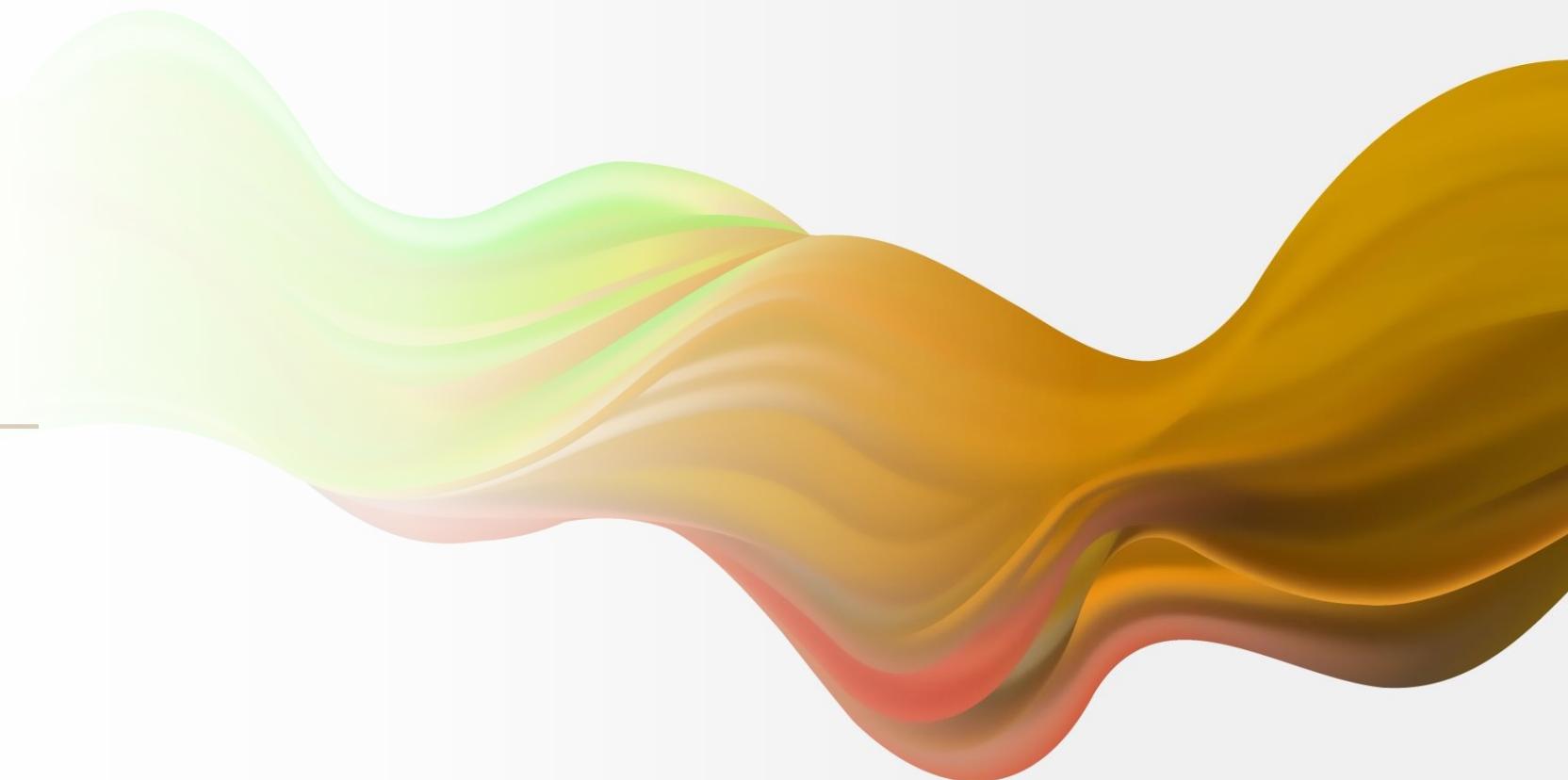




Docker & Kubernetes

Elie Bismuth



Présentation

Elie Bismuth, Développeur Web Full Stack

<https://www.linkedin.com/in/elie-bismuth/>

contact@elie-bismuth.com

Tour de table

Nom, prénom, âge

Formations, expériences, futur job

Connaissance Docker & Kubernetes



Objectif du cours

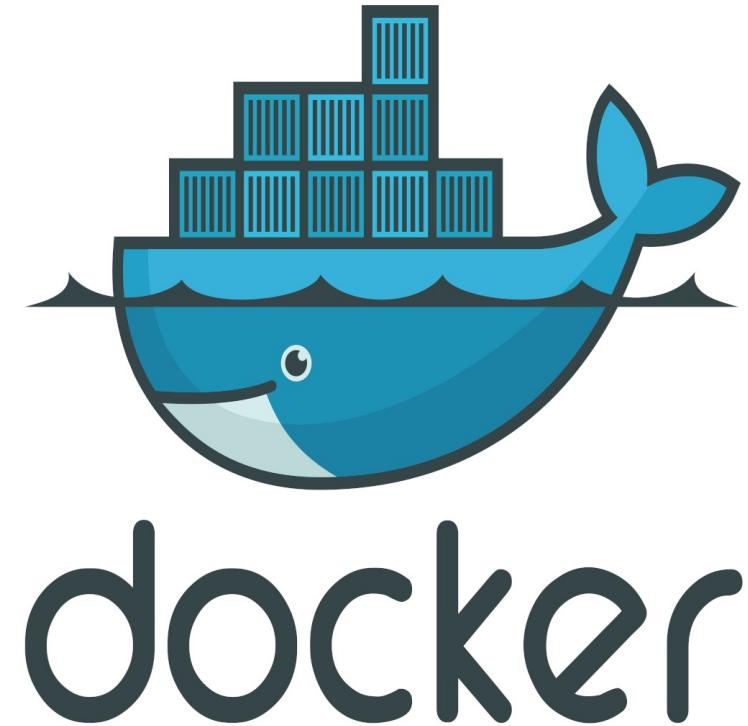
Comprendre ce qu'est Docker et comment l'utiliser

Gérer une application multi containers avec docker-compose

Comprendre ce qu'est Kubernetes et comment l'utiliser



Docker

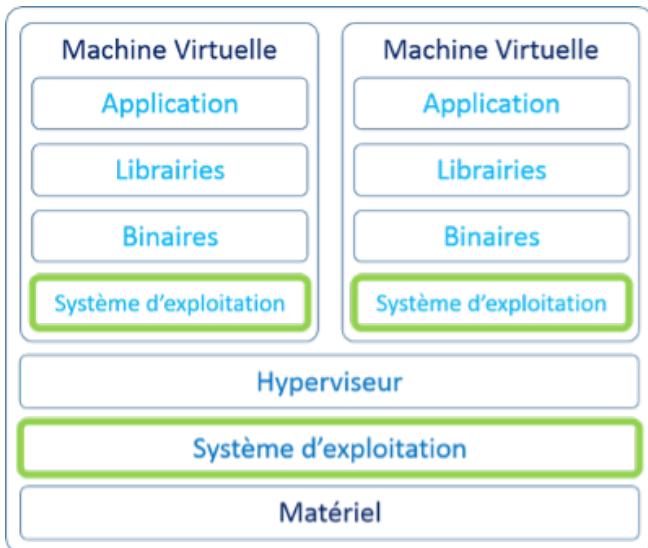


Qu'est-ce que Docker

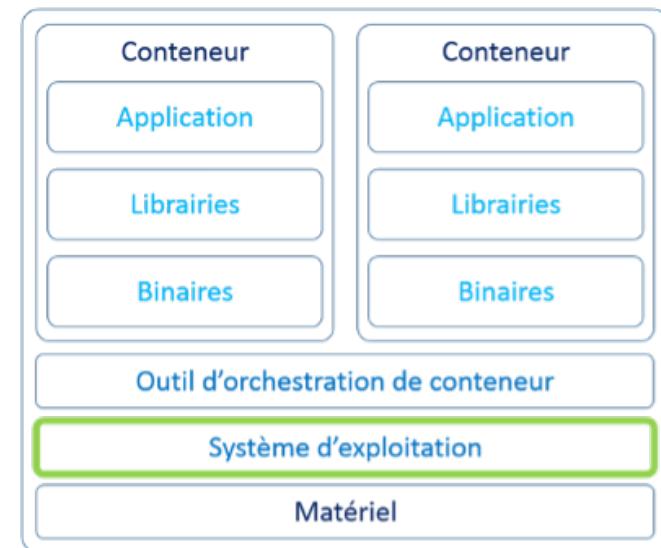
Docker est une technologie de **conteneurisation** qui facilite la gestion de dépendance au sein d'un projet et ce, à tous les niveaux (développement et déploiement).

Disponible sur Linux, Windows et Mac OS, le mécanisme de Docker se centre autour des **conteneurs** et de leur orchestration, et c'est en cela que la **conteneurisation** se différencie de la **virtualisation**.

Virtualisation :



Conteneurisation :



Pourquoi Docker

Un problème fréquent lors de l'installation d'un projet est de passer plusieurs heures à essayer d'installer et de lancer un projet car beaucoup de facteurs problématiques peuvent entrer en jeu, tels que le système d'exploitation, la base de données, ou encore la version du langage de développement.

Avec Docker, tous ces problèmes de dépendances de versions et de machines n'existent plus



Concepts clés

Il existe trois concepts clés dans Docker : **les conteneurs, les images et les fichiers Docker (Dockerfile)**

- Un **conteneur** est un espace dans lequel une application tourne avec son propre environnement. Les applications qu'un conteneur peut faire tourner sont de tous types : site web, API, db, etc. Chaque conteneur est une instance d'une image.
- Les **images** représentent le contexte que plusieurs conteneurs peuvent exécuter. Elles sont aux conteneurs ce que les classes sont aux objets en Programmation Orientée Objet : un moule.
- Un **Dockerfile** est un fichier qui liste les instructions à exécuter pour build une image. Il est lu de haut en bas au cours du processus de build.

Nous verrons tous cela en détails dans le cours

Docker Desktop

Pour utiliser Docker sur notre pc, nous allons avoir besoin d'installer [Docker Desktop](#)

Docker Desktop est un logiciel destiné aux développeurs, qui propose un ensemble d'outils améliorant la productivité de l'utilisateur, ainsi qu'un environnement Kubernetes local

Docker Desktop est composé entre autres de deux éléments importants:

1. **Docker Client (Docker CLI)**, outil avec le quelle nous allons interagir en ligne de commande
2. **Docker Server (Docker Daemon)**, l'outil responsable de créer les images, lancer les containers, etc

Installation

Suivre les instructions du TP0 sur [ce lien](#) pour l'installation de Docker

Utiliser le Docker Client

```
→ ~ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:2498fce14358aa50ead0cc6c19990fc6ff866ce72aeb5546e1d59caac3d0d60f
Status: Downloaded newer image for hello-world:latest

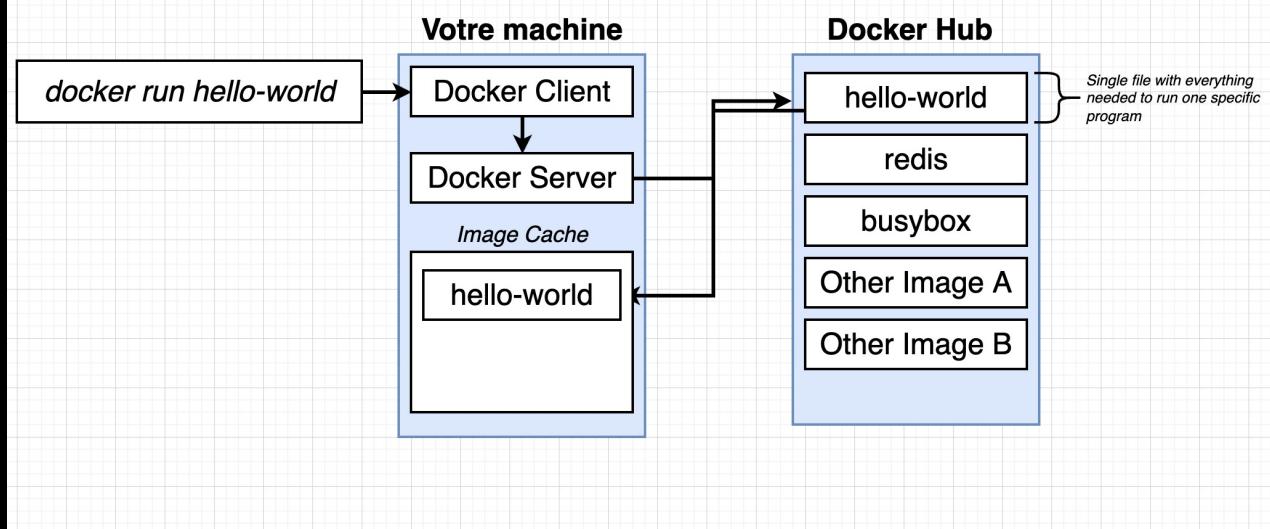
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

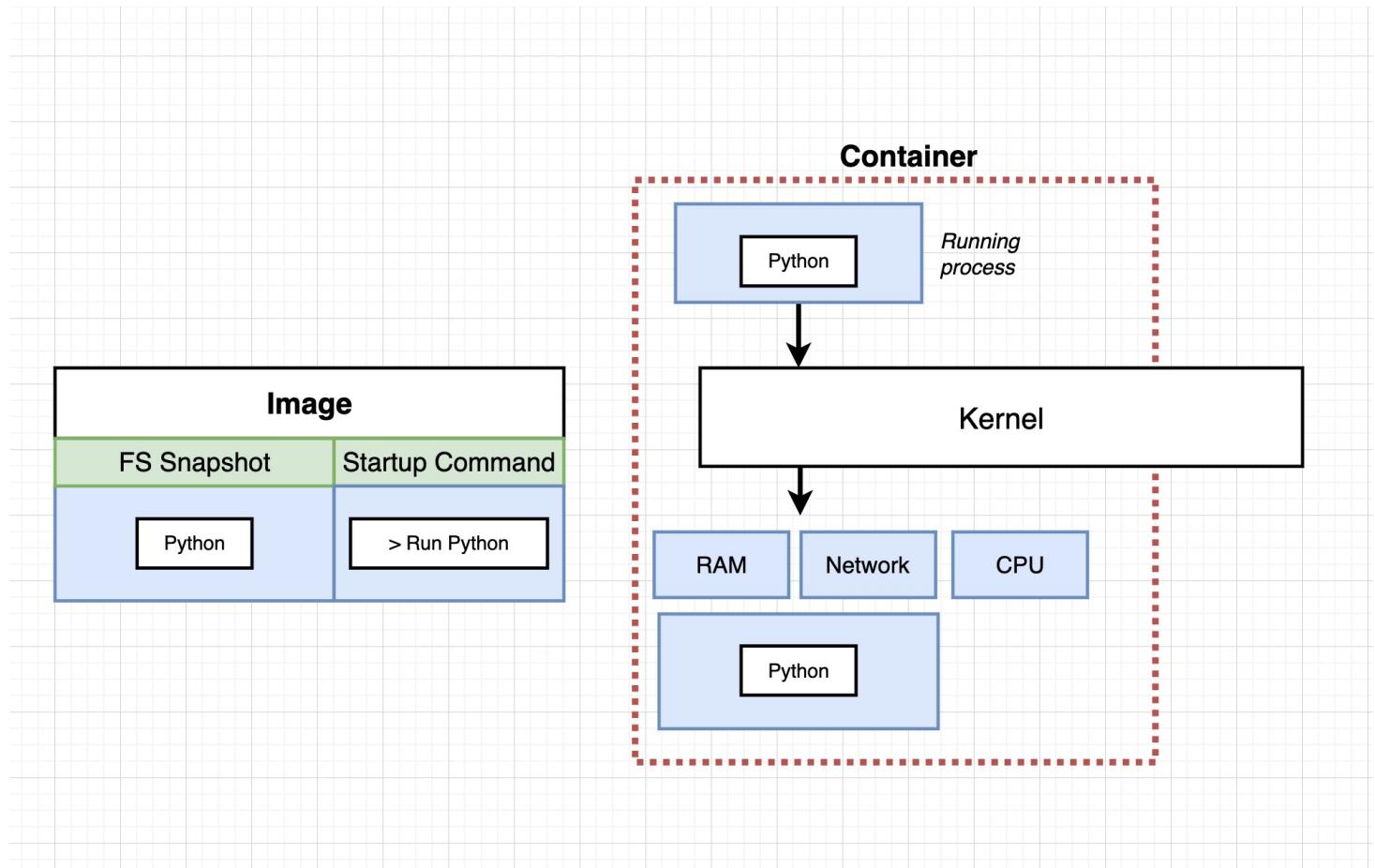
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```



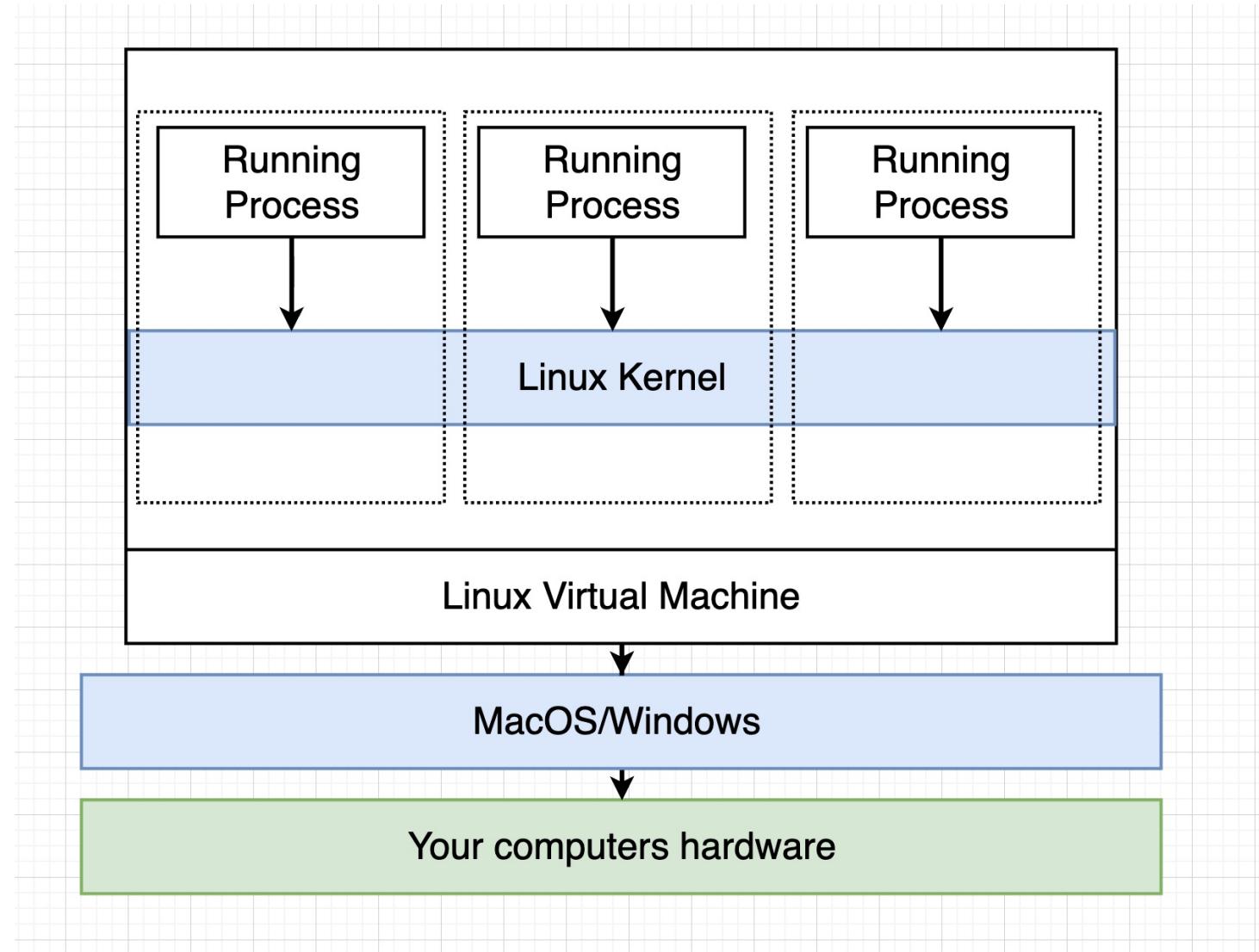
Container

- Une image contient une copie des fichiers de base de notre machine, ainsi qu'une commande
- Lorsqu'on crée un container avec une image, le Kernel sélectionne une partie du disque dur de notre machine pour l'assigner à ce container uniquement
- Ainsi, lorsque le processus du container est en cours, le Kernel utilise uniquement les ressources assignées au container, contenant dans notre cas le langage Python et rien d'autre



Comment Docker fonctionne

- Le fait de pouvoir regrouper des ressources par namespace n'est possible qu'avec Linux
- En réalité, lorsqu'on installe le Docker Client sur Mac/Windows, on installe une machine virtuelle Linux, qui communique avec notre OS pour utiliser les ressources de notre machine





Manipuler des containers et des images

TP1

Durant ce TP, vous allez apprendre et tester les commandes docker utilisées pour manipuler des containers et des images.

Le TP est disponible sur [ce lien](#)



Créer des images

Comment créer ses propres images

Jusqu'à maintenant, nous avons utilisé des images créées par d'autres personnes.

Nous allons maintenant voir comment créer nos propres images Docker.

Pour créer une image Docker, il faut créer un fichier intitulé **Dockerfile**.

Ce fichier définit la configuration et le comportement des containers qui seront créées à partir de cette image

Ensuite, il faut utiliser le Docker Client pour lancer cette image, qui va être envoyée au Docker Server qui va rendre l'image utilisable.

En général, la structure du **Dockerfile** reste la même et suit les étapes suivantes:

1. Spécifier une image de base
2. Lancer des commandes pour installer certains programmes ou effectuer certaines actions
3. Spécifier une commande à exécuter lors de la création du container

Dockerfile

Pour créer un container à partir d'un Dockerfile, il faut lancer la commande:

docker build .

L'argument après l'instruction build signifie: dans quel dossier se trouve le Dockerfile (le . signifie le répertoire courant)

Lorsque Docker crée un container à partir d'une image, il met en cache les étapes de création. Modifier l'ordre de ces étapes à un impact sur le cache lors d'un nouveau build

```
# Utiliser une image existante comme image de base
FROM php:7.4-cli

# Télécharger et installer des dépendances
RUN docker-php-ext-install pdo pdo_mysql
RUN pecl install apcu
RUN docker-php-ext-enable apcu

# Copie les fichiers du répertoire courant dans le dossier /usr/src/myapp du
# container
COPY . /usr/src/myapp

# Défini le répertoire de travail des instructions RUN, CMD, ENTRYPOINT, COPY et
# ADD qui le suivent dans le Dockerfile
WORKDIR /usr/src/myapp

# Dire à l'image quelle commande lancer au démarrage du container
CMD ["php", "./index.php"]
```

Image de base

L'image de base est importante: elle définit à quoi ressemblera notre container avant que l'on effectue nos propres actions.

Dans notre exemple, l'image spécifiée étant une image PHP, le container aura php installé et configuré. Nous pouvons ensuite télécharger les dépendances que l'on souhaite, lancer des commandes, etc

Il existe des images de bases pour chaque langage ou technologie (Python, Redis, Node, MySQL...)

Commandes Dockerfile

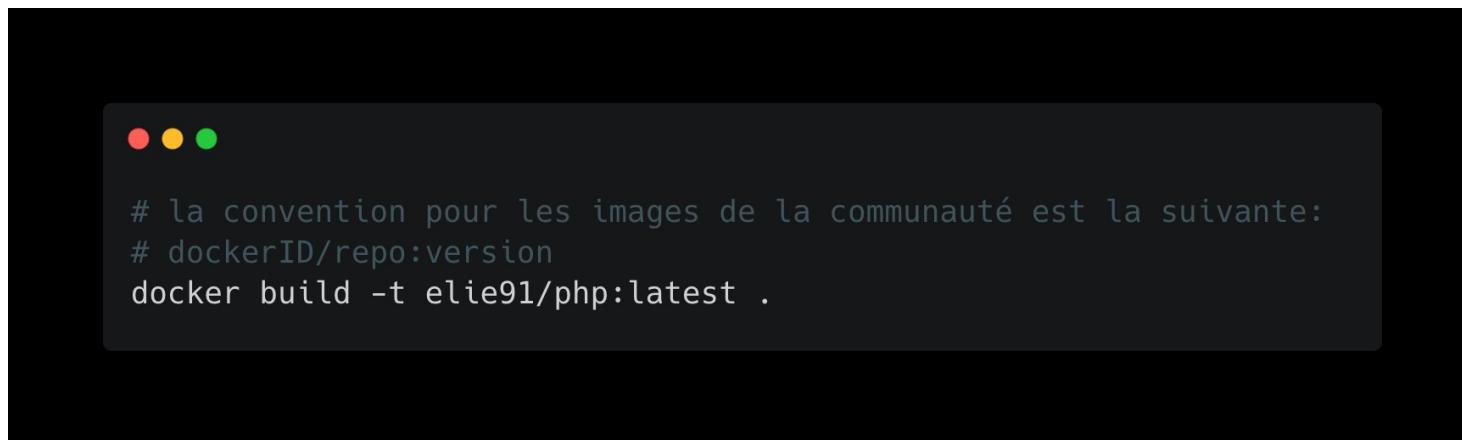
Voici la liste des principales commandes pour un Dockerfile:

- **FROM**: Définit l'image de base pour les instructions suivantes.
- **RUN**: Exécute une commande dans une nouvelle couche au-dessus de l'image actuelle et validera les résultats. L'image validée résultante sera utilisée pour la prochaine étape dans le Dockerfile. Un Dockerfile peut contenir plusieurs instructions RUN
- **CMD**: Fournit une commande par défaut au lancement du container. Il ne peut y avoir qu'une seule instruction CMD dans un Dockerfile
- **EXPOSE**: Informe Docker que le conteneur écoute sur les ports réseaux spécifiés lors de l'exécution
- **ENV**: Définit une variable d'environnement
- **COPY**: Copie les nouveaux fichiers ou répertoires de la source vers la destination

Tagger une image

Actuellement, lorsque l'on crée notre image à partir du Dockerfile, il faut récupérer l'id de l'image manuellement pour lancer le container
Cela signifie aussi que plusieurs images sont créées à chaque build avec des id différents

On peut tagger une image, pour utiliser le tag de l'image au lieu de son ID, comme nous l'avons fait avec les images externes (docker run redis)
Pour tagger une image, on peut ajouter l'option **-t** à la commande docker build:

A screenshot of a terminal window with a dark background. In the top left corner, there are three small colored dots: red, yellow, and green. The terminal displays the following text:

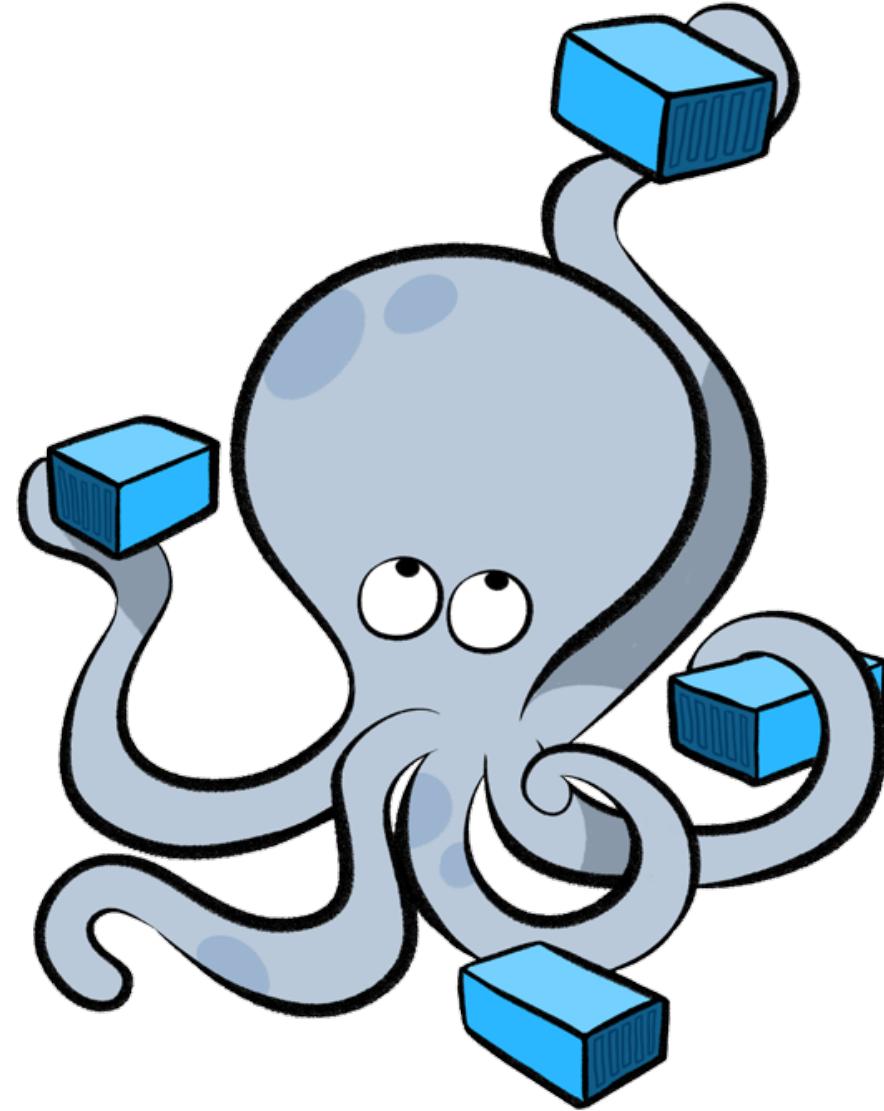
```
# la convention pour les images de la communauté est la suivante:  
# dockerID/repo:version  
docker build -t elie91/php:latest .
```

TP2

Dans ce TP, vous utiliserez Docker pour créer une image permettant de lancer une application NodeJS

Vous trouverez le TP sur [ce lien](#)

Docker Compose



Qu'est ce que Docker Compose

- Docker Compose est un outil en ligne de commande qui est installé avec Docker
- Il est utilisé pour gérer et démarrer plusieurs containers docker en même temps
- Docker compose automatise certaines options que l'on doit passer manuellement à la commande docker run, notamment pour les ports, les variables d'environnement, et le networking (communication entre les containers)

docker-compose.yml

```
# Spécifie la version du docker-compose
version: "3.6"

# Liste des services docker qui composent notre app
services:

    # Définit le nom du service
    postgres:
        # Définit l'image à utiliser
        image: postgres:12-alpine
        # Définit le mapping de ports
        ports:
            - 7082:5432
        # Spécifie les variables d'environnement
        environment:
            - PGDATA=/data/postgres
            - POSTGRES_DB=star_wars
            - POSTGRES_USER=star_wars_user
            - POSTGRES_PASSWORD=star_wars_password

    # Définit le nom du service
    back:
        # Spécifie à quel endroit se trouve le Dockerfile pour build l'image
        build: "back"
        # Spécifie le répertoire de travail par défaut
        working_dir: "/app"
        # Définit le mapping de ports
        ports:
            - 4000:4000
        # Exprime une dépendance avec le service postgres
        # Le service postgres se lancera avant le service back
        depends_on:
            - postgres
```

Redémarrage automatique

- Avec Docker Compose, on peut également spécifier le comportement que le container doit adopter dans le cas où il plante à cause d'une erreur ou s'il s'arrête pour une raison ou pour un autre
- Ce comportement est défini grâce à l'option restart
- Cette option peut prendre quatre valeurs:

```
back:  
  build: "back"  
  working_dir: "/app"  
  
  # un seul restart par service  
  
  # n'essaye pas de redémarrer le container s'il s'arrête  
  restart: "no"  
  
  # essaye dans tous les cas de redémarrer le container  
  restart: "always"  
  
  # essaye de redémarrer le container seulement s'il s'arrête  
  # suite à un code erreur  
  restart: "on-failure"  
  
  # essaye dans tous les cas de redémarrer le container sauf  
  # si le container est stoppé manuellement  
  restart: "unless-stopped"  
  
ports:  
  - 4000:4000  
depends_on:  
  - postgres
```



Commandes

```
● ● ●

# build tous les containers spécifiés dans le docker-compose
$ docker-compose build

# run tous les containers spécifiés dans le docker-compose
$ docker-compose up

# build et run
$ docker-compose up --build

# run tous les containers en background
$ docker-compose up -d

# stop tous les containers du docker-compose en cours
$ docker-compose down

# liste les containers du docker-compose en cours
$ docker-compose ps
```

TP3

Dans ce TP, nous allons voir comment utiliser docker-compose pour faire fonctionner une app NodeJS, Postgres, et React

Il s'agit d'un cas que vous rencontrerez souvent dans la vie réelle en tant que développeur

Nous allons également voir comment mettre en place un workflow de développement et un workflow de production avec docker et docker-compose

Le TP est disponible sur [ce lien](#)



Kubernetes



Kubernetes

- D'anciens développeurs de Borg écrivent K8s en Go
- Directement pensé pour utiliser Docker (engine)
- Directement dans l'optique d'en faire un projet OpenSource
- Version 1.0 en Juin 2015

Kubernetes permet d'éliminer de nombreux processus manuels associés au déploiement et à la mise à l'échelle des applications conteneurisées

Kubernetes vous aide à gérer facilement et efficacement des clusters au sein desquels vous aurez rassemblé des groupes d'hôtes exécutant des conteneurs Linux.

Concrètement, Kubernetes aide à optimiser l'utilisation des ressources et les coûts en faisant évoluer automatiquement les ressources en fonction de la demande

Pourquoi Kubernetes

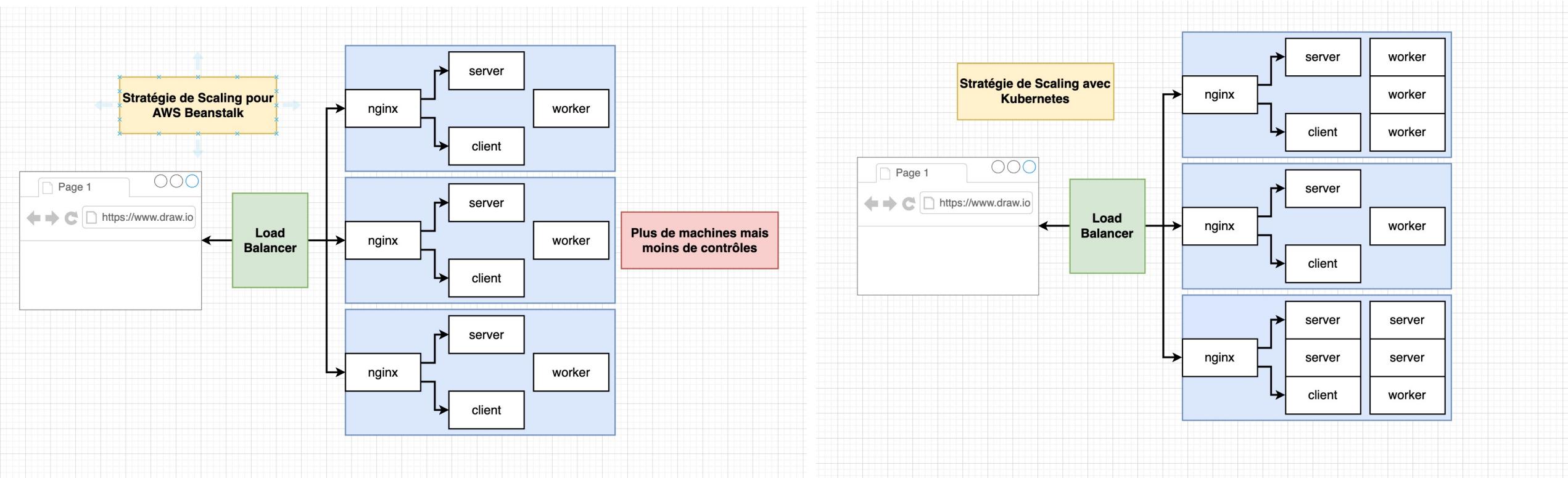
- Définir et déployer des applications multi-conteneurs
- Répartir les conteneurs sur une flotte d'hôtes (nœuds)
- Optimiser et adapter le placement des conteneurs
- Surveiller la santé des conteneurs
- Définir et appliquer des contraintes de niveaux de services
- Gérer la disponibilité et la scalabilité des conteneurs
- Gérer le provisionnement et l'accès au stockage
- Isoler les conteneurs
- Limitation de ressources
- Sécurité (vision multi-tenant)

Tout ça de manière dynamique et pour des milliers de conteneurs !

Développement et production

- Pour utiliser Kubernetes lors de la phase de développement, il faut soit utiliser le programme Minikube qui permet de créer une VM contenant un Cluster, soit utiliser directement la solution Kubernetes intégrée à Docker Desktop
- Minikube ou Kubernetes Docker Desktop installent un programme appelé kubectl.
- Ce programme permet de gérer les containers au sein d'un cluster. Minikube ou Kubernetes Docker Desktop s'occupent juste de gérer la VM en elle-même, et kubectl est le programme principal qui va nous permettre de gérer notre cluster
- Pour déployer un cluster Kubernetes, plusieurs solutions existent, les plus connues étant Elastic Container Service (**EKS**) d'Amazon, et Google Cloud Kubernetes Engine (**GKE**) de Google
- Minikube n'est utilisé qu'en phase de développement, mais le programme kubectl est également utilisé en production par les solutions citées plus haut pour la gestion du cluster

Example de Scaling

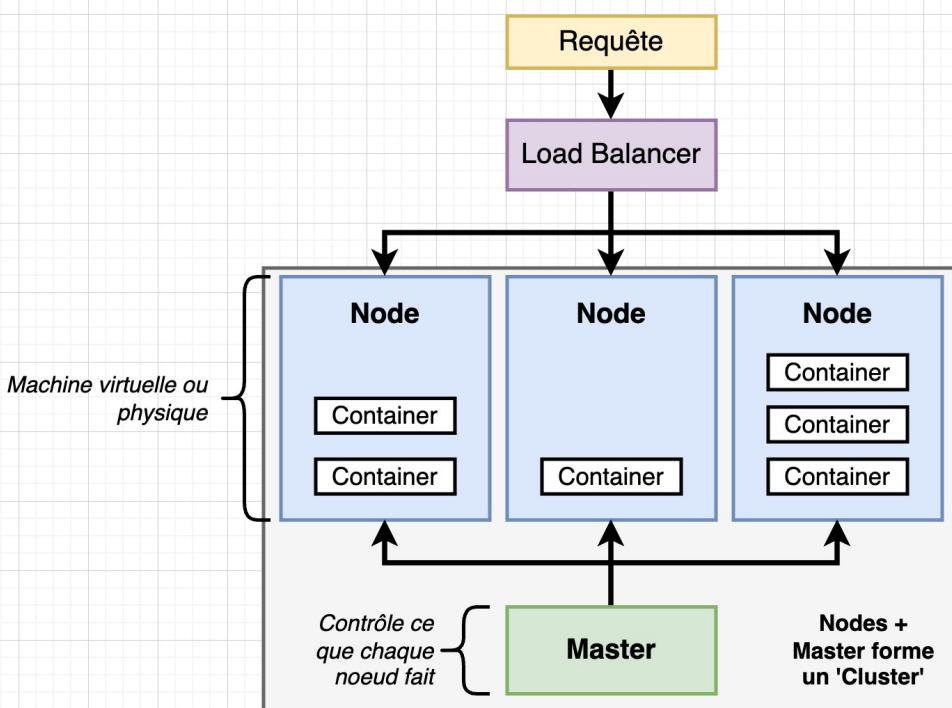


Installation

Voici [le lien du TP4](#)

Durant ce TP, vous allez installer Kubernetes pour créer et utiliser un cluster sur votre machine

Cluster



Important

Kubernetes est un système pour déployer des apps conteneurisées

Les **Nodes** sont des machines individuelles (ou virtuelles) qui exécutent des conteneurs

Les **Masters** sont des machines (ou machines virtuelles) avec un ensemble de programmes pour gérer les noeuds

Kubernetes ne build pas nos images comme le ferait Docker Compose. Il s'attend à les récupérer déjà build d'un endroit comme Docker Hub

Le master décide où exécuter chaque container - chaque noeud peut exécuter un ensemble de containers

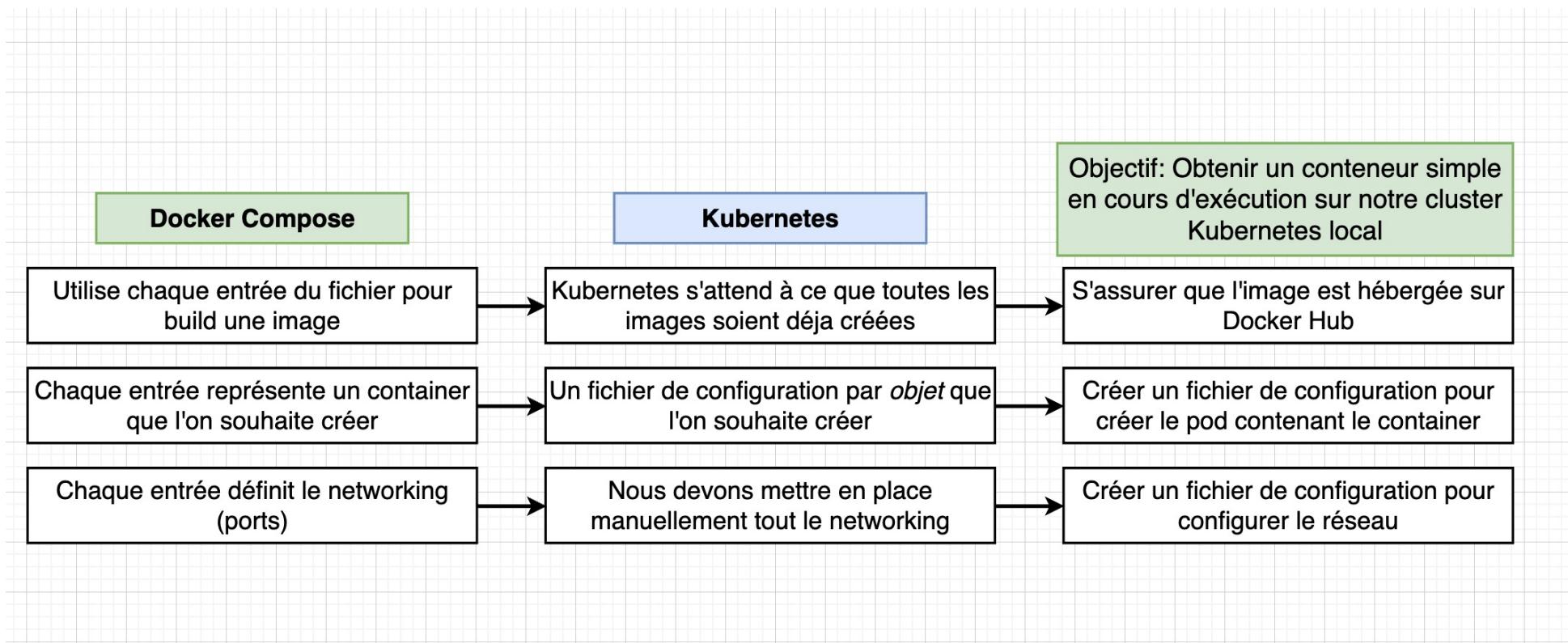
Pour déployer quelque chose, nous mettons à jour l'état souhaité du master avec un fichier de configuration

Le master travaille constamment pour répondre à l'état souhaité

Important

Notre premier cluster

Nous allons voir comment créer notre premier cluster pour utiliser notre image react du projet Star Wars



Notre premier cluster

```
# client-pod.yaml

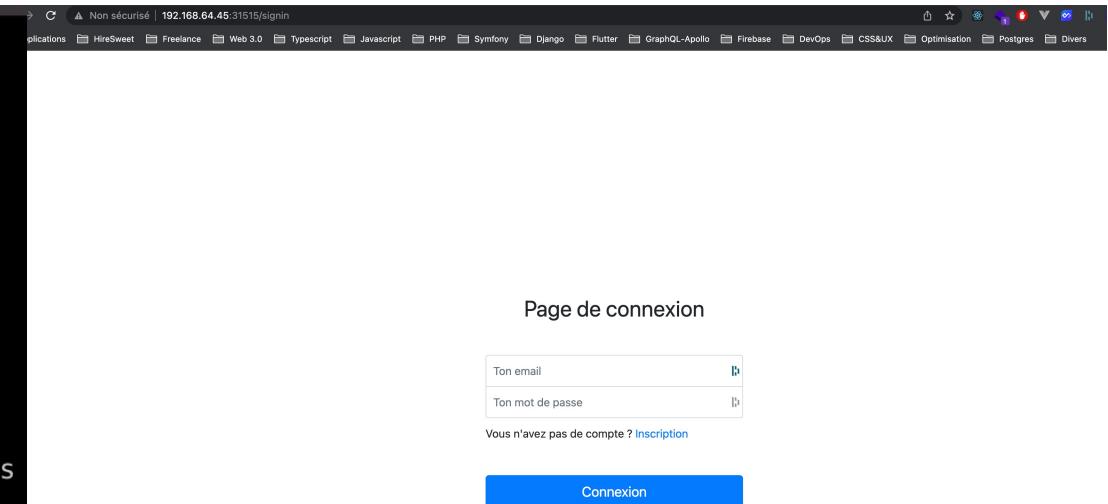
apiVersion: v1
kind: Pod
metadata:
  name: client-pod
  labels:
    component: front
spec:
  containers:
    - name: client
      image: elie91/starwars-front
      ports:
        - containerPort: 3000
```

```
# client-node-port.yaml

apiVersion: v1
kind: Service
metadata:
  name: client-node-port
spec:
  type: NodePort
  ports:
    - port: 3050
      targetPort: 3000
      nodePort: 31515
  selector:
    component: front
```

Notre premier cluster

```
→ 4-Kubernetes-FirstCluster git:(main) ✘ ls
client-node-port.yaml client-pod.yaml
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl apply -f client-pod.yaml
pod/client-pod created
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl apply -f client-node-port.yaml
service/client-node-port created
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
client-pod  1/1     Running   0          27s
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl get services
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
client-node-port  NodePort    10.107.253.222  <none>           3050:31515/TCP  27s
kubernetes       ClusterIP   10.96.0.1      <none>           443/TCP        7m54s
→ 4-Kubernetes-FirstCluster git:(main) ✘ minikube ip
192.168.64.45
→ 4-Kubernetes-FirstCluster git:(main) ✘
```



Explication

Un fichier de configuration Kubernetes est utilisé pour créer des **objets**. Ces objets servent à différentes choses: exécuter un conteneur, surveiller un conteneur, configurer le networking ...

Tous les fichiers contiennent les informations suivantes:

apiVersion: Définit les types d'objets que l'on peut utiliser dans le fichier. Certains objets sont disponibles dans l'apiVersion v1, d'autres dans apps/v1

kind: Définit le type d'objet que l'on souhaite créer: Pod, Service, Volume ...

metadata: Définit de la donnée que l'on passe à l'objet, telle que son nom, son sélecteur ...

spec: Définit plus précisément le type d'objet que l'on souhaite créer, définit les containers dans le cas d'un pod, le networking dans le cas d'un service, etc

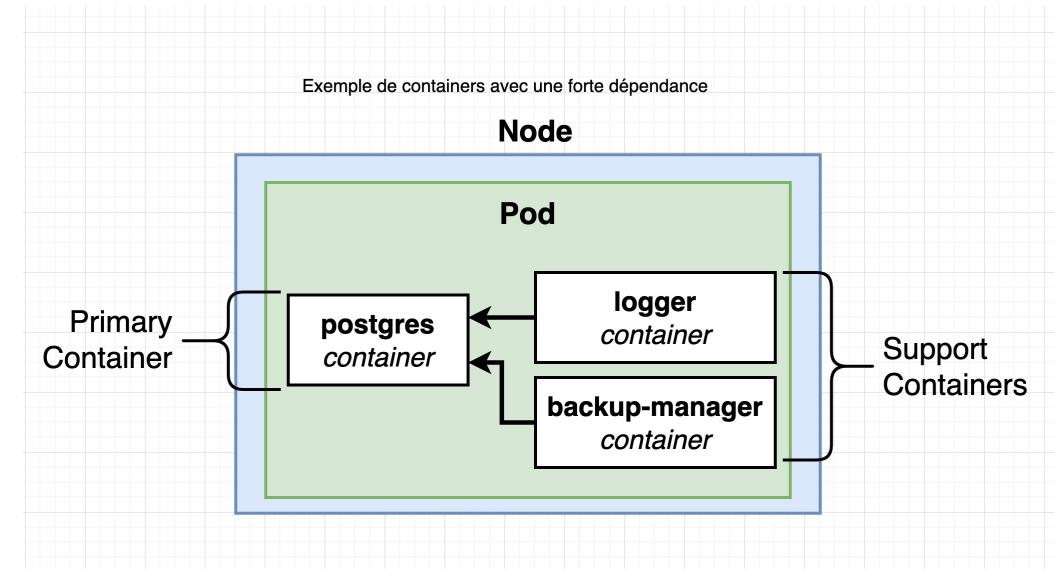
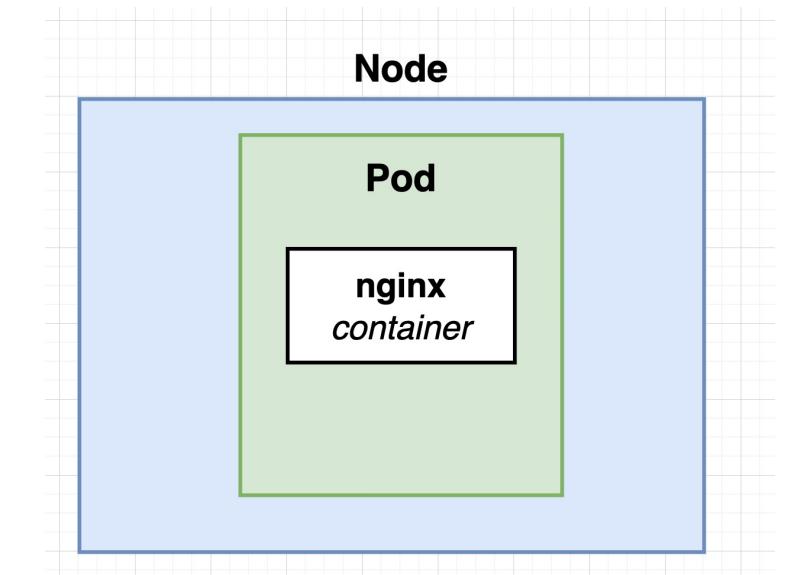
Pod

Un **Pod** est l'objet basique que l'on utilise pour exécuter un ou plusieurs containers. Un **Pod** est contenu dans un **Node** Kubernetes.

On ne peut pas déployer uniquement un container avec Kubernetes : Un container doit toujours être contenu par un **Pod** pour être exécuté.

Le but d'un **Pod** est de permettre ce regroupement de containers avec un objectif très similaire ou des containers qui doivent absolument être déployés ensemble et doivent fonctionner ensemble.

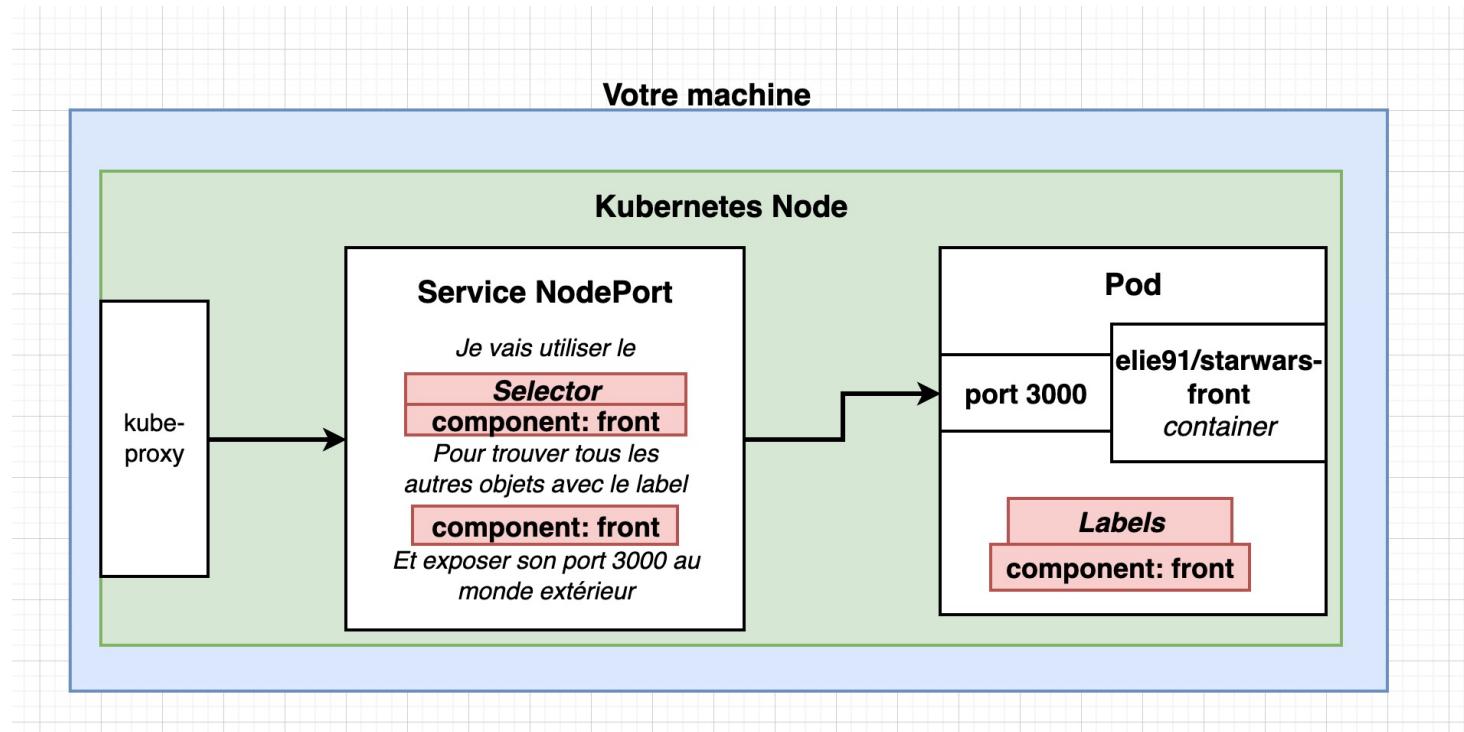
Dans le cas de notre application, nous utiliserons uniquement un container par **Pod** car nous n'avons pas ce besoin de regroupement de containers



Service

Un **service** Kubernetes est utilisé pour mettre en place le networking (mappage réseau) dans un cluster Kubernetes. Il existe plusieurs type de service, les plus utilisées étant:

- **ClusterIP**: Expose un Pod aux autres objets du cluster
- **NodePort**: Expose un Pod au monde extérieur (recommandé uniquement pour la phase de développement !)
- **LoadBalancer**: Ancienne façon de diriger le trafic réseau au sein du Cluster
- **Ingress**: Expose un ensemble de service au monde extérieur



Récapitulatif

```
● ● ●  
# Définit l'apiVersion du fichier  
apiVersion: v1  
  
# Crée un objet de type Pod  
kind: Pod  
  
# Définit les metadata de l'objet  
metadata:  
  name: client-pod  
  labels:  
    component: front  
  
# Définit les specs du Pod  
spec:  
  # Liste des containers que contient notre Pod  
  containers:  
    # Nom, image et port du container  
    # Le port déclaré ici n'est pas suffisant pour que le container soit  
    accessible  
    - name: client  
      image: elie91/starwars-front  
      ports:  
        - containerPort: 3000
```

```
● ● ●  
# Définit l'apiVersion du fichier  
apiVersion: v1  
  
# Crée un objet de type Service  
kind: Service  
  
# Définit les metadata de l'objet  
metadata:  
  name: client-node-port  
  
# Définit les specs du service  
spec:  
  # Précise quel type de service nous souhaitons créer  
  type: NodePort  
  
  # Définit le port mapping  
  ports:  
    # Port accessible par les autres Pod  
    - port: 3050  
      # Port cible  
      targetPort: 3000  
      # Port accessible via le navigateur  
      nodePort: 31515  
  
  # Définit les autres objets que l'on souhaite faire communiquer avec notre  
  # service en matchant le selector défini  
  selector:  
    component: front
```

Impératif vs Déclaratif

Il y'a deux façons de créer et gérer un cluster Kubernetes:

Déclarative: On spécifie via des **fichiers** de configuration à quoi notre Cluster doit ressembler

Impérative: On spécifie via des **commandes** à quoi notre Cluster doit ressembler

Avec l'approche impérative, il faut vérifier manuellement l'état de son cluster, et le mettre à jour manuellement pour atteindre l'état désiré.

Avec l'approche déclarative, nous sommes assurés que l'état du cluster sera toujours similaire à celui décrit et demandé dans les fichiers de configuration. Nous n'avons pas à se soucier de vérifier constamment l'état du cluster pour le mettre à jour manuellement.

Chaque modification sur le fichier de configuration entraînera une mise à jour du cluster pour atteindre l'état spécifié.

L'approche déclarative est donc l'approche recommandée

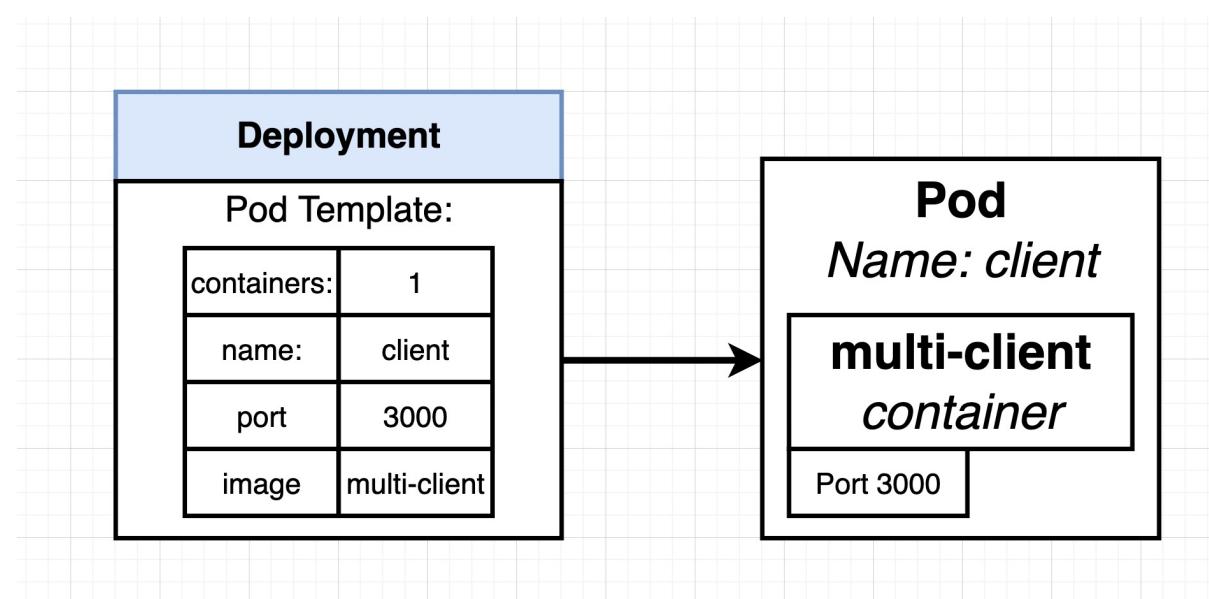
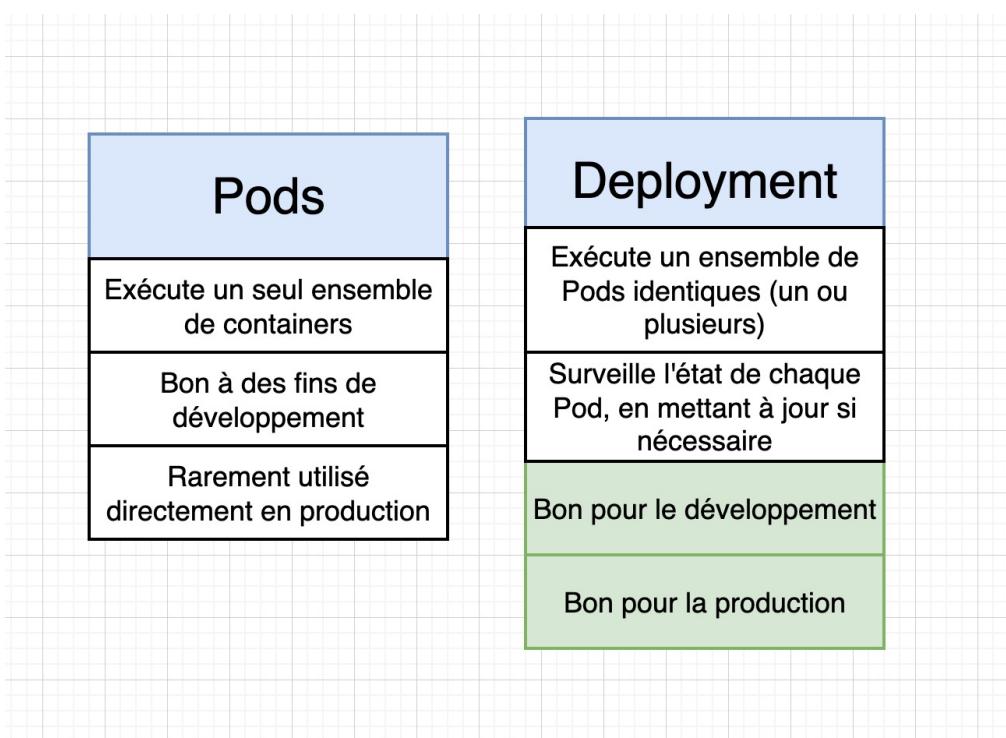
Limitations des Pod

- Créer un objet de type Pod pour gérer un container contient certaines limites.
- En effet, dans le cas où l'on souhaite modifier le fichier de configuration du Pod, certaines propriétés ne peuvent pas être modifiées, telles que le port exposé au sein du container, le nombre de containers ou le nom du Pod
- Il faut donc supprimer manuellement le Pod et en créer un nouveau
- De plus, si l'on souhaite que notre cluster contienne plusieurs Pod de notre container client, il faut pour cela créer autant de fichier de configuration que de Pod attendus

C'est pour cela qu'on utilise plutôt des objets **Deployment** pour gérer des Pod dans un cluster Kubernetes

Deployment

Le rôle d'un **Deployment** est de maintenir un ensemble de Pods identiques, en s'assurant qu'ils aient la bonne configuration et que le bon nombre de Pods soit présent au sein du Cluster





client-deployment.yaml

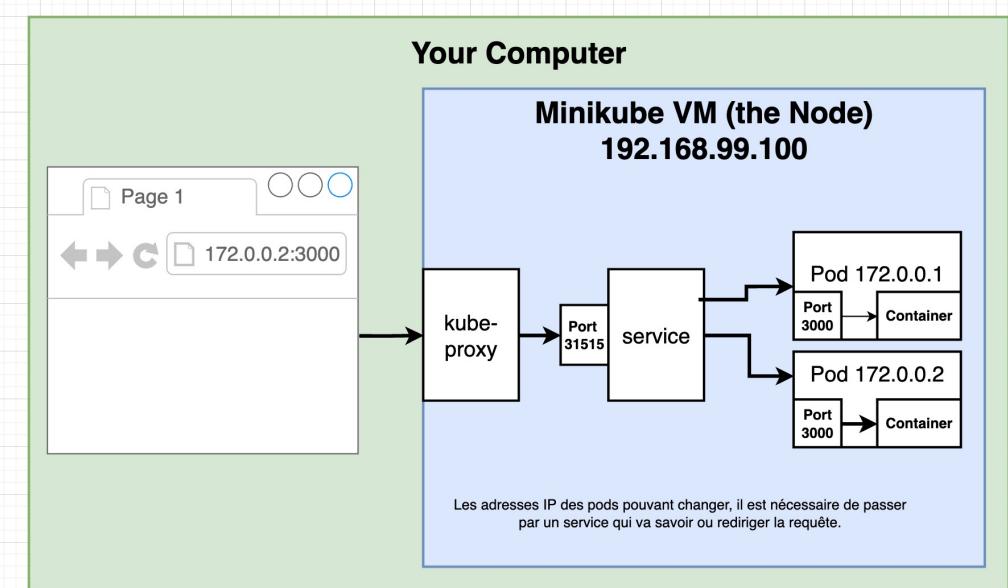
```
● ● ●

apiVersion: apps/v1
kind: Deployment
metadata:
  name: client-deployment
spec:
  # Spécifie le nombre de Pods que ce Deployment va créer et surveiller
  replicas: 2
  selector:
    matchLabels:
      component: front

  # Définit le template de configuration qui sera utilisé pour chacun des pods
  # créé par le Deployment
  template:
    # Définit les metadata pour tous les Pods
    metadata:
      labels:
        component: front
    # Définit les spec pour tous les Pods
    spec:
      containers:
        - name: client
          image: elie91/starwars-front
          ports:
            - containerPort: 3000
```

Appliquer un Deployment

```
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
client-pod 1/1     Running   0          39s
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl delete pod client-pod
pod "client-pod" deleted
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl get pods
No resources found in default namespace.
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl apply -f client-deployment.yaml
deployment.apps/client-deployment created
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
client-deployment-6c4598b7ff-5ppc9 1/1     Running   0          10s
client-deployment-6c4598b7ff-9r26v 1/1     Running   0          10s
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl get deployment
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
client-deployment 2/2     2           2           14s
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl get services
NAME            TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
client-node-port  NodePort   10.98.130.4 <none>       3050:31515/TCP  84s
kubernetes       ClusterIP  10.96.0.1   <none>       443/TCP       2m10s
→ 4-Kubernetes-FirstCluster git:(main) ✘ minikube ip
192.168.64.46
→ 4-Kubernetes-FirstCluster git:(main) ✘ kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP          NODE   NOMINATED-NODE   READINESS   GATES
client-deployment-6c4598b7ff-5ppc9 1/1     Running   0      2m53s   172.17.0.4   minikube   <none>        <none>
client-deployment-6c4598b7ff-9r26v 1/1     Running   0      2m53s   172.17.0.3   minikube   <none>        <none>
```



Reconfigurer Docker

- Pour les personnes utilisant Minikube, si vous ouvrez votre terminal et lancez la commande **`docker ps`**, vous ne verrez pas les containers qui sont au sein du cluster Kubernetes
 - Cela est dû au fait que la virtuelle machine créée par Minikube possède sa propre version du Docker Client et du Docker Daemon
 - Pour résoudre cela, on peut reconfigurer temporairement le terminal pour lui dire d'utiliser la version de Docker sur notre machine au lieu de celle au sein de la VM
 - La commande exécuter est la suivante : **`eval $(minikube docker-env)`**
 - L'intérêt à cela est de pouvoir utiliser les commandes Docker classiques pour débugger nos containers, bien que la plupart de ces commandes sont aussi disponibles avec kubectl

```

→ ~ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
→ ~ kubectl get pods
NAME READY STATUS RESTARTS AGE
client-deployment-6c4598b7ff-5ppc9 1/1 Running 0 12h
client-deployment-6c4598b7ff-9r26v 1/1 Running 0 12h
→ ~ eval $(minikube docker-env)
→ ~ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
0991338f1a46 6e38f40d628d "/storage-provisioner" 10 hours ago Up 10 hours
1c062e44b674 elie91/starwars-front "/docker-entrypoint..." 13 hours ago Up 13 hours
c30b58b2e5f5 elie91/starwars-front "/docker-entrypoint..." 13 hours ago Up 13 hours
d432347b718c k8s.gcr.io/pause:3.2 "/pause"
57762fd89f88 k8s.gcr.io/pause:3.2 "/pause"
b59cb5ffb88 bfe3a36ebd25 "/coredns -conf /etc..."
95ae441a8507 k8s.gcr.io/pause:3.2 "/pause"
89b68939cd62 k8s.gcr.io/pause:3.2 "/pause"
33c0840fb6ea 43154ddb57a8 "/usr/local/bin/kube..."
4763e8e9fc45 k8s.gcr.io/pause:3.2 "/pause"
1eed95c4b614 ed2c44fbdd78 "kube-scheduler --au..."
42be0a7c4823 a27166429d98 "kube-controller-man..."
6b6bb9f2dc0_0
4c53fa85ddd5 a8c2fdb8bf76 "kube-apiserver --ad..."
94b6440317b1 0369cf4303ff "etcd --advertise-cl..."
8d6635c35040 k8s.gcr.io/pause:3.2 "/pause"
21e0dacc6237 k8s.gcr.io/pause:3.2 "/pause"
a422e808eec3 k8s.gcr.io/pause:3.2 "/pause"
f968fb7366a3 k8s.gcr.io/pause:3.2 "/pause"
→ ~

```

Volumes

Les **volumes** Kubernetes sont des objets qui servent à persister de la donnée d'un container sur notre machine physique

Il est nécessaire d'utiliser des volumes lorsque notre application contient une base de données, pour sauvegarder la donnée même si le Pod et ses containers sont supprimés pour x raisons

En général, il est recommandé d'utiliser un seul replica du Pod qui gère la base de données. En utiliser plusieurs est source de problèmes, du au fait que plusieurs containers vont utiliser et modifier le même volume.

Il existe différents types de volumes Kubernetes:

Volumes

- **Volume:** Un volume Kubernetes sauvegarde la donnée à l'intérieur d'un Pod. Cela signifie que si un container au sein du Pod crash, le nouveau container créé aura accès à la donnée. Par contre, la donnée sauvegardée ne survivra pas si le Pod lui-même crash. Il n'est donc pas recommandé généralement d'utiliser un Volume
- **Persistent Volume:** Un Persistent Volume Kubernetes sauvegarde la donnée à l'extérieur d'un Pod, dans un endroit spécifique du disque dur de **notre machine**. Ainsi, même dans le cas où le Pod lui-même crash et qu'un nouveau est ensuite créé par le master, ce nouveau Pod aura accès à la donnée sauvegardée dans le Persistent Volume
- **Persistent Volume Claim:** Un Persistent Volume Claim, à la différence d'un Persistent Volume, va attacher au Pod **une partie du disque dur physique** de notre machine.

Imaginons que notre disque dur contient 100 gigas d'espace libre, et que l'on a besoin de 10 gigas pour notre application : Un **PV** (Persistent Volume) sauvegardera la donnée directement dans le disque dur et va mettre les 100 gigas à disposition, alors qu'un **PVC** (Persistent Volume Claim) va créer un espace de 10 gigas pris du disque dur physique qu'il va **attacher** au Pod

Secrets

Les secrets Kubernetes sont des objets qui permettent de créer une information sécurisée dans le Cluster.
On les utilise pour stocker par exemple les mots de passe, comme celui de la connexion à la base de données.
Ainsi, il n'est pas nécessaire de mettre le mot de passe en dur dans le fichier de configuration

Nous verrons comment créer et utiliser un Secret lors du prochain TP

Ingress

Il existe plusieurs moyen de diriger le trafic au sein du réseau. Comme nous l'avons expliqué durant le cours, le service **NodePort** n'est pas recommandé, et le service **LoadBalancer** est une ancienne façon de procéder.

Le meilleur moyen est d'utiliser le service **Ingress**. On spécifie dans un fichier de configuration les règles de routage à mettre en place. Cela va créer deux choses dans le cluster:

- Un Ingress Controller, qui a pour but de vérifier que ces règles sont bien appliqués
- Un Pod contenant nginx, qui va diriger le trafic réseau au sein de notre cluster

Nous verrons comment mettre en place Ingress dans le dernier TP du cours

Commandes Minikube

```
# Démarrer le cluster local  
$ minikube start (--vm-driver=hyperkit)  
  
# Affiche le statut du cluster  
$ minikube status  
  
# Affiche l'adresse IP du cluster  
$ minikube ip  
  
# Arrête le cluster  
$ minikube stop  
  
# Supprime le cluster  
$ minikube delete  
  
# configure le docker-client local pour accéder au docker-server à l'intérieur  
du cluster  
$ eval $(minikube docker-env)  
  
# Ouvre le dashboard minikube  
$ minikube dashboard
```



Commandes Kubernetes

```
● ● ●

# Applique le fichier de config ou les fichiers au sein d'un dossier au sein du
cluster
$ kubectl apply -f <filename or directory>

# Supprime un objet via son fichier de configuration
$ kubectl delete -f <filename>

# Supprime un objet via son nom
$ kubectl delete <object_type> <object_name>

# Affiche les pods du cluster
$ kubectl get pods

# Affiche les pods du cluster avec plus d'informations
$ kubectl get pods -o wide

# Affiche les services du cluster
$ kubectl get services

# Affiche les deployment du cluster
$ kubectl get deployment

# Affiche les persistent volumes du cluster
$ kubectl get pvc

# Affiche les persistent volumes claim du cluster
$ kubectl get pvc

# Affiche les secrets du cluster
$ kubectl get secrets

# Affiche la liste des événements survenus dans le cluster
$ kubectl get events

# Affiche les logs de l'objet
$ kubectl logs <object_name>

# Ouvre un terminal au sein de l'objet spécifié
$ kubectl exec -it <object_name> -- sh

# Affiche la configuration du cluster
$ kubectl config view

# Affiche des informations détaillées sur l'objet qui à le type spécifié
$ kubectl describe <object_type> <object_name>

# Met à jour le déploiement pour récupérer la dernière version de l'image si
celle-ci a été modifiée
$ kubectl rollout restart -f <deployment_file_name>

# Crée un secret de type clé valeur
$ kubectl create secret generic <secret_name> --from-literal key=value
```

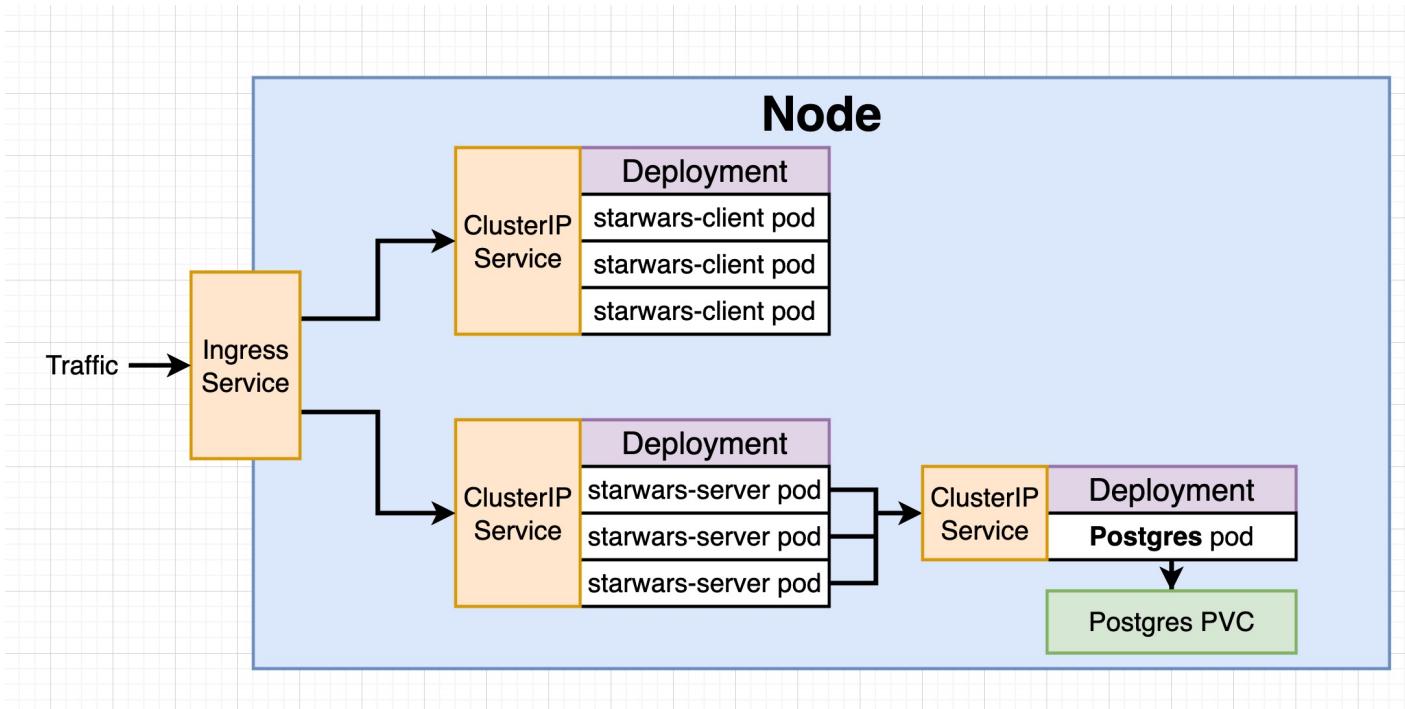
TP

Dans ce dernier TP, vous allez mettre en pratique tout ce que nous avons vu durant le cours sur Kubernetes.

Le but du TP est de transformer notre app star wars qui fonctionne actuellement avec Docker en un cluster Kubernetes.

Voici l'architecture que nous allons mettre en place au sein du cluster.

Le TP est disponible [sur ce lien](#).





Exemple de déploiement Kubernetes

```
image: docker:latest

services:
  - docker:dind

stages:
  - test
  - build
  - deploy

test:
  stage: test
  script: echo "Running tests"
  only:
    - master

.docker-login:
  before_script:
    - echo $DOCKER_PASSWORD | docker login -u $DOCKER_USERNAME --password-stdin

build_back:
  stage: build
  extends: .docker-login
  script:
    - docker build -t elie91/starwars-back -f back/Dockerfile.prod ./back
    - docker push elie91/starwars-back
  only:
    - master

build_front:
  stage: build
  extends: .docker-login
  script:
    - docker build -t elie91/starwars-front -f front/Dockerfile.prod ./front
    - docker push elie91/starwars-front
  only:
    - master

deploy_prod:
  stage: deploy
  image: google/cloud-sdk
  script:
    - echo "Deploy to production server"
    # Se connecte au compte google cloud avec le fichier de credentials
    - echo "$GCP_SERVICE_KEY" > gcloud-service-key.json
    - gcloud auth activate-service-account --key-file gcloud-service-key.json
    # Précise l'id et la zone du projet
    - gcloud config set project $GCP_PROJECT_ID
    - gcloud config set compute/zone europe-west4-a
    - gcloud container clusters get-credentials <project_id>
    - gcloud auth configure-docker
    # Applique les fichiers de configuration kubernetes
    - kubectl apply -f k8s/
    # Restart le back et le front pour prendre en compte la nouvelle image
    - kubectl rollout restart deployment/back-deployment
    - kubectl rollout restart deployment/front-deployment
  environment:
    name: prod
    url: https://$CI_ENVIRONMENT_SLUG.$CI_PROJECT_NAME.$MY_DOMAIN
  only:
    - master
```



QCM



Vos retours sur le cours



Fin du cours