Prérequis d'installation
DOCKER & MINIKUBE

Minimum Hardware & Os
==========

- Un processeur compatible avec la virtualisation (VT-x ou AMD-V)
- Virtualisation activée sur le processeur (voir bios) (`grep -E --color 'vmx|svm' /proc/cpuinfo`)
- 2go de mémoire (4 ou + de préférence)
- 2 coeurs (vcpu) ou plus
- 35 Go espace disque ou +
- Linux noyau supérieur ou égal à 6.8
- Distribution linux de type desktop, un bon candidat est Ubuntu 24.04.2, cette installation se déroule sur ce type de distribution standard, il faudra adapter les commandes selon votre distribution.
- Accès internet.
- Gestionnaire de conteneur ou machine virtuelle : Docker version engine 27 ou plus installé
- Sinon l'un de ces systèmes de virtualisation : QEMU, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox, ou VMware Fusion/Workstation.
- OpenJDK 21 ou 24 installé et accessible via le « $PATH » de votre environnement
- Maven 3.9.11 ou plus installé et accessible via le « $PATH » de votre environnement

L'ensemble peut être une machine physique complète ou une machine virtuelle qui est capable de respecter ces prérequis.

Compilation avec springboot
==========

- Installation d'un environnement Maven nécessaire
- Idéalement l'environnement de dernière version est idéal,
- Actuellement : la version 3.9.11
- Accessible via le prompt et le $PATH

Documentations
==========
https://knative.dev/docs/install/quickstart-install/
https://minikube.sigs.k8s.io/docs/
https://kubernetes.io/fr/

Déroulement de l'installation pour minikube avec téléchargement des binaires

Notes : installation possible sur mac Os, fortement déconseillé sur Windows.

A partir d'une installation Ubuntu version 24.04.1 avec docker installé
==========
Les étapes ci-dessous doivent être scrupuleusement respectées dans l'ordre présenté.
Les commandes sont à exécuter ligne par ligne pour plus de contrôle.

Installer docker & curl (si non installé : veuillez vérifier avant !)
==========

```
$> sudo snap install curl  # ou
$> sudo apt  install curl
```

```
$> sudo snap install docker # ou
$> sudo apt  install docker  # voir le td sur Docker.
```

Autoriser l'utilisateur par défaut à accéder à docker (impératif sinon minikube & knative ne trouvent pas docker)

```
----------
$> sudo groupadd docker
$> sudo usermod -aG docker $USER
$>  newgrp docker
----------
```

Un reboot du système peut être nécessaire pour prise en compte. (Non systématique).

Installation de openjdk 21
==========
```
$> sudo apt install openjdk-21-jdk
ou
$> sudo apt install openjdk-24-jdk
```

Tester :
```
$> java --version
$> javac --version
```

Installation de maven
==========
$> wget https://dlcdn.apache.org/maven/maven-3/3.9.11/binaries/apache-maven-3.9.9-bin.tar.gz
```
$> tar xzvf apache-maven-3.9.11-bin.tar.gz
$> sudo mv apache-maven-3.9.11 $HOME/maven
$> vim ~/.bashrc (modifier le $PATH pour ajouter le dossier bin de maven
$> source .bashrc
```

Installer kubectl (juste pour l'installation, on utilisera plutôt minikube kubectl)
==========
```
$> export os="linux/amd64"
$> curl -LO "https://dl.k8s.io/release/$(curl -sL
https://dl.k8s.io/release/stable.txt)/bin/$os/kubectl"
$> chmod a+x ./kubectl
$> sudo mv ./kubectl /usr/local/bin/kubectl
```

Instaler minikube
==========
```
$> curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
$> chmod a+x minikube
$> sudo mv ./minikube /usr/local/bin/minikube
```

Tester !
```
$> minikube version
```

Exécuter minikube
==========
```
$> minikube start
```

Installation de Lens-Desktop
==========
note : Lens est une plateforme Kubernetes graphique qui fournit des outils pour une interaction transparente avec les clusters Kubernetes et un environnement pour un travail sécurisé et efficace au sein des équipes et des organisations.

```
$> curl -fsSL https://downloads.k8slens.dev/keys/gpg | gpg --dearmor | sudo
tee /usr/share/keyrings/lens-archive-keyring.gpg > /dev/null
```

```
$> echo "deb [arch=amd64 signed-by=/usr/share/keyrings/lens-archive-keyring.gpg]
https://downloads.k8slens.dev/apt/debian stable main" | sudo tee
/etc/apt/sources.list.d/lens.list > /dev/null
```

```
$> sudo apt update && sudo apt install lens
```

Tester

$> lens-desktop => au premier démarrage vous devrez créer un compte sur le site https://k8slens.dev/

Le premier démarrage peut prendre plusieurs minutes !

Approche Kubernetes

Approches de Gestion : Impérative vs Déclarative

- **Approche impérative** : gestion des ressources Kubernetes en exécutant des commandes directes.
- **Approche déclarative** : gestion des ressources via des fichiers de configuration YAML, avec une vision "état désiré".

Lorsqu'il s'agit de gérer les ressources dans Kubernetes, deux approches principales sont utilisées : l'approche impérative et l'approche déclarative. Chaque méthode présente ses avantages, en fonction des besoins spécifiques en matière de gestion, de contrôle et d'automatisation des ressources.

- **Approche impérative**

L'approche impérative consiste à gérer les ressources Kubernetes en exécutant des commandes directes. Elle est utile pour des tâches ponctuelles ou pour effectuer rapidement des modifications spécifiques dans le cluster. En utilisant des commandes kubectl en ligne de commande, cette méthode permet un retour immédiat des actions entreprises. Cependant, elle peut être plus difficile à maintenir sur le long terme, car les modifications ne sont pas enregistrées sous forme de fichiers de configuration, rendant le suivi des changements plus complexe.

- Exemple d'utilisation impérative :

```
kubectl create deployment nginx --image=nginx
kubectl scale deployment nginx --replicas=3
```

- **Cas d'Usage:**
- **Tests et Développement Rapide**

- Pour les développeurs qui ont besoin de créer, tester et détruire des ressources rapidement dans un environnement de développement ou de test, l'approche impérative est efficace. Par exemple, un développeur peut créer un pod pour tester un service sans avoir à écrire de fichier YAML.

- **Tâches Ad Hoc et debug**

- Lorsqu'il est nécessaire de réaliser des tâches ponctuelles, comme la mise à l'échelle rapide d'un déploiement ou le dépannage d'une ressource. Les commandes directes permettent d'apporter des changements immédiats sans avoir à modifier des fichiers de configuration.

- **Interventions Manuelles**

- Pour les opérations qui nécessitent une intervention humaine, comme le déploiement rapide d'une application temporaire pour une démonstration ou un événement spécifique, l'approche impérative permet une flexibilité rapide.

- **Approche déclarative**

L'approche déclarative permet de gérer les ressources via des fichiers de configuration YAML, en définissant l'état désiré des objets Kubernetes. En appliquant ces fichiers avec des commandes comme kubectl apply, on laisse à Kubernetes la tâche de faire correspondre l'état actuel de chaque ressource à cet état désiré. Cette méthode facilite la gestion des ressources

sur le long terme, car les fichiers de configuration servent de source de vérité, simplifiant ainsi l'automatisation et le suivi des modifications.

- Exemple de fichier déclaratif YAML pour un déploiement :

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

- **Cas d'Usage:**
- **Gestion des Environnements de Production**

- Dans des environnements de production où la stabilité et la prévisibilité sont essentielles, l'approche déclarative permet de définir l'état souhaité des ressources et de s'assurer que Kubernetes maintient cet état. Cela facilite les mises à jour, les rollbacks et la gestion des configurations.

- **Automatisation et CI/CD**

- Dans les pipelines d'intégration et de déploiement continu (CI/CD), l'approche déclarative permet d'automatiser le déploiement d'applications. Les fichiers YAML peuvent être stockés dans un contrôle de version, garantissant que les modifications de configuration sont suivies et peuvent être répliquées facilement.

- **Infrastructure as Code (IaC)**

- Lors de la mise en œuvre de pratiques d'IaC, l'approche déclarative permet de gérer l'infrastructure de manière systématique et versionnée. Les fichiers de configuration peuvent être utilisés pour configurer non seulement les applications, mais aussi les ressources d'infrastructure nécessaires.

- **Gestion de Configuration**

- Pour les équipes qui doivent gérer des configurations complexes et des dépendances entre différentes ressources, l'approche déclarative permet de décrire les relations et les configurations dans des fichiers distincts, facilitant ainsi leur gestion.

Les approches impérative et déclarative peuvent être utilisées conjointement, mais pour les environnements de production, l’approche déclarative est généralement privilégiée pour sa fiabilité et sa facilité d'automatisation.

Ressources de Base: pod

- **Pod** : unité de base de déploiement, contenant un ou plusieurs conteneurs.

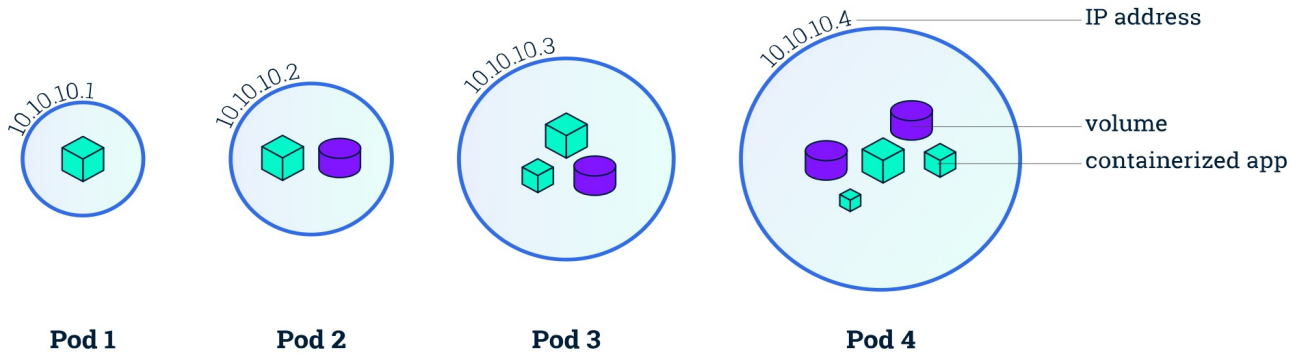
- **Labels et Annotations** : méthodes de marquage pour l’organisation, la sélection, et l’ajout de métadonnées aux objets.

Le Pod:

Un pod est l'unité de base dans Kubernetes. C'est la plus petite unité déployable qui peut contenir un ou plusieurs containers.

Certaines ressources partagées pour ces conteneurs.:

- Stockage partagé, en tant que Volumes
- Mise en réseau, en tant qu'adresse IP d'un unique cluster
- Informations sur l'exécution de chaque conteneur, telles que la version de l'image du conteneur ou les ports spécifiques à utiliser
-



- Les labels sont des paires clé-valeur qui sont attachées aux objets Kubernetes pour les catégoriser et les organiser.

- Les annotations sont des métadonnées supplémentaires attachées à un objet Kubernetes pour fournir des informations supplémentaires.

Exemple avec le service nginx:

- Exemples de pods Kubernetes avec des labels et des annotations en mode imperatif:

```
# créer un pod nginx en mode imperatif
kubectl run nginx-pod-imp --image=nginx:latest --labels=app=web,env=production

# ces deux commandes permettent de rediriger la sortie dans un fichier.
kubectl run redis-pod --image=redis:latest --labels=app=database,tier=backend --
dry-run=client -o yaml > redis-pod.yaml
kubectl run nginx-pod --image=nginx:latest --labels=app=web,env=production --
dry-run=client -o yaml > nginx-pod.yaml
```

- Le meme pode nginx en mode declaratif: Copier le continu dans le ficher nginx-pod-decla.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-decla
  labels:
    app: web
    env: production
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
```

```
# executer le fichier nginx-pod-decla.yaml

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

```
coul@home-lab ~/Documents/indus/formation-blog/kubernetes (k8s: kubernetes-admin@kubeadm-cluster-dev)
$ kubectl run nginx-pod-imp --image=nginx:latest --labels=app=web,env=production
pod/nginx-pod-imp created
coul@home-lab ~/Documents/indus/formation-blog/kubernetes (k8s: kubernetes-admin@kubeadm-cluster-dev)
$ kubectl apply -f pod/nginx-pod.yaml
pod/nginx-pod-decla created
coul@home-lab ~/Documents/indus/formation-blog/kubernetes (k8s: kubernetes-admin@kubeadm-cluster-dev)
$ kubectl get po -o wide
NAME             READY   STATUS    RESTARTS   AGE   IP             NODE                NOMINATED NODE   READINESS GATES
nginx-pod-decla  1/1     Running   0          6s    10.11.135.71   kubeadm-worker-02   <none>           <none>
nginx-pod-imp    1/1     Running   0          11s   10.11.135.70   kubeadm-worker-02   <none>           <none>
coul@home-lab ~/Documents/indus/formation-blog/kubernetes (k8s: kubernetes-admin@kubeadm-cluster-dev)
$
```

Pour acceder à notre nginx-pod-imp pour faire des debug, on peut faire un port forward:

```
kubectl port-forward nginx-pod-imp 8088:80
```

```
coul@home-lab ~/Documents/indus/formation-blog/kubernetes (k8s: kubernetes-admin@kubeadm-cluster-dev)
$ kubectl port-forward nginx-pod-imp 8088:80
Forwarding from 127.0.0.1:8088 -> 80
Handling connection for 8088
```

Accéder à : http://127.0.0.1:8088

Parcourt de la commande kubectl:

Quand on execute kubectl apply -f nginx-prod-decla.yaml, Kubernetes suit plusieurs étapes pour créer le pod ou la resoruce demandé. Voici le déroulement de la communication entre les composants Kubernetes, de l'application de la déclaration jusqu'à la création du pod :

1. **kubectl** :

- L'outil kubectl transmet la ressource définie dans le fichier YAML (nginx-prod-decla.yaml) à l'API Server.
- kubectl envoie une requête HTTP à l'API Server (via REST) pour créer ou mettre à jour la ressource dans etcd.

2. **API Server** :

- L'API Server reçoit la requête et vérifie que la ressource YAML est valide.
- Elle authentifie et autorise la requête (en fonction des contrôles RBAC configurés) avant de la traiter.
- Une fois validée, l'API Server enregistre la configuration de la ressource dans la base de données etcd.

3. **etcd** :

- etcd est la base de données distribuée dans Kubernetes où toutes les configurations et l'état des ressources sont stockés.
- Quand l'API Server enregistre la nouvelle ressource (ici le Pod), etcd conserve cette information. C'est ici que la "source de vérité" de l'état du cluster est maintenue.

4. **Scheduler** :

- Le Scheduler (planificateur) est continuellement à l'écoute des ressources non assignées dans etcd.
- Quand il détecte que le Pod doit être créé mais n'a pas encore été associé à un nœud, il analyse les critères de planification (ressources disponibles, affinités, tolérances, etc.).
- Le Scheduler choisit alors le nœud le plus approprié pour exécuter le Pod et met à jour etcd pour l'assigner au nœud sélectionné.

5. **Kubelet (sur le nœud sélectionné)** :

- Le kubelet du nœud choisi par le Scheduler interroge périodiquement l'API Server pour vérifier les nouvelles ressources qui lui sont assignées.
- Il détecte le nouveau Pod et se charge de le créer.
- Kubelet interagit avec le runtime de conteneur (Docker, containerd, etc.) pour télécharger les images nécessaires et lancer le conteneur du Pod.

6. P**od en cours d'exécution** :

- Une fois le Pod créé et le conteneur en cours d'exécution, Kubelet surveille son état et maintient la communication avec l'API Server pour informer de tout changement d'état.
- Si le Pod tombe en panne, Kubelet peut, selon la stratégie de redémarrage, essayer de redémarrer le conteneur pour respecter l'état désiré défini dans le manifeste YAML.

Ainsi, la communication entre les composants Kubernetes – du kubectl au kubelet – s'assure que la ressource demandée passe par plusieurs validations et processus avant que le Pod ne soit effectivement créé et rendu disponible dans le cluster.

Exemple avec le pod prometheus node-exporter:

- Pod simple avec des labels et des annotations mode imperative

```
kubectl run monitoring-pod --image=prometheus/node-exporter:latest --
labels=app=monitoring
--annotations=prometheus.io/scrape=true,prometheus.io/port=8080,team=ops
kubectl run monitoring-pod --image=prometheus/node-exporter:latest --
labels=app=monitoring
--annotations=prometheus.io/scrape=true,prometheus.io/port=8080,team=ops --dry-
run=client -o yaml > monitoring-pod.yaml
```

L'utilisation de fichiers de configuration YAML est une pratique courante pour décrire les ressources Kubernetes. Voici un exemple de fichier YAML pour déployer un pod simple :

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    prometheus.io/scrape: true
    prometheus.io/port: 8080
    team: ops
  labels:
    app: monitoring
  name: monitoring-pod
spec:
  containers:
  - image: prometheus/node-exporter:latest
    name: monitoring-pod
```

Dans cet exemple, le pod "monitoring-pod" est étiqueté avec "app: monitoring". Il comporte également deux annotations spécifiques à Prometheus ("prometheus.io/scrape" et "prometheus.io/port") qui indiquent que ce pod doit être scrappé par le serveur Prometheus pour collecter des métriques. L'annotation "team" fournit des informations supplémentaires sur l'équipe responsable du pod.

Exemple avec le Pod redis:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-pod
  labels:
    app: database
    tier: backend
```

```
  annotations:
    description: "This pod runs a Redis database."
spec:
  containers:
  - name: redis-container
    image: redis:latest
```

Dans cet exemple, le pod "redis-pod" a les labels "app: database" et "tier: backend". Ces labels peuvent être utilisés pour sélectionner des pods spécifiques à l'aide de sélecteurs lors de la création de services ou de déploiements.

Exemple de Pod avec 2 ou 3 containers:

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
  - name: busybox-container
    image: busybox:latest
    command: ['sh', '-c', 'while true; do echo "Hello from BusyBox"; sleep 3600;
done']
```

Dans cet exemple, nous avons un pod nommé "multi-container-pod" avec deux containers. Le premier container est basé sur l'image NGINX et expose le port 80. Le deuxième container est basé sur l'image BusyBox et exécute une commande qui imprime "Hello from BusyBox" toutes les heures.

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod-3
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
  - name: busybox-container
    image: busybox:latest
    command: ['sh', '-c', 'while true; do echo "Hello from BusyBox"; sleep 3600;
done']
  - name: alpine-container
    image: alpine:latest
    command: ['sh', '-c', 'while true; do echo "Hello from Alpine"; sleep 3600;
done']
```

Dans cet exemple, nous avons un pod nommé "multi-container-pod-3" avec trois containers. Le premier container est basé sur l'image NGINX, le deuxième sur l'image BusyBox, et le troisième sur l'image Alpine. Chacun des containers exécute une commande qui imprime un message toutes les heures.

Ces exemples illustrent comment vous pouvez définir plusieurs containers au sein d'un même pod. Chaque container dans le pod peut avoir des responsabilités différentes et partager des ressources et de l'espace de stockage, ce qui est utile pour divers scénarios d'application et de microservices.

Ces exemples illustrent comment vous pouvez utiliser des labels pour organiser et catégoriser vos pods, ainsi que comment utiliser des annotations pour fournir des informations supplémentaires sur vos pods, telles que des descriptions, des configurations spécifiques ou des métriques à collecter.

Commandes de debug:

```
kubectl get pod/pod_name
kubectl describe pod/pod_name
kubectl logs pod/pod_name

# si le pod contient plusieurs containers
kubectl exec -it pod_name -c my-container -- /bin/bash

# si le pod contient un seul container
kubectl exec -it pod_name -- /bin/bash
```

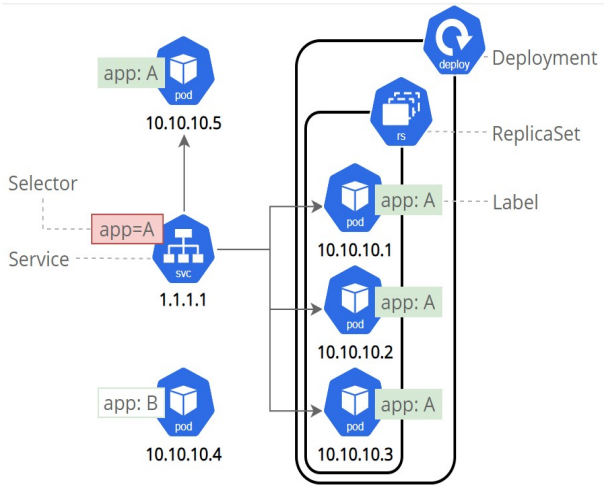
Pour supprimer le pod:

```
kubectl delete pod nginx-pod-imp
kubectl delete -f nginx-pod.yaml
```

Services et Types de Services

- Service : Abstraction réseau pour exposer les applications au sein ou à l'extérieur du cluster.
- Types de Services : ClusterIP, NodePort, LoadBalancer, et ExternalName / externalIP.
- DNS dans Kubernetes

nous allons plonger dans les concepts de réseau au sein de Kubernetes. Nous allons aborder les différentes notions telles que les services, le DNS et les politiques réseau. De plus, nous allons explorer comment configurer les services pour exposer des applications à l'intérieur et à l'extérieur du cluster.



Concepts de réseau dans Kubernetes

Avant de plonger dans les détails, il est important de comprendre les fondamentaux du réseau dans Kubernetes. Nous allons explorer les trois principaux concepts qui sont les piliers de la gestion du réseau :

- **Services Kubernetes** : Les services sont une abstraction qui permet d'exposer des applications aux utilisateurs ou à d'autres services au sein du cluster. Ils offrent un moyen de

communication stable et fiable avec les applications, indépendamment de l'endroit où elles sont déployées.

- **DNS dans Kubernetes** : Le service DNS de Kubernetes permet aux conteneurs de communiquer entre eux en utilisant des noms au lieu d'adresses IP. Cela facilite grandement la gestion des connexions entre les différents composants de l'application.

- **Politiques Réseau** : Les politiques réseau définissent les règles pour contrôler les communications entre les pods. Cela permet d'assurer un niveau de sécurité et d'isolation dans l'environnement.

Configuration de services pour exposer des applications

Les services Kubernetes jouent un rôle essentiel pour exposer vos applications aux utilisateurs et aux autres services. Voici comment configurer différents types de services pour répondre à vos besoins :

1. **ClusterIP** : Expose le service au sein du cluster en utilisant une adresse IP interne.

2. **NodePort** : Expose le service à l'extérieur du cluster en utilisant un port spécifique sur chaque nœud.

3. **LoadBalancer** : Expose le service à l'extérieur du cluster et provisionne un équilibreur de charge externe.

4. **ExternalName** : Renvoie des noms DNS à l'extérieur du cluster.

5. **Headless** : Renvoie directement aux IPs des pods sans kubeproxy

- **Le serice type ClusterIP**

Le service ClusterIP permet d'exposer une application à l'intérieur du cluster en utilisant une adresse IP interne. Voici un exemple de configuration YAML pour un service ClusterIP :

```
apiVersion: v1
kind: Service
metadata:
  name: caddy-service-clusterip
spec:
  selector:
    app: caddy-web-app  # Sélecteur pour cibler les pods avec ces labels
  ports:
    - protocol: TCP
      port: 8080  # Port du service caddy-web-app
      targetPort: 80 # Port sur lequel caddy écoute à l'intérieur du pod
---

apiVersion: v1
kind: Pod
metadata:
  name: caddy-pod
  labels:
    app: caddy-web-app
    tier: front-end
  annotations:
    description: "This pod runs a caddy web app"
spec:
  containers:
  - name: caddy-container
    image: caddy:latest
```

```
# creer le ficher clusterip.yaml et appliquer le
kubectl apply -f clusterip.yaml
kubectl get pod,svc
```

Dans cet exemple, le service caddy-service-clusterip expose l'application associée aux pods sur le port 8080 avec l'étiquette app: caddy-web-app en utilisant le port 80 à l'intérieur du cluster.

Depuis un autre pode, on peut faire un curl sur le service name: caddy-service-clusterip:8080

Test avec busybox

```
kubectl run ubuntu-debug --rm -it --image=ubuntu -- /bin/bash

# installer les tools de debug dans le le pod ubuntu-debug
apt update && apt install -y dnsutils curl net-tools inetutils-ping

# lancer le curl depuis le pod ubuntu-debug
curl http://caddy-service-clusterip.default.svc.cluster.local:8080
```

Pour supprimer:

```
kubectl delete -f clusterip.yaml
```

- **Le serice type NodePort**

Le service NodePort expose l'application à l'extérieur du cluster en utilisant un port spécifique entre 30000-32767 sur chaque nœud. Voici un exemple de configuration YAML pour un service NodePort :

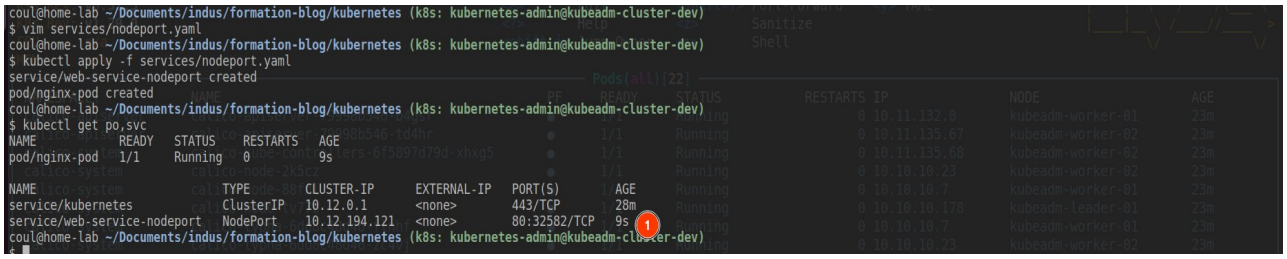
```
apiVersion: v1
kind: Service
metadata:
  name: web-service-nodeport
  labels:
    app: web
    env: production
spec:
  selector:
    app: web
  ports:
  - name: name-of-service-port
    protocol: TCP
    port: 80
    targetPort: http-web-svc
  type: NodePort  # nodePort: 30007

---

apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: web
    env: production
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
```

```
      - containerPort: 80
        name: http-web-svc
```

```
# creer le ficher nodeport.yaml et appliquer le
kubectl apply -f nodeport.yaml
kubectl get pod,svc
```



On voit bien le service a pris le port aléatoire 32582

Pour accéder à l'application, on prend l'ip de n'importe quel node: http://node_ip:32582

Dans cet exemple, le service web-service-nodeport expose l'application associée aux pods avec l'étiquette app: web en utilisant le nodeport 32582 à l'extérieur du cluster.

On peut également fixer ce port avec ce exemple:

```
apiVersion: v1
kind: Service
metadata:
  name: web-service-nodeport
  labels:
    app: web
    env: production
spec:
  selector:
    app: web
  ports:
  - name: name-of-service-port
    protocol: TCP
    port: 80
    targetPort: http-web-svc
    nodePort: 30007
```

- Le service type LoadBalancer

Le service LoadBalancer expose l'application à l'extérieur du cluster en provisionnant un équilibreur de charge externe. Voici un exemple de configuration YAML pour un service LoadBalancer :

```
apiVersion: v1
kind: Service
metadata:
  name: web-service-lb
  labels:
    app: web
    env: dev
spec:
  selector:
```

```
    app: web
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer

---

apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: web
    env: dev
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
```

Dans cet exemple, le service web-service-lb expose l'application associée aux pods avec l'étiquette app: web en utilisant le port 80 à l'extérieur du cluster et provisionne un équilibreur de charge externe.

Pour que ça le service de type loadbalancer fonctionne, il faut installer un addons quand vous êtes on-premise. Mais si c'est le cloud votre cloud provider va piloter le composant cloud-controller-manager et fournir une ip à au service. On vera dans la suite l'utilisation on-premise.

C'est pour ces raisons que le service reste en mode pending. Mais l'application est accessible comme si c'est le service de type nodeport.

- Le service type ExternalName

Le service ExternalName renvoie des noms DNS à l'extérieur du cluster. Voici un exemple de configuration YAML pour un service ExternalName :

```
apiVersion: v1
kind: Service
metadata:
  name: mon-service-external
spec:
  type: ExternalName
  externalName: my.database.example.com

---

apiVersion: v1
```

```
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 49152
  externalIPs:
    - 198.51.100.32
```

Dans cet exemple, le service mon-service-external renvoie les requêtes au nom DNS my.database.example.com à l'extérieur du cluster.

Ces exemples illustrent comment utiliser différents types de services pour exposer des applications dans un environnement Kubernetes. Chaque type de service offre des avantages spécifiques en fonction des besoins de déploiement et de connectivité.

- Le service type Headless

Un service "headless" (sans tête) est un type de service dans Kubernetes qui ne fournit pas de service d'équilibrage de charge (LoadBalancer) ni de service de type ClusterIP (ClusterIP service). Au lieu de cela, un service headless est utilisé pour obtenir des adresses IP directes des pods associés à une application sans introduire de proxy réseau (comme un LoadBalancer ou un ClusterIP) entre les clients et les pods.

Voici comment un service headless fonctionne :

1. P**as de ClusterIP** : Contrairement aux services normaux, un service headless n'a pas d'adresse IP ClusterIP attribuée. Cela signifie qu'il n'y a pas de proxy réseau intermédiaire entre les clients et les pods.

2. **DNS** : Un service headless est associé à une entrée DNS spéciale. Par exemple, si vous avez un service headless nommé "mon-service", le DNS de ce service serait "mon-service.namespace.svc.cluster.local". Cette entrée DNS pointe vers les adresses IP directes des pods associés au service.

3. A**dresses IP directes** : Lorsqu'un client résout le DNS du service headless, il obtient une liste d'adresses IP directes des pods. Les clients peuvent alors se connecter directement aux pods à l'aide de ces adresses IP.

Les services headless sont utiles dans divers scénarios, notamment :

- **StatefulSets** : Les StatefulSets sont souvent utilisées pour des applications nécessitant des identifiants uniques et stables pour chaque pod (par exemple, les bases de données). Les services headless garantissent que chaque pod a une adresse IP stable et prévisible.

- **Services de découverte** : Les services headless sont utilisés pour exposer des applications qui fournissent des informations de découverte aux autres parties de l'infrastructure.

- **Accès direct aux pods** : Ils permettent un accès direct aux pods sans passer par un équilibrage de charge, ce qui peut être utile pour le débogage ou la communication entre des parties spécifiques de l'application.

En résumé, un service headless dans Kubernetes est un type de service qui permet d'obtenir des adresses IP directes des pods associés à une application via le DNS, sans utiliser de proxy réseau. Cela peut être très utile pour des cas d'utilisation spécifiques où une communication directe avec les pods est nécessaire.

```
apiVersion: v1
kind: Service
metadata:
  name: my-headless-service
  namespace: mon-espace-de-noms
spec:
  clusterIP: None   # C'est ce qui indique qu'il s'agit d'un service headless
  selector:
    app: mon-app     # Sélectionne les pods associés à ce service
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

```
nslookup my-headless-service.mon-espace-de-noms.svc.cluster.local
```

- **DNS dans Kubernetes**

Kubernetes offre un service de résolution de noms (DNS) intégré qui permet aux conteneurs de communiquer entre eux par le biais de noms au lieu d'adresses IP. Le service DNS permet de découvrir dynamiquement les adresses IP des pods et des services au sein du cluster. Dans cluster notre test, on utilise coredns

Exemple détaillé pour la configuration du service DNS

Pour illustrer l'utilisation du service DNS dans Kubernetes, voici un exemple de déploiement qui exécute deux pods et les expose via un service, avec utilisation du service DNS pour la communication entre les pods :

- Créez un fichier YAML pour le déploiement :

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-dns
  labels:
    app: web
    env: dns
spec:
  containers:
  - name: nginx-container
    image: nginx:latest

---

apiVersion: v1
kind: Service
metadata:
  name: dns-service
  labels:
    app: web
    env: dns
spec:
  selector:
    app: web
  ports:
```

```
    - protocol: TCP
      port: 80
      targetPort: 80
```

- Appliquez les fichiers YAML à votre cluster :

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

- Obtenez les adresses IP des pods à l'aide de la résolution DNS :

Les pods exécutés par le déploiement sont automatiquement assignés des noms de sous-domaine en fonction de leur étiquette, par exemple nom-du-pod.mon-service.default.svc.cluster.local

Vous pouvez utiliser ces noms de domaine pour communiquer entre les pods, comme illustré dans l'exemple suivant :

```
kubectl exec -it nginx-dns -- /bin/sh

# À l'intérieur du conteneur du pod
curl nom-du-pod-2.mon-service.default.svc.cluster.local
```

Cet exemple montre comment configurer et utiliser le service DNS intégré de Kubernetes pour la résolution des noms entre les pods. Les noms de domaine générés sont basés sur les étiquettes des pods et permettent une communication efficace à l'intérieur du cluster.

Gestion des Espaces : Namespaces

- Namespace : séparation logique pour organiser et isoler les ressources dans le cluster.
- Namespace avec labels
- Namespace avec annotations
- Ajout de ResourceQuota

Un namespace dans Kubernetes est une manière de diviser logiquement un cluster en plusieurs espaces isolés. Cela permet de gérer, organiser et isoler les ressources (comme les Pods, Services, ConfigMaps, etc.) pour différents projets, équipes ou environnements (par exemple, développement, test et production) dans un même cluster. Les objets pris en compte sont de type << namespaced >>

Pourquoi utiliser des Namespaces ?

Les namespaces sont particulièrement utiles pour :

- **Isoler les ressources** : Limiter l'impact des ressources d'une équipe ou d'un environnement sur les autres.
- **Gérer les autorisations** : Assigner des rôles spécifiques pour le contrôle d'accès (RBAC) à certains utilisateurs ou équipes dans chaque namespace.
- **Appliquer des quotas** : Limiter les ressources (CPU, mémoire, nombre de Pods, etc.) pour éviter qu'un namespace ne consomme tout le cluster.
- **Organiser le cluster** : Faciliter la gestion et l'organisation des ressources dans des projets différents dans un même cluster.

Exemple de namespace : test-ns

```
apiVersion: v1
kind: Namespace
metadata:
  name: test-ns

---

apiVersion: v1
kind: Pod
metadata:
  namespace: test-ns
  annotations:
    prometheus.io/scrape: true
    prometheus.io/port: 8080
    team: ops
  labels:
    app: monitoring
  name: monitoring-pod
spec:
  containers:
  - image: prometheus/node-exporter:latest
    name: monitoring-pod
```

Annotation sur un Namespace :

De la même manière, vous pouvez ajouter des Annotations (Annotations) à un Namespace pour fournir des informations supplémentaires. Voici comment ajouter une Annotation à un Namespace :

```
apiVersion: v1
kind: Namespace
metadata:
  name: front
  labels:
    environment: production
  annotations:
    description: "Cet espace de noms est utilisé pour l'environnement de
production."

---

apiVersion: v1
kind: Pod
metadata:
  namespace: front
  name: nginx-pod
  labels:
    app: web
    env: production
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
```

Dans cet exemple, nous avons ajouté une Annotation "description" avec la valeur "Cet espace de noms est utilisé pour l'environnement de production" au Namespace "front". Les Annotations peuvent être utilisées pour fournir des détails supplémentaires sur un Namespace.

Ces exemples montrent comment définir un Namespace, ajouter des Labels pour le catégoriser et ajouter des Annotations pour fournir des informations supplémentaires. Vous pouvez personnaliser davantage ces éléments en fonction de vos besoins spécifiques.

Namespace avec ResourceQuota:

Namespace avec une limite de 2 Pods à l'aide d'un ResourceQuota.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-quota
  namespace: dev-ns
spec:
  hard:
    pods: "2"

---

apiVersion: v1
kind: Namespace
metadata:
  name:  dev-ns
  labels:
    environment: production
  annotations:
    description: "Cet espace de noms est utilisé pour l'environnement de
production."

---

apiVersion: v1
kind: Pod
metadata:
  namespace:  dev-ns
  name: nginx-pod
  labels:
    app: web
    env: production
spec:
  containers:
  - name: nginx-container
    image: nginx:latest

---

apiVersion: v1
kind: Pod
metadata:
  namespace:  dev-ns
  name: web-pod
  labels:
    app: web
    env: production
spec:
  containers:
  - name: nginx-container
    image: caddy:latest
```

Volumes hostPath et emptyDir

Dans kuernetes, les volumes sont utilisés pour stocker des données persistantes ou temporaires que des conteneurs peuvent partager ou utiliser. Deux types de volumes fréquemment utilisés sont hostPath et emptyDir. Chacun de ces volumes sert des objectifs différents en fonction de la durée de vie et de la nature des données.

1. hostPath Volume

Le volume hostPath permet à un pod d'accéder à un répertoire sur le système de fichiers du nœud hôte (machine physique ou virtuelle où tourne Kubernetes). Ce type de volume est utile lorsqu'il faut partager des données entre un conteneur et son hôte ou lorsqu'on souhaite accéder à un répertoire spécifique sur le nœud.

- **Caractéristiques :**

- **Accès au système de fichiers de l'hôte** : Un hostPath pointe vers un répertoire sur le nœud hôte. Le pod peut accéder à ce répertoire ou à des fichiers sur l'hôte.
- **Partage de données entre l'hôte et les pods** : Ce volume permet de partager des fichiers ou des répertoires entre les conteneurs et le système d'exploitation hôte.
- **Utilisation limitée** : C'est souvent utilisé dans des environnements où des données doivent être partagées entre les conteneurs et le nœud, mais il est limité dans un contexte de production Kubernetes car l'accès aux fichiers de l'hôte peut créer des problèmes de sécurité.

- **Exemple d'utilisation avec nginx:**

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-example
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: hostpath-volume
  volumes:
  - name: hostpath-volume
    hostPath:
      path: /var/www/data
      type: DirectoryOrCreate
```

- Dans cet exemple :

- path: /var/www/data : Il s'agit du répertoire sur l'hôte auquel le conteneur Nginx va accéder.
- mountPath: /usr/share/nginx/html : Ce répertoire dans le conteneur sera mappé avec le répertoire /var/www/data sur l'hôte.
- type: DirectoryOrCreate : Cela spécifie que le volume est un répertoire et sera créé automatiquement. Il existe d'autres options comme File, Socket, etc.

```
kubectl apply -f hostpath-volume.yaml

kubectl get po -o wide
```

```
coul@home-lab ~ (k8s: kubernetes-admin@kubeadm-cluster-dev)
$ kubectl get po -o wide
NAME              READY   STATUS    RESTARTS   AGE   IP             NODE              NOMINATED NODE   READINESS GATES
hostpath-example  1/1     Running   0          25s   10.11.132.53   kubeadm-worker-01 <none>           <none>
coul@home-lab ~ (k8s: kubernetes-admin@kubeadm-cluster-dev)
```

- **Exepmple hostPath FileOrCreate configuration example**

Le manifeste suivant définit un Pod qui monte /var/local/data à l'intérieur du conteneur du Pod. Si le nœud n'a pas déjà de chemin /var/local/data, kubelet le crée en tant que répertoire, puis le monte dans le Pod.

Si un /var/local/data élément existe déjà mais n'est pas un répertoire, le Pod échoue. De plus, kubelet tente de créer un fichier nommé /var/local/data/data.txt à l'intérieur de ce répertoire (vu depuis l'hôte) ; si quelque chose existe déjà à ce chemin et n'est pas un fichier normal, le Pod échoue.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-webserver
spec:
  os: { name: linux }
  nodeSelector:
    kubernetes.io/os: linux
  containers:
  - name: test-webserver
    image: registry.k8s.io/test-webserver:latest
    volumeMounts:
    - mountPath: /var/www/data
      name: data-dir
    - mountPath: /var/www/data/data.txt
      name: data-file
  volumes:
  - name: data-dir
    hostPath:
      # Ensure the file directory is created.
      path: /var/www/data
      type: DirectoryOrCreate
  - name: data-file
    hostPath:
      path: /var/www/data/data.txt
      type: FileOrCreate
```

- **Cas d’utilisation** :

- Accès aux fichiers de configuration sur le nœud hôte.
- Accès aux journaux ou fichiers de données partagés entre l’hôte et les pods.
- **Inconvénients** :

- Il peut y avoir des risques de sécurité, car les pods ont accès à un répertoire sur le nœud hôte.
- Ce type de volume n’est pas portable entre les nœuds, car il est directement lié à l’infrastructure physique ou virtuelle sous-jacente.

```
kubectl apply -f hostpath-volume.yaml

kubectl get po -o wide
```

2. **EmptyDir Volume:**

Le volume emptyDir est utilisé pour stocker des données temporaires qui sont créées et supprimées avec le pod. Ce volume est vidé lorsque le pod est supprimé. Il est utile pour les données temporaires telles que les fichiers de cache, les fichiers temporaires ou pour les cas où les conteneurs d’un pod doivent partager des données en interne.

- **Caractéristiques** :

- **Données temporaires** : Les données stockées dans un emptyDir ne sont pas persistantes et sont effacées dès que le pod est supprimé.
- **Partage de données entre conteneurs** : Si plusieurs conteneurs dans un pod ont besoin de partager des données, ils peuvent le faire facilement via un emptyDir.
- **Pas de persistance au-delà du pod** : Ce volume est utile pour des données qui ne nécessitent pas de persistance entre les redémarrages du pod.

- Exemple1 données de cache:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      sizeLimit: 500Mi
```

- Exemple2:

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-emptydir
spec:
  containers:
  - name: container-1
    image: busybox:latest
    command: ["sh", "-c", "echo 'Hello from Container 1' > /data/file.txt"]
    volumeMounts:
    - name: emptydir-volume
      mountPath: /data
  - name: container-2
    image: busybox:latest
    command: ["cat", "/data/file.txt"]
    volumeMounts:
    - name: emptydir-volume
      mountPath: /data
  volumes:
  - name: emptydir-volume
    emptyDir: {}
```

- Exemple3:

```
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-example
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
```

```
    - mountPath: /usr/share/nginx/html
      name: emptydir-volume
  volumes:
  - name: emptydir-volume
    emptyDir: {}
```

- **Dans cet exemple** :

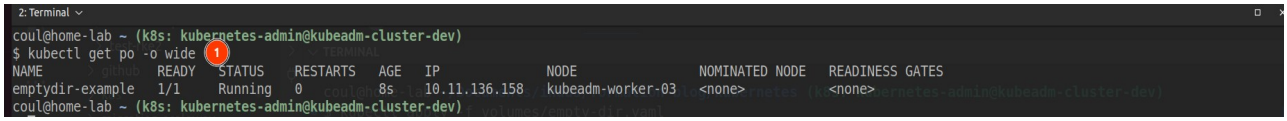
- emptyDir: {} : Le volume emptyDir est configuré sans paramètres supplémentaires, ce qui signifie qu'il sera simplement un répertoire temporaire monté dans le conteneur.
- mountPath: /usr/share/nginx/html : Ce répertoire est monté dans le conteneur Nginx pour stocker des fichiers temporaires.
- **Cas d'utilisation** :

- Fichiers temporaires : Stockage de fichiers temporaires générés par un processus dans un pod, comme des logs ou des fichiers de traitement.
- Cache partagé entre conteneurs : Lorsque plusieurs conteneurs d'un pod doivent partager des données temporaires (par exemple, des données de cache).
- **Inconvénients** :

- Les données sont perdues lorsque le pod est supprimé. Cela rend emptyDir inapproprié pour le stockage de données persistantes ou critiques.

```
kubectl apply -f  empty-dir.yaml

kubectl get po -o wide
```



3. **Comparaison : hostPath vs emptyDir**

Caractéristique	hostPath	emptyDir
Type de stockage	Répertoire ou fichier sur l'hôte	Répertoire temporaire dans le pod
Durée de vie des données	Les données persistent au-delà de la vie du pod	Les données sont perdues lorsque le pod est supprimé
Partage de données	Partage entre l'hôte et les pods	Partage entre les conteneurs d'un même pod
Cas d'utilisation	Accès aux fichiers ou répertoires sur l'hôte	Fichiers temporaires, cache, partage interne entre conteneurs
Portabilité	Lié à l'hôte et non portable	Portable (uniquement dans le pod)
Sécurité	Risque de sécurité si mal configuré	Pas de risque de sécurité spécifique

Conclusion

- hostPath est utilisé lorsque vous avez besoin d'accéder à des répertoires ou fichiers spécifiques sur l'hôte, mais il doit être utilisé avec précaution, surtout en production, en raison des risques de sécurité.

- emptyDir est adapté pour stocker des données temporaires ou partagées entre les conteneurs d'un même pod, mais les données ne sont pas persistantes une fois que le pod est supprimé.

Chaque type de volume a des cas d'utilisation spécifiques en fonction de la persistance, de la sécurité et de la portée des données.

INSTALLATION de KNATIVE

Pré-requis pour l'installation
==========
Une instance kubernetes installé
Minikube est un bon candidat (voir installation ci dessus).

Installer knative
==========

```
$> wget https://github.com/knative/client/releases/download/knative-v1.20.0/kn-linux-amd64
$> mv kn-linux-amd64 kn
$> chmod a+x kn
$> sudo mv kn /usr/local/bin
$> kn version
```

Installer knative quickstart
==========

```
$> wget https://github.com/knative-extensions/kn-plugin-quickstart/releases/download/knative-v1.20.0/kn-quickstart-linux-amd64
$> mv kn-quickstart-linux-amd64 kn-quickstart
$> chmod a+x kn-quickstart
$> sudo mv kn-quickstart /usr/local/bin

$> kn quickstart --help
```

Run quickstart
==========

```
$> kn quickstart minikube
```

En fin d'installation à la demande de quickstart, lancer dans une seconde console et exécutez ceci

```
$> minikube tunnel --profile knative
```

Toujours laisser actif lors de l'utilisation de knative.

Installer la version conforme à minikube de kubectl (option)
==========
Installer kubectl à partir de minikube

```
$> minikube kubectl -- get pods -A
```

Note :
Pour utiliser kubectl de minikube, on commencera toujours l'appel comme ceci :

```
$> minikube kubectl -- XxXxXxXxXxXxXx
```

Exemple : `$> minikube kubectl -- get pods -n=monespace`

Installer knative function CLI
==========

```
$> wget https://github.com/knative/func/releases/download/knative-v1.17.0/func_linux_amd64
$> mv func_linux_amd64 func
$> chmod a+x func
$> sudo mv func /usr/local/bin
```

```
$> func version
```

Tester une fonction (https://knative.dev/docs/getting-started/first-service/)
==========

```
$> kn service create hello --image ghcr.io/knative/helloworld-go:latest --port
8080 --env TARGET=World
```

Service 'hello' created to latest revision 'hello-00001' is available at URL:
http://hello.default.10.110.144.98.sslip.io
curl http://hello.default.10.110.144.98.sslip.io
Hello World!

Démarrer/arrêter minikube pour utiliser knative
==========

```
$> minikube start
$> minikube tunnel --profile knative
```

Arrêter minikube
==========

```
$> minikube stop
```