

# TP02 - Orchestration et conteneurs

TP effectué par Vincent LAGOGUÉ, Tom THIOULOUSE, Alexis PLESSIAS, David TEJEDA et Thomas PEUGNET.

## Installation

Nous commençons par effectuer l'installation de Kubernetes par le gestionnaire de paquet Homebrew (macos) à l'aide de ces commandes.

```
# Minikube
$ brew install minikube

# Kubectl
$ curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl"

# Ajout de l'alias
echo "alias k='kubectl'" >> ~/.zshrc

# Installation de kubecolor
$ brew install hidetatz/tap/kubecolor
```

```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
zsh completions have been installed to:
/usr/local/share/zsh/site-functions
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io
/release/stable.txt)/bin/darwin/amd64/kubectl"

% Total    % Received % Xferd  Average Speed   Time    Time     Time  Cu
rrent
                                Dload  Upload  Total  Spent    Left  Sp
eed
0         0    0     0    0     0      0      0  --:--:-- --:--:-- --:--:--
100    138  100    138    0     0    422     0  --:--:-- --:--:-- --:--:--
428
0 50.1M    0     0    0     0      0      0  --:--:-- 0:00:01 --:--:--
2 50.1M    2 1055k    0     0   529k     0  0:01:36 0:00:01 0:01:35 1
4 50.1M    4 2063k    0     0   774k     0  0:01:06 0:00:02 0:01:04 1
10 50.1M   10 5150k    0     0  1222k     0  0:00:42 0:00:04 0:00:38 1
11 50.1M   11 6158k    0     0  1260k     0  0:00:40 0:00:04 0:00:36 1
16 50.1M   16 8352k    0     0  1477k     0  0:00:34 0:00:05 0:00:29 1
19 50.1M   19 10.0M    0     0  1456k     0  0:00:35 0:00:07 0:00:28 1
21 50.1M   21 11.0M    0     0  1463k     0  0:00:35 0:00:07 0:00:28 1
25 50.1M   25 13.0M    0     0  1481k     0  0:00:34 0:00:08 0:00:26 1
29 50.1M   29 15.0M    0     0  1578k     0  0:00:32 0:00:09 0:00:23 1
32 50.1M   32 16.1M    0     0  1502k     0  0:00:34 0:00:10 0:00:24 1
33 50.1M   33 17.0M    0     0  1497k     0  0:00:34 0:00:11 0:00:23 1
41 50.1M   41 21.0M    0     0  1642k     0  0:00:31 0:00:13 0:00:18 1
43 50.1M   43 22.0M    0     0  1631k     0  0:00:31 0:00:13 0:00:18 1
53 50.1M   53 27.0M    0     0  1894k     0  0:00:27 0:00:14 0:00:13 2
64 50.1M   64 32.2M    0     0  2110k     0  0:00:24 0:00:15 0:00:09 3
71 50.1M   71 36.0M    0     0  2220k     0  0:00:23 0:00:16 0:00:07 3
80 50.1M   80 40.1M    0     0  2330k     0  0:00:22 0:00:17 0:00:05 4
87 50.1M   87 44.0M    0     0  2417k     0  0:00:21 0:00:18 0:00:03 4
100 50.1M  100 50.1M    0     0  2627k     0  0:00:19 0:00:19 --:--:-- 4
794k
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
vim ~/.zshrc
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
source ~/.zshrc
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

# Premiers pas

Lancement de minikube avec la commande `minikube start`.

```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s

> gcr.io/k8s-minikube/kicbase...: 356.17 MiB / 453.90 MiB 78.47% 27.
> preloaded-images-k8s-v18-v1...: 400.80 MiB / 403.35 MiB 99.37% 43.
> gcr.io/k8s-minikube/kicbase...: 364.99 MiB / 453.90 MiB 80.41% 27.
> preloaded-images-k8s-v18-v1...: 403.35 MiB / 403.35 MiB 100.00% 45
.57 M
> gcr.io/k8s-minikube/kicbase...: 373.87 MiB / 453.90 MiB 82.37% 28.
> gcr.io/k8s-minikube/kicbase...: 383.86 MiB / 453.90 MiB 84.57% 28.
> gcr.io/k8s-minikube/kicbase...: 392.61 MiB / 453.90 MiB 86.50% 28.
> gcr.io/k8s-minikube/kicbase...: 400.69 MiB / 453.90 MiB 88.28% 29.
> gcr.io/k8s-minikube/kicbase...: 408.87 MiB / 453.90 MiB 90.08% 29.
> gcr.io/k8s-minikube/kicbase...: 417.46 MiB / 453.90 MiB 91.97% 29.
> gcr.io/k8s-minikube/kicbase...: 426.66 MiB / 453.90 MiB 94.00% 30.
> gcr.io/k8s-minikube/kicbase...: 433.98 MiB / 453.90 MiB 95.61% 30.
> gcr.io/k8s-minikube/kicbase...: 442.13 MiB / 453.90 MiB 97.41% 30.
> gcr.io/k8s-minikube/kicbase...: 450.99 MiB / 453.90 MiB 99.36% 31.
> gcr.io/k8s-minikube/kicbase...: 453.90 MiB / 453.90 MiB 100.00% 41
.76 M
🔥 Création de docker container (CPU=2, Memory=1952Mo) ...
❗ Ce container rencontre des difficultés pour accéder à https://registry
.k8s.io
💡 Pour extraire de nouvelles images externes, vous devrez peut-être conf
igurer un proxy : https://minikube.sigs.k8s.io/docs/reference/networking/p
roxy/
🐳 Préparation de Kubernetes v1.28.3 sur Docker 24.0.7...
  ■ Génération des certificats et des clés
  ■ Démarrage du plan de contrôle ...
  ■ Configuration des règles RBAC ...
🔗 Configuration de bridge CNI (Container Networking Interface)...
  ■ Utilisation de l'image gcr.io/k8s-minikube/storage-provisioner:v5
🔍 Vérification des composants Kubernetes...
🌟 Modules activés: default-storageclass, storage-provisioner

❗ /usr/local/bin/kubectl est la version 1.30.0, qui peut comporter des i
ncompatibilités avec Kubernetes 1.28.3.
  ■ Vous voulez kubectl v1.28.3 ? Essayez 'minikube kubectl -- get pods
-A'
🏁 Terminé ! kubectl est maintenant configuré pour utiliser "minikube" cl
uster et espace de noms "default" par défaut.
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

La commande `minikube status` donne le résultat suivant.

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>  
└─ minikube status  
minikube  
type: Control Plane  
host: Running  
kubelet: Running  
apiserver: Running  
kubeconfig: Configured
```

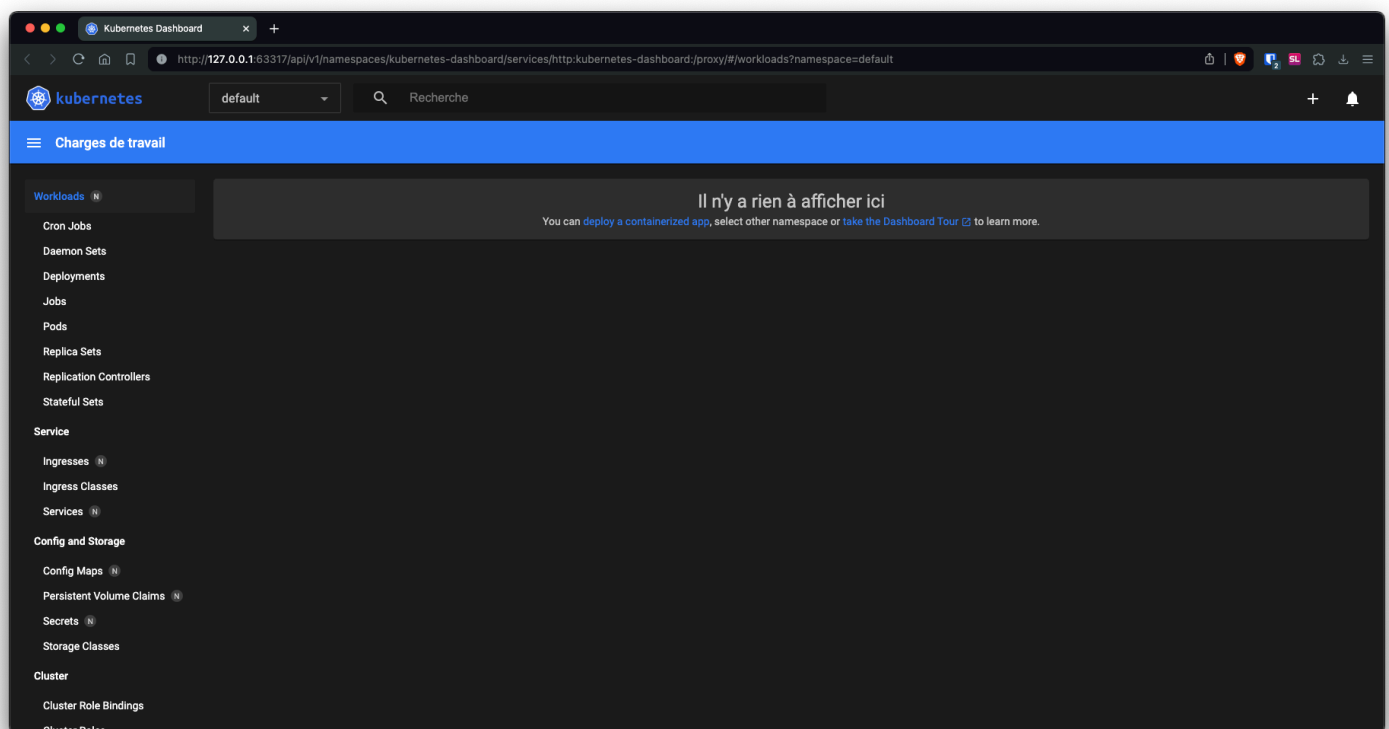
14 ↩

La commande `kubectl cluster-info` donne le résultat suivant.

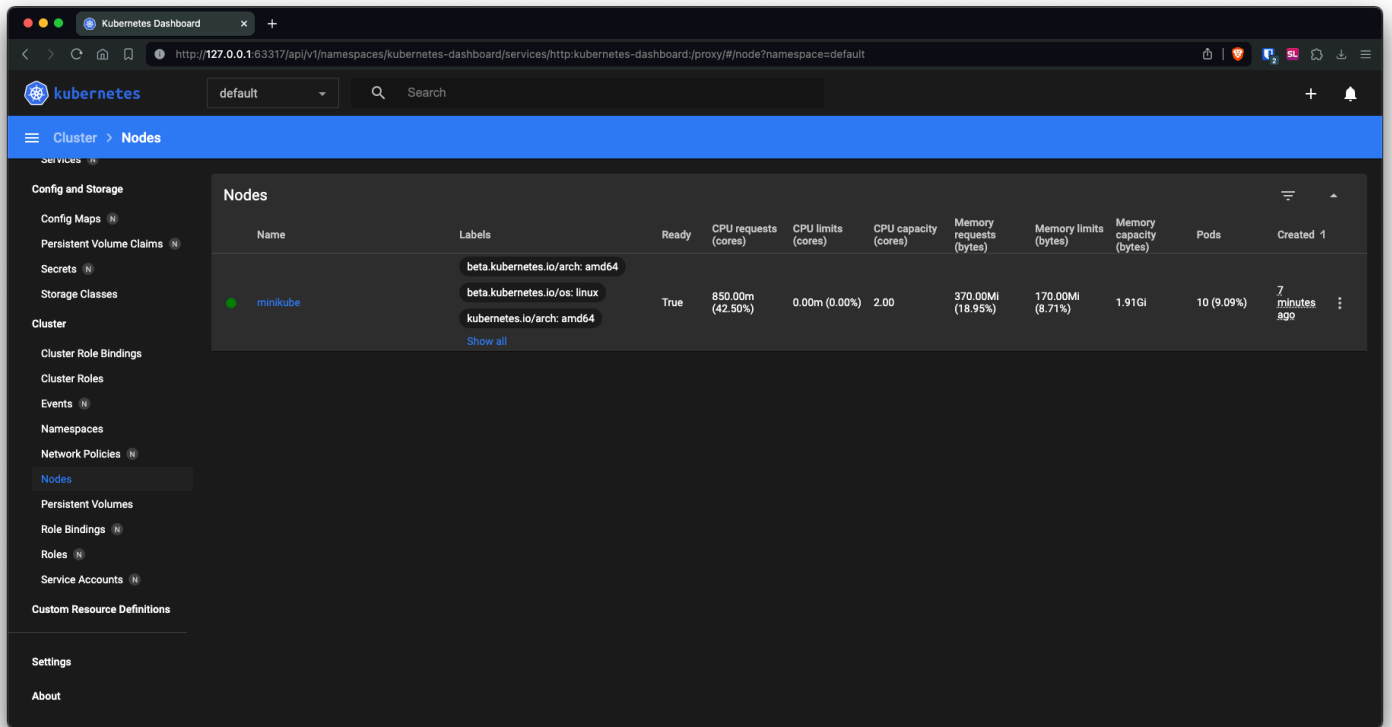
```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>  
└─ kubectl cluster-info  
Kubernetes control plane is running at https://127.0.0.1:32769  
CoreDNS is running at https://127.0.0.1:32769/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy  
  
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Activation du dashboard avec les commandes suivantes: `minikube addons enable dashboard` et `minikube addons enable metrics-server`.

On peut accéder au dashboard avec la commande `minikube dashboard`.



Nous pouvons voir la liste des nodes présents ici.



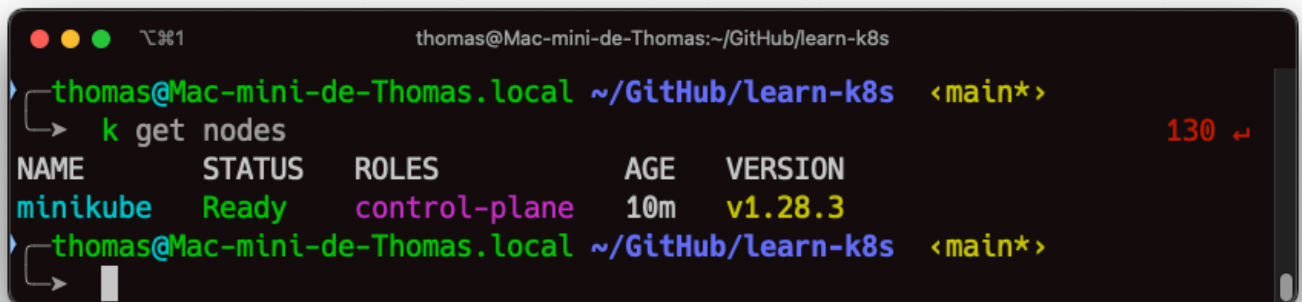
Nous pouvons voir la liste des namespaces présents.

Nous avons donc :

- 1 node: minikube
- 5 namespaces

# Objets Kubernetes

La commande `k get nodes` donne le résultat suivant:



Nous voyons donc un seul et unique node dans notre cluster. C'est normal, nous n'avons pas encore créé de services ou déployé quelque chose.

La commande `k get namespaces` donne le résultat suivant:

```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k get namespaces
NAME                STATUS    AGE
default             Active    12m
kube-node-lease     Active    12m
kube-public         Active    12m
kube-system         Active    12m
kubernetes-dashboard Active    8m14s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Nous avons donc bel et bien 5 namespaces, comme nous l'avions constaté sur le navigateur.

La commande `k describe ns/default` nous donne le résultat suivant:

```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k describe ns/default
Name:         default
Labels:       kubernetes.io/metadata.name=default
Annotations:  <none>
Status:       Active

No resource quota.

No LimitRange resource.
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Il n'y a pas de `quota` ni de `LimitRange`. Nous avons cependant le label `kubernetes.io/metadata.name=default`.

Pour obtenir la définition de notre namespace `default` en yaml, nous utilisons la commande suivante: `k get ns/default -o yaml`

Nous obtenons le résultat suivant :



```
thomas@Mac-mini-de-Thomas: ~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k get ns/default -o yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: "2024-04-18T13:13:12Z"
  labels:
    kubernetes.io/metadata.name: default
  name: default
  resourceVersion: "42"
  uid: a1da6422-018e-4e02-8a10-e047c153acf0
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Nous créons ensuite notre namespace `tp2` à l'aide de la commande suivante:

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k create ns tp2
namespace/tp2 created
```

Pour le supprimer, nous utilisons la commande suivante:

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k delete ns tp2
namespace "tp2" deleted
```

## Pod nginx

Nous créons notre namespace `tp2` à l'aide de la commande suivante:

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k create ns tp2
namespace/tp2 created
```

Puis, nous créons un fichier `pod.yaml` qui aura le contenu suivant:

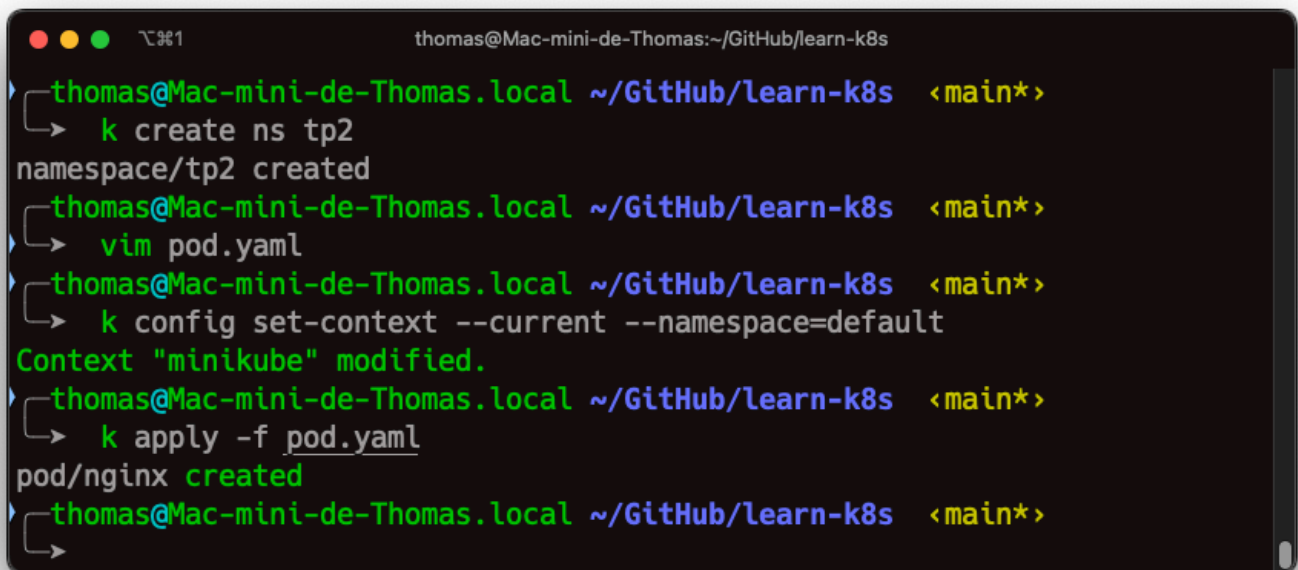
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Etant donné que nous évoluons maintenant exclusivement dans le namespace `tp2`, nous mettons automatiquement toutes nos futures commandes `kubectl` dans ce namespace à l'aide de la commande suivante.

```
$ k config set-context --current --namespace=default
```

Nous appliquons maintenant notre `fichier.yaml` à l'aide de la commande `k apply -f pod.yaml`.

Nous obtenons le résultat suivant:



```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
└─> k create ns tp2
namespace/tp2 created
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
└─> vim pod.yaml
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
└─> k config set-context --current --namespace=default
Context "minikube" modified.
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
└─> k apply -f pod.yaml
pod/nginx created
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
└─>
```

Pour vérifier le bon lancement de notre pod, nous utilisons la commande `k get pods`:



```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k get pod
NAME      READY   STATUS    RESTARTS   AGE
nginx     1/1     Running   0           35s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Pour obtenir davantage d'informations sur nos pods, et en particulier notre pod `nginx`, nous utilisons la commande `k describe pod`:

```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s <main*>
k describe pod
Name:      nginx
Namespace: default
Priority:   0
Service Account: default
Node:      minikube/192.168.49.2
Start Time: Thu, 18 Apr 2024 15:35:07 +0200
Labels:    app=nginx
Annotations: <none>
Status:     Running
IP:         10.244.0.6
IPs:
  IP: 10.244.0.6
Containers:
  nginx:
    Container ID:  docker://47951659f016d00f690e25f312ddedc55446acfcac6ba69b6e41bf6db55f930f
    Image:         nginx
    Image ID:      docker-pullable://nginx@sha256:d2cb0992f098fb075674730da5e1c6cccd4890516e448a1db96e0245c1b7fca
    Port:         80/TCP
    Host Port:    0/TCP
    State:        Running
      Started:    Thu, 18 Apr 2024 15:35:19 +0200
    Ready:        True
    Restart Count: 0
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-hwt5n (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-hwt5n:
    Type:          Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds: 3607
    ConfigMapName:    kube-root-ca.crt
    ConfigMapOptional: <nil>
```

Nous avons donc un pod `nginx` en status `Running` et ayant en `ContainerID` (pour `nginx`) `docker://47951659f016d00f690e25f312ddedc55446acfcac6ba69b6e41bf6db55f930f` et l'adresse IP `10.244.0.6`.

Nous pouvons attacher un shell à notre conteneur à l'aide de la commande `k exec pod/nginx -it -- bash`.

```
kubecolor exec pod/nginx -it -- bash

thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k exec pod/nginx -it -- bash
root@nginx:/#
```

Il y a 3 processus `nginx`. Avoir un conteneur minimaliste permet d'avoir des pods moins demandeur en performance.

```
kubecolor exec pod/nginx -it -- bash

root@nginx:/# ps -edf
UID          PID    PPID  C STIME TTY          TIME CMD
root           1         0  0 14:33 ?           00:00:00 nginx: master process
nginx        29         1  0 14:33 ?           00:00:00 nginx: worker process
nginx        30         1  0 14:33 ?           00:00:00 nginx: worker process
root         31         0  0 14:36 pts/0       00:00:00 bash
root        237        31  0 14:38 pts/0       00:00:00 ps -edf
root@nginx:/#
```

Nous supprimons notre pod créé par notre `fichier.yaml` à l'aide de la commande `k delete -f pod.yaml`

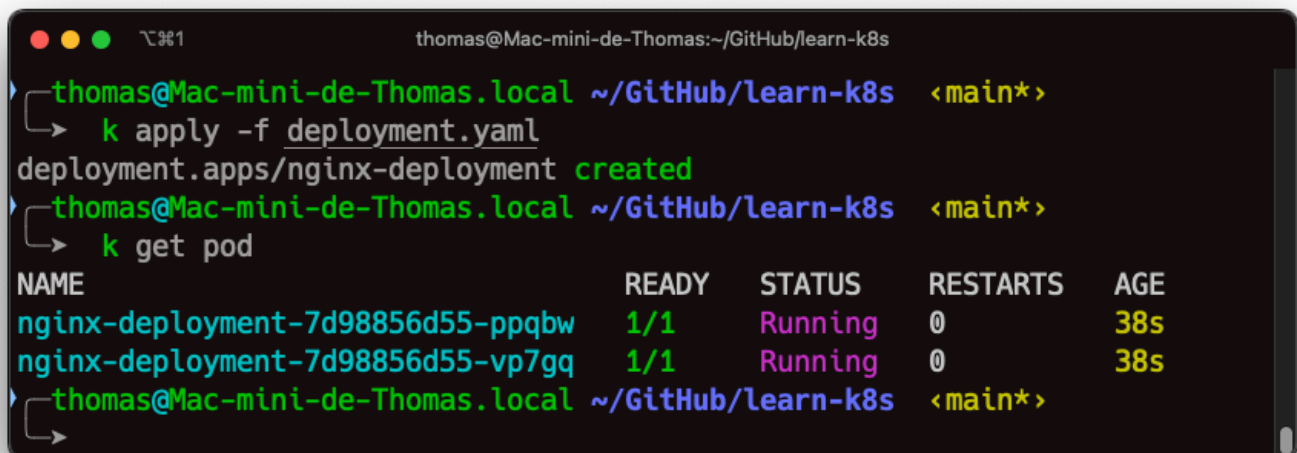
## Déploiement

Nous créons un fichier `deployment.yaml` qui aura le contenu suivant:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
          resources:
```

```
limits:
  memory: 512Mi
  cpu: "1"
requests:
  memory: 256Mi
  cpu: "0.2"
```

Nous appliquons ce déploiement à notre cluster à l'aide de la commande `k apply -f deployment.yaml`. Ce fichier nous indique qu'il y aura 2 pods de créés. Le retour de la commande nous indique le nom de notre déploiement.



```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
➤ k apply -f deployment.yaml
deployment.apps/nginx-deployment created
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
➤ k get pod
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-7d98856d55-ppqbw   1/1     Running   0           38s
nginx-deployment-7d98856d55-vp7gq   1/1     Running   0           38s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Pour vérifier le status de de notre déploiement, nous obtenons le résultat suivant:

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
➤ k rollout status deployment.apps/nginx-deployment
deployment "nginx-deployment" successfully rolled out
```

L'état de notre déploiement est `successfully rolled out`.

Nous obtenons la liste de replicaset à l'aide de la commande `k get rs`. Nous constatons que nous avons un replicaset nommé `nginx-deployment-7d98856d55`.

Pour avoir davantage d'informations sur notre déploiement, nous utilisons la commande `k describe deployment` et obtenons le résultat suivant :

```
thomas@Mac-mini-de-Thomas: ~/GitHub/learn-k8s
k describe deployment
Name: nginx-deployment
Namespace: default
CreationTimestamp: Thu, 18 Apr 2024 15:48:58 +0200
Labels: <none>
Annotations: deployment.kubernetes.io/revision: 1
Selector: app=nginx
Replicas: 2 desired | 2 updated | 2 total | 2 available | 0
unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels: app=nginx
  Containers:
    nginx:
      Image: nginx
      Port: 80/TCP
      Host Port: 0/TCP
      Limits:
        cpu: 1
        memory: 512Mi
      Requests:
        cpu: 200m
        memory: 256Mi
      Environment: <none>
      Mounts: <none>
  Volumes: <none>
  Node-Selectors: <none>
  Tolerations: <none>
Conditions:
  Type           Status  Reason
  ----           -
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet: nginx-deployment-7d98856d55 (2/2 replicas created)
Events:
  Type           Reason             Age           From           Message
  ----           -
  ...
```

Nous modifions donc notre fichier `deployment.yaml` pour mettre le nombre de replicas à 10.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
```

```
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 10
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
          resources:
            limits:
              memory: 512Mi
              cpu: "1"
            requests:
              memory: 256Mi
              cpu: "0.2"
```

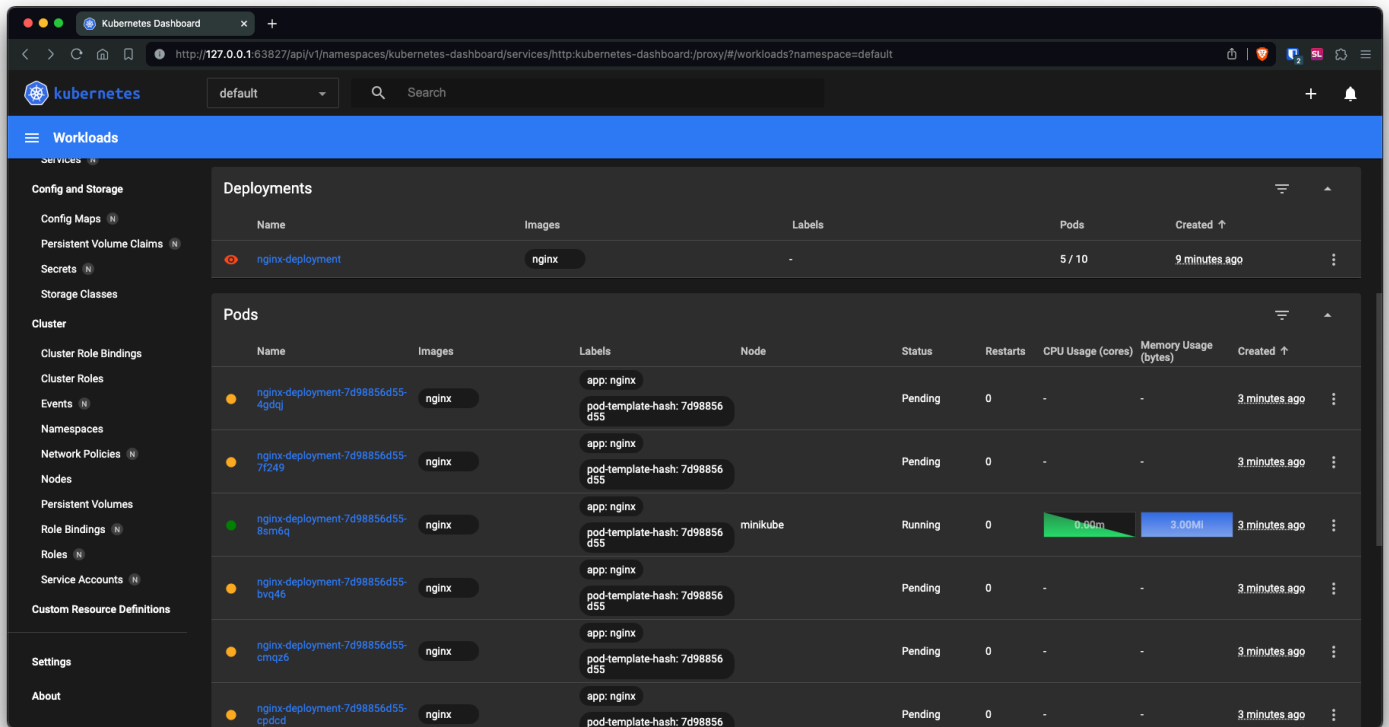
Nous obtenons la liste des pods déployés à l'aide de la commande `k get pod`, qui nous donne le résultat suivant:

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k get pod
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-7d98856d55-2dcj6	1/1	Running	0	54s
nginx-deployment-7d98856d55-4gdqj	0/1	Pending	0	53s
nginx-deployment-7d98856d55-7f249	0/1	Pending	0	53s
nginx-deployment-7d98856d55-8sm6q	1/1	Running	0	53s
nginx-deployment-7d98856d55-bvq46	0/1	Pending	0	53s
nginx-deployment-7d98856d55-cmqz6	0/1	Pending	0	53s
nginx-deployment-7d98856d55-cpdcd	0/1	Pending	0	53s
nginx-deployment-7d98856d55-j5x2b	1/1	Running	0	53s
nginx-deployment-7d98856d55-ppqbw	1/1	Running	0	7m23s
nginx-deployment-7d98856d55-vp7gq	1/1	Running	0	7m23s

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

En consultant le dashboard, nous constatons que seuls 5 des 10 pods ont été démarrés.



Les autres n'ont pas été déployés car le CPU n'est pas suffisant.

Pour supprimer notre déploiement, nous utilisons la commande `k delete -f deployment.yaml`.

## Créer un service interne

Nous créons un fichier `service.yaml` qui aura le contenu suivant:

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

Nous appliquons notre service par la commande `k apply -f service.yaml`.

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
└─ k apply -f service.yaml
service/nginx created
```

Nous pouvons lister nos différents services à l'aide de la commande `k get svc`, qui nous donne le résultat suivant:



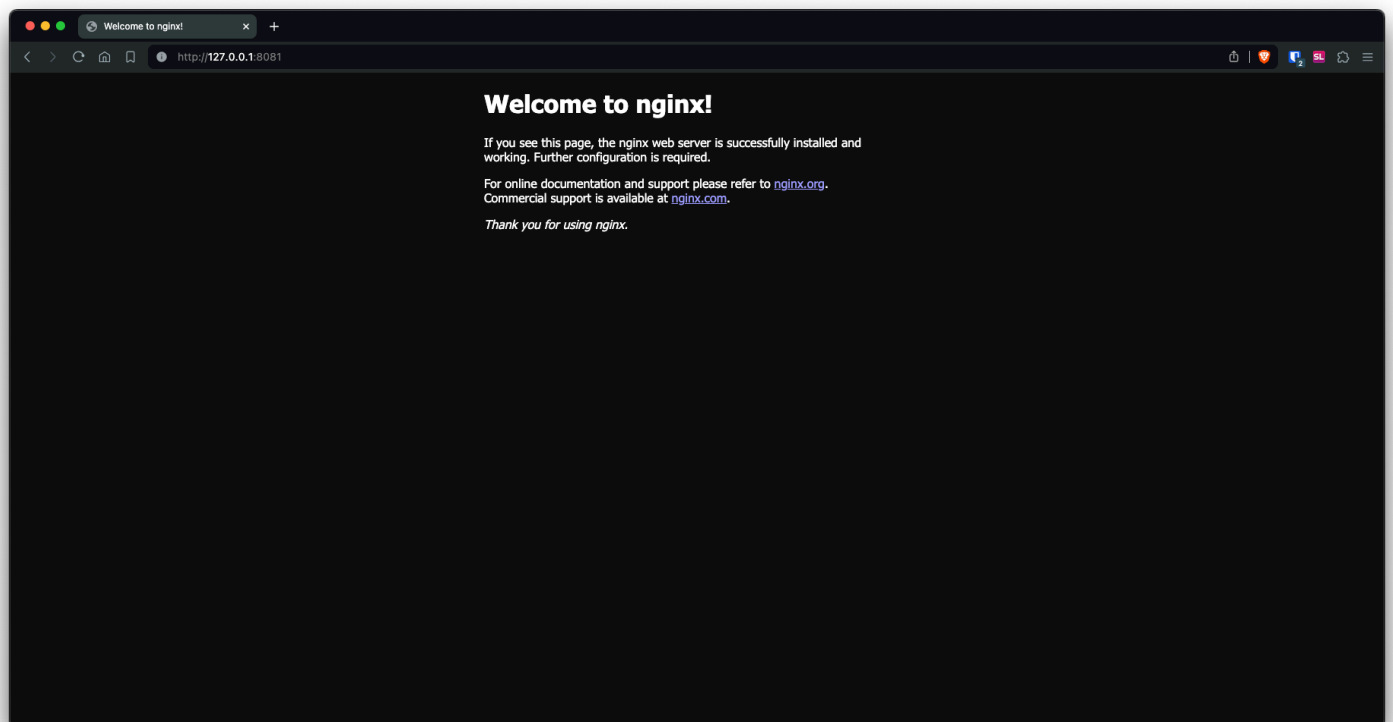
```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k get svc
NAME          TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
kubernetes    ClusterIP     10.96.0.1     <none>         443/TCP    53m
nginx         ClusterIP     10.101.244.39 <none>         80/TCP     16s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Pour obtenir davantage d'informations sur notre service, nous utilisons la commande `k describe service/nginx`.

Nous pouvons donc constater que l'adresse IP est `10.101.244.39`.

Nous configurons du port forwarding entre `8080` et `80` à l'aide de la commande `k port-forward svc/nginx 8080:80`.

```
kubecolor port-forward svc/nginx 8081:80 — 74X5
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k port-forward svc/nginx 8081:80
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80
```



# Loadbalancer

Nous exécutons la commande `minikube tunnel` dans un terminal séparé.

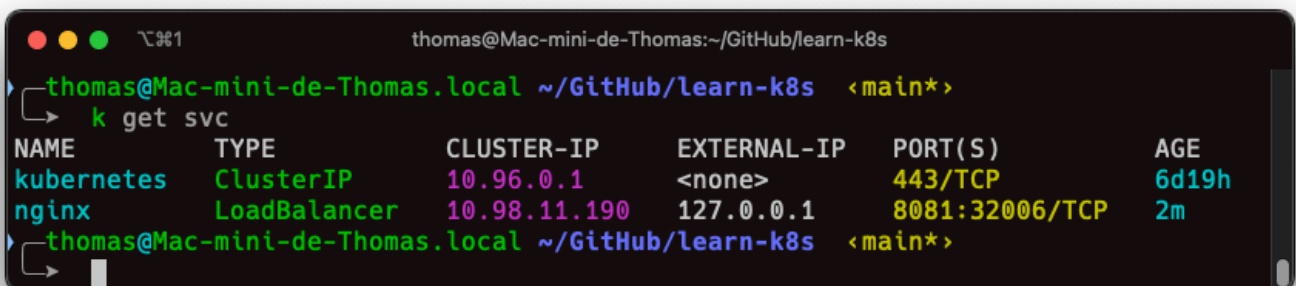
Nous créons un service de loadbalancing avec le contenu suivant:

lb.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 80
```

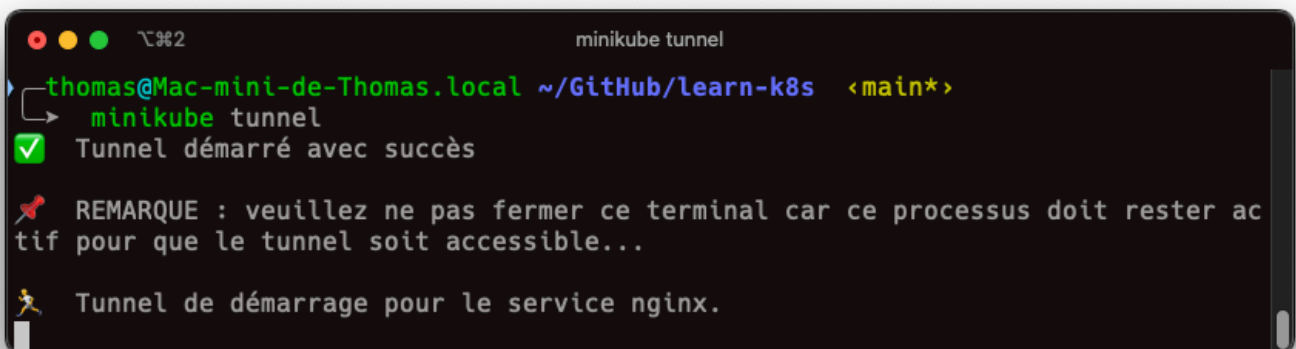
**Note:** Nous utilisons le port 8081 car le 8080 est déjà utilisé par un autre service externe à ce TP.

Nous exécutons ensuite la commande `k get svc` et récupérons l'adresse IP de notre LB.



```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k get svc
NAME          TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes    ClusterIP     10.96.0.1     <none>         443/TCP          6d19h
nginx         LoadBalancer 10.98.11.190  127.0.0.1     8081:32006/TCP   2m
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Nous voyons donc notre adresse IP `127.0.0.1:8081` en External IP.

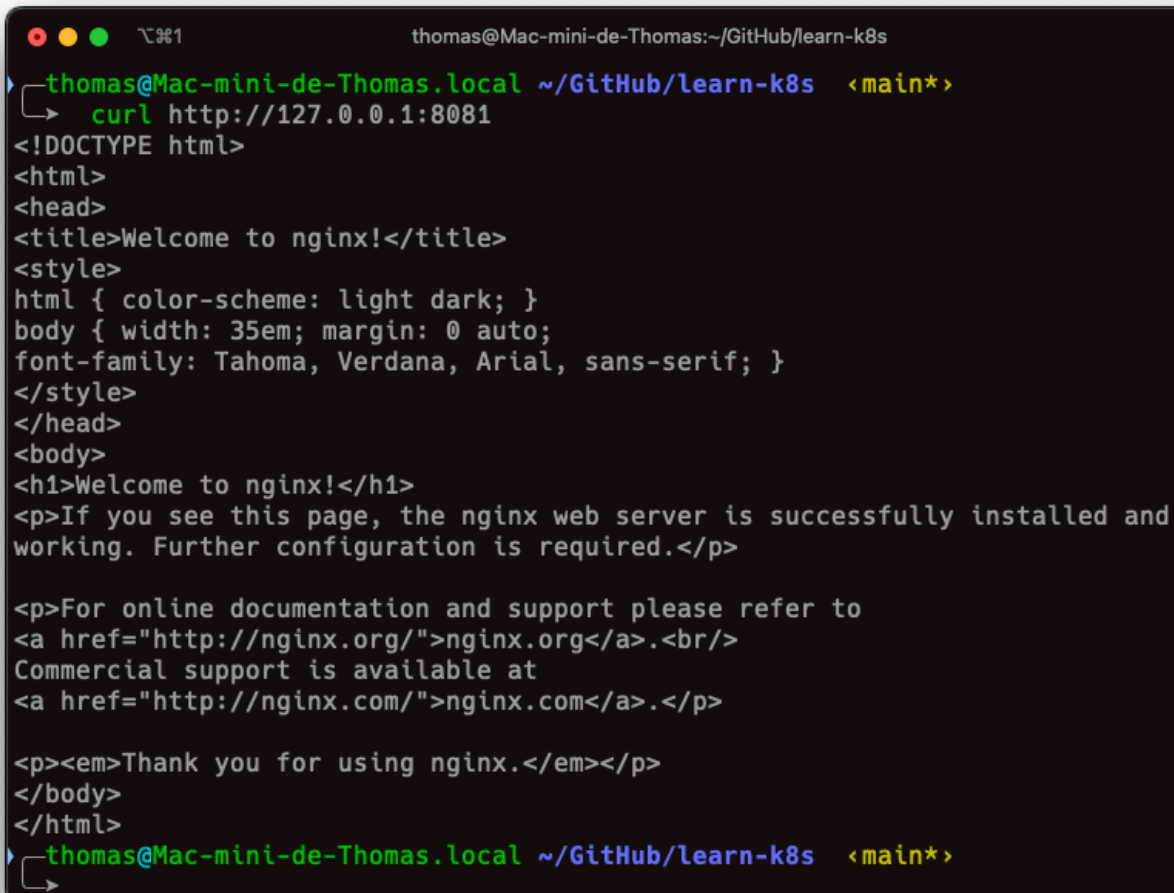


```
minikube tunnel
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
minikube tunnel
✓ Tunnel démarré avec succès

REMARQUE : veuillez ne pas fermer ce terminal car ce processus doit rester ac
tif pour que le tunnel soit accessible...

Tunnel de démarrage pour le service nginx.
```

Puis, nous exécutons un `curl http://127.0.0.1:8081` et obtenons le résultat suivant :



```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
$ curl http://127.0.0.1:8081
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

## Ingress

Nous ajoutons le support pour ingress avec la commande suivante:

```
$ minikube addons enable ingress
```

Puis, nous pouvons constater que notre namespace est bien créé avec nos différents objets grâce à la commande suivante:

```
$ k get all -n ingress-nginx
```

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k get all -n ingress-nginx
NAME                                READY    STATUS    RESTARTS   AGE
pod/ingress-nginx-admission-create-lfgx9    0/1     Completed 0           116s
pod/ingress-nginx-admission-patch-6whtr     0/1     Completed 1           116s
pod/ingress-nginx-controller-7c6974c4d8-8r8g4 1/1     Running   0           116s

NAME                                TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)                                AGE
service/ingress-nginx-controller      NodePort      10.108.137.92 <none>         80:30592/TCP,443:32037/TCP           116s
service/ingress-nginx-controller-admission ClusterIP      10.110.224.36 <none>         443/TCP                               116s

NAME                                READY    UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ingress-nginx-controller 1/1      1             1           116s

NAME                                DESIRED    CURRENT    READY    AGE
replicaset.apps/ingress-nginx-controller-7c6974c4d8 1          1          1        116s

NAME                                COMPLETIONS   DURATION   AGE
job.batch/ingress-nginx-admission-create    1/1           8s         116s
job.batch/ingress-nginx-admission-patch     1/1           8s         116s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Puis, nous créons un fichier `ingress.yaml` ayant le contenu suivant:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
    - host: nginx.info
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: nginx
                port:
                  number: 8081
```

Puis, après avoir `k apply -f ingress.yaml`, nous pouvons vérifier notre ingress s'est bien lancée:

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s — 74X6
k get ingress
NAME    CLASS    HOSTS          ADDRESS          PORTS    AGE
nginx   nginx    nginx.info     192.168.49.2     80       28m
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Nous ajoutons donc une entrée dans notre `/etc/hosts` :

192.168.49.2

nginx.info

Après de multiples tentatives, nous ne parvenons pas à obtenir un `curl` satisfaisant (aucune réponse du serveur) sur l'url `http://nginx.info`. Il semblerait que le problème vienne de `minikube`, qui n'ajoute pas la route sur la machine hôte.

## Ressources Quota

Nous ajoutons un fichier `resource_quota.yaml` ayant le contenu suivant

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resource-quota
spec:
  hard:
    limits.memory: "2Gi"
    limits.cpu: "2"
```

Nous l'appliquons avec `k apply -f resource_quota.yaml`.

Puis nous modifions notre `deployment.yaml` pour avoir le contenu suivant:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
          resources:
            limits:
              memory: 512Mi
              cpu: "0.75"
            requests:
              memory: 256Mi
              cpu: "0.2"
```

Que nous appliquons avec `k apply -f deployment.yaml`.

Nous pouvons constater que notre resource quota s'est bien déployée:

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
k describe ns/default
Name:         default
Labels:       kubernetes.io/metadata.name=default
Annotations:  <none>
Status:       Active

Resource Quotas
Name:         my-resource-quota
Resource      Used    Hard
-----
limits.cpu    1500m  2
limits.memory 1Gi    2Gi

No LimitRange resource.
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

Puis, nous appliquons notre LimitRange qui a le contenu suivant:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
  - default:
      cpu: 500m
    defaultRequest:
      cpu: 500m
    max:
      cpu: 1
    min:
      cpu: 100m
    type: Container
```

Puis `k apply -f limit_range.yaml`.

Ensuite, nous récupérons les informations détaillées du namespace avec `k describe ns/tp2`.

[BrokenFileError: image-20240517061500789: file not found]

Nous pouvons bien voir la limit range bien en place.

Nous créons maintenant un Pod sans limite de ressources dans `pod_new.yaml` :



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx

spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Nous tapons ensuite la commande `k describe pod/nginx` et pouvons constater ceci au niveau des limites en CPU.

[BrokenFileError: image-20240517063000456: file not found]

Nous finissons par supprimer le pod et le limit range avec les commandes suivantes:

```
$ k delete pod/nginx
```

```
$ k delete l
```

## HPA

Nous créons un nouveau fichier `hpadeployment.yaml` ayant le contenu suivant:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: registry.k8s.io/hpa-example
        ports:
        - containerPort: 80
      resources:
        limits:
          cpu: 500m
        requests:
```

```
    cpu: 200m
  apiVersion: v1
  kind: Service
  metadata:
    name: php-apache
    labels:
      run: php-apache
  spec:
    ports:
      - port: 80
    selector:
      run: php-apache
```

Puis, nous faisons `k apply -f hpadeployment.yaml`.

Ensuite, nous créons notre autoscale avec la commande suivante:

```
$ k autoscale hpadeployment php-apache --cpu-percent=50 --min=1 --max=10
```

Puis, nous augmentons la charge dans un autre terminal avec la commande suivante:

```
$ k run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c
"while sleep 0.01; do wget -q -O - http://php-apache; done; k run -i --tty load-generator --rm
--image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O -
http://php-apache; done"
```

Pour suivre l'hpa nous utilisons `k get hpa php-apache -watch`.

```
[BrokenFileError: image-20240517064500123: file not found]
```

Nous pouvons en effet constater que, au bout de quelques instants, nous avons une augmentation du nombre de replicas créés.

```
[BrokenFileError: image-20240517070000890: file not found]
```

Nous arrêtons de générer la charge en killant notre autre terminal, et pouvons en effet constater que le nombre de réplicas a bien réduit.

```
[BrokenFileError: image-20240517073000567: file not found]
```

## CronJob

Nous créons un fichier `cron.yaml` ayant le contenu suivant:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello

spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
```

```

template:
  spec:
    containers:
      - name: hello
        image: busybox
        imagePullPolicy: IfNotPresent
        command:
          - /bin/sh
          - -c
          - date; echo Hello from the Kubernetes cluster
        restartPolicy: OnFailure

```

Nous pouvons voir la liste des cronjobs à l'aide de la commande `k get cj`, ainsi que la liste des jobs avec `k get jobs`.

[BrokenFileError: image-20240517080000134: file not found]

Nous regardons les logs de notre CronJob à l'aide la commande `k logs cronjobs.batch/hello` et observons le résultat suivant:

[BrokenFileError: image-20240517081500912: file not found]

## Storage

Nous créons un volume d'1Go persistant avec le fichier `volume.yaml` ayant le contenu suivant:

```

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-pvc-claim

spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi

```

Nous exécutons ensuite les commandes `k get pvc` et `k get pv` et obtenons le résultat suivant:

[BrokenFileError: image-20240517082500789: file not found]

Nous créons maintenant un nouveau Pod, dans un fichier `storage_pod.yaml` ayant le contenu suivant:

```

kind: Pod
apiVersion: v1
metadata:
  name: task-my-pod
spec:
  volumes:
    - name: my-pv-claim
      persistentVolumeClaim:
        claimName: my-pv-claim

```

```
containers:
  - name: nginx
    image: nginx
    ports:
      - containerPort: 80
    volumeMounts:
      - mountPath: "/usr/share/nginx/html"
        name: my-pv-claim
```

Nous appliquons le pod avec un `k apply -f storage_pod.yaml` puis nous rentrons dans le pod avec la commande suivante:

```
$ k exec pod/task-pv-pod -it -- bash
```

Nous nous plaçons dans `usr/share/nginx/html`.

Le retour de notre `curl http://127.0.0.1/test.html` est le suivant:

[BrokenFileError: image-20240517082751023: file not found]

Puis, nous supprimons notre pod avec `k delete pod task-pv-pod`, nous le recréons avec `k apply -f storage_pod.yaml`, nous remplaçons dans le conteneur et re-effectuons notre `curl`.

[BrokenFileError: image-2024051708371351: file not found]

## Control Plane

- Question sur les composants du control plane
  - Nous obtenons les composants à l'aide de la commande suivante:
 

```
kubectl get pods -n kube-system
```

 Les composants visibles sont déployés comme des pods.

```
thomas@Mac-mini-de-Thomas:~ — 74X13
thomas@Mac-mini-de-Thomas.local ~
➤ kubectl get pods -n kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-5dd5756b68-22tb8	1/1	Running	3 (3m35s ago)	28d
etcd-minikube	1/1	Running	3 (3m35s ago)	28d
kube-apiserver-minikube	1/1	Running	3 (3m35s ago)	28d
kube-controller-manager-minikube	1/1	Running	3 (3m35s ago)	28d
kube-proxy-dbrdn	1/1	Running	3 (3m35s ago)	28d
kube-scheduler-minikube	1/1	Running	3 (3m35s ago)	28d
metrics-server-7c66d45ddc-4ldtx	1/1	Running	7 (2m31s ago)	28d
storage-provisioner	1/1	Running	8 (2m30s ago)	28d

```
➤ thomas@Mac-mini-de-Thomas.local ~
```

Nous créons un serviceaccount à l'aide de la commande `kubectl create serviceaccount my-service-account -n tp2`.

Nous créons un `role.yaml` ayant le contenu suivant:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: my-role
  namespace: tp2
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["services"]
  verbs: ["get", "list", "create", "update"]
- apiGroups: [""]
  resources: ["deployments"]
  verbs: ["get", "list", "create", "update"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "list"]

```

Nous l'appliquons à l'aide de la commande `kubectl apply -f role.yaml`.

Puis, nous créons un `role_binding.yaml` ayant le contenu suivant:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: my-role-binding
  namespace: tp2
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: tp2 # default was previously here
roleRef:
  kind: Role
  name: my-role
  apiGroup: rbac.authorization.k8s.io

```

Nous appliquons ce role binding à l'aide de la commande `kubectl apply -f role-binding.yaml`.

## Questions

- **Vérification des droits du ServiceAccount :**

```
$ kubectl auth can-i get pods --as=system:serviceaccount:tp2:my-service-account -n tp2
```

Nous pouvons constater que tout est configuré correctement.

```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
kubect1 auth can-i get pods --as=system:serviceaccount:tp2:my-service-account -n tp2
yes
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

- Nous lançons la commande suivante:

```
$ kubect1 auth can-i get pods --as=system:serviceaccount:tp2:my-service-account -n default
```

Nous avons `no` comme réponse, ce qui est parfaitement logique: Le RoleBinding n'accorde les droits que dans le namespace `tp2`, et non dans le `default`.

```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
kubect1 auth can-i get pods --as=system:serviceaccount:tp2:my-service-account -n default
no
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

- Nous lançons la commande suivante:

```
$ kubect1 auth can-i get svc --as=system:serviceaccount:tp2:my-service-account -n tp2
```

Nous avons `yes` en réponse, car le serviceAccount a bien les droits pour les services dans le namespace `tp2`.

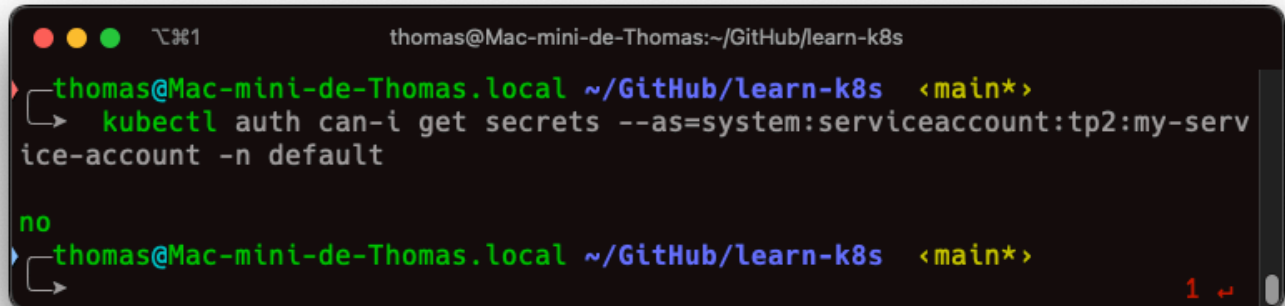
```
thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
kubect1 auth can-i get svc --as=system:serviceaccount:tp2:my-service-account -n tp2
yes
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```

- Nous lançons la commande suivante:



```
$ kubectl auth can-i get secrets --as=system:serviceaccount:tp2:my-service-account -n default
```

Nous avons bien `no` comme réponse, car le ServiceAccount n'a pas de droits sur les secrets dans le namespace `default`, comme mentionné précédemment.

A terminal window with a dark background. The title bar shows three colored circles (red, yellow, green) and the text "thomas@Mac-mini-de-Thomas:~/GitHub/learn-k8s". The prompt is "thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main\*>". The user enters the command "kubectl auth can-i get secrets --as=system:serviceaccount:tp2:my-service-account -n default". The output is "no". The prompt returns. A red "1" and a red arrow icon are visible in the bottom right corner of the terminal window.

```
thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>  
└─▶ kubectl auth can-i get secrets --as=system:serviceaccount:tp2:my-service-account -n default  
  
no  
└─▶ thomas@Mac-mini-de-Thomas.local ~/GitHub/learn-k8s <main*>
```