fr

LaMeDuSe

DOK : Docker, créer et administrer ses conteneurs virtuels d'applications

Ressources des TP

1. TP : Déploiement d'une plateforme de test
2. TP : Déploiement, publication et analyse d'un déploiement
3. TP : Utilisation de deployment
4. TP : Déploiement d'une base de données et d'une application
5. TP : Déploiement de conteneur et gestion de la montée en charge
6. TP : Déploiement d'un cluster

1- TP: Déploiement d'une plateforme de test

Sujet 1 : Utilisation de Minikube

1. Installation de Docker
2. Installation de Minikube
3. Démarer un cluster de 2 noeuds
4. Utiliser Kubectl pour explorer les différentes ressources
5. Lancer la dashboard

2- TP: Déploiement, publication et analyse d'un déploiement.

Sujet 1 : Déploiement

1. Créer un fichier de configuration YAML pour un Deployment.
    a. image : nginxdemos/nginx-hello
2. Déployer l'application avec kubectl apply -f deployment.yaml
3. Exposer le déploiement
    a. kubectl expose deployment <deployment-name> --type=NodePort --port=80
    b. (optionnel) essayer de ne pas utiliser la commande et de réaliser le yaml soi-même

3- TP: Utilisation de deployment

Sujet 1 : Déploiement

1. Déployer une Application
    a. Écrire un fichier YAML pour le Deployment avec les spécifications de l'application (image : nginxdemos/nginx-hello)
2. Mise à Jour de l'Application : Modifier l'image de l'application
    a. image: nginx
3. Scalabilité et Gestion des Réplicas
    a. Utiliser la commande scale pour modifier le nombre de répliques.
    b. Observer la montée ou la descente en charge via kubectl get pods.
4. Rollback d'un Déploiement :
    a. utiliser rollout undo pour revenir à la version stable précédente.

4- TP: Déploiement d'une base de données et d'une application

Sujet 1 : Informations

- variable pour la base de donnée
  - MYSQL_ROOT_PASSWORD=somewordpress
  - MYSQL_DATABASE=wordpress
  - MYSQL_USER=wordpress
  - MYSQL_PASSWORD=wordpress
- variable pour le serveur wordpress
  - WORDPRESS_DB_HOST=db
  - WORDPRESS_DB_USER=wordpress
  - WORDPRESS_DB_PASSWORD=wordpress
  - WORDPRESS_DB_NAME=wordpress
- Images
  - mariadb:10.6.4-focal
  - wordpress:6.6.1-php8.2-apache

4- TP: Déploiement d'une base de données et d'une application

Sujet 1 : Partie 1 : Déploiement de la base de donnée

- Création du fichier de déploiement
  - Utiliser un statefulset
  - Créer un service (cluster IP)
- Application du fichier de déploiement
- Vérification du déploiement

4- TP: Déploiement d'une base de données et d'une application

Sujet 1 : Partie 2 : Déploiement de l’application

- Création du fichier de déploiement
  - Utiliser un deployment
  - Créer un service (cluster IP)
- Application du fichier de déploiement
- Vérification du déploiement

5- TP: Déploiement de conteneur et gestion de la montée en charge

Sujet 1 : Partie 2 : Déploiement de l’application

- Création du fichier de déploiement
  - Utiliser un deployment
  - Créer un service (cluster IP)
- Création d’une politique d’autoscaling
- Test avec un outil comme “hey” pour faire monter la charge
  - hey -z 5m -c 1000 http://<node-ip>:<node-port>

6- TP: Déploiement d'un cluster

Sujet 1 :

- Création des machines virtuelles
- Installation des prérequis
- Mise en place du noeud maitre
  - Démarrage du service
  - Récupérer le token
- Mise en place du noeud worker
  - Configurer le noeud avec le token récupéré
  - Démarage du service
- Mise en place du noeud d’aministration
  - Installer kubectl
  - Mettre le fichier kubeconfig

6- TP: Déploiement d'un cluster

Sujet 2 : Utiliser le cluster

- déployer wordpress et la base de donnée sur le cluster

paul.millet@lameduse.fr

LaMeDuSe

UBE : Kubernetes, mise en œuvre

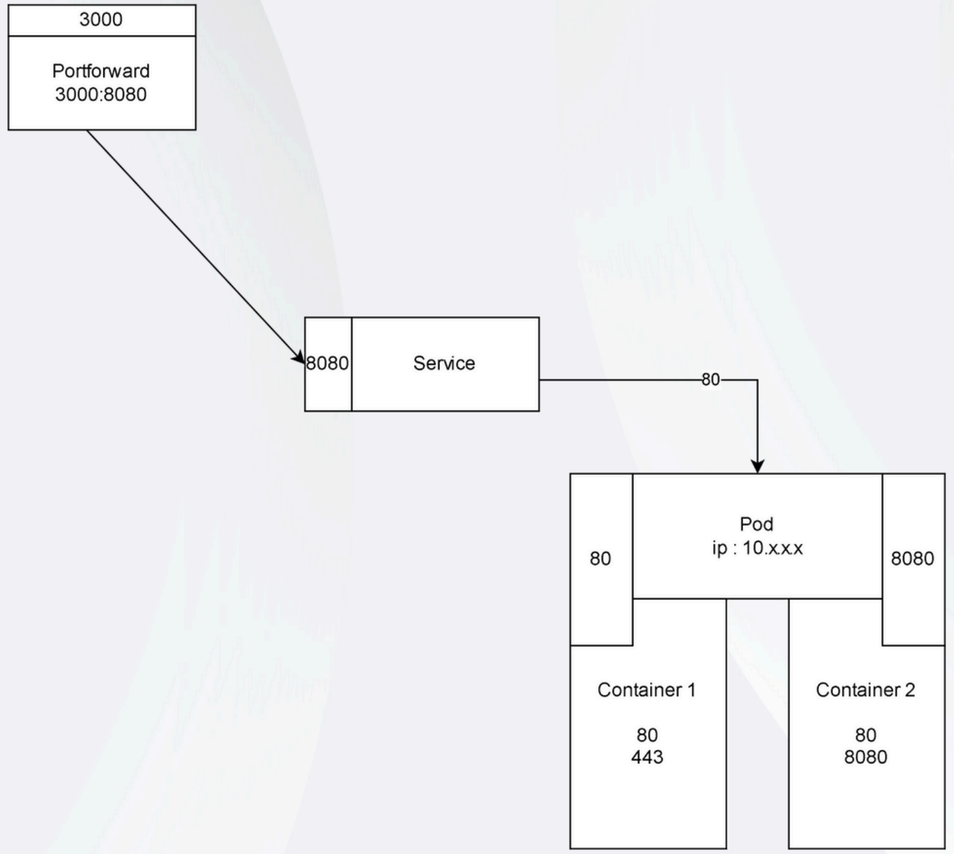
LaMeDuSe

Introduction à Kubernetes

Annexes :

- A. Schemas diverses

Introduction à Kubernetes



Introduction à Kubernetes