Retour au site

Uptime Formation - Docker - Avril 2021

Supports de formation : Elie Gavoty et Hadrien Pélissier Conçus initialement dans le cadre d'un cursus Uptime Formation. Sous licence CC-BY-NC-SA - Docker - Avril 2021

Table des matières :

Introduction au DevOps
Cours 1 - Présentation
Cours 2 - Les playbooks Ansible
Cours 3 - Organiser un projet
Cours 4 - Sécurité et Cloud
Préparation
TP1 - Mise en place et Ansible ad-hoc
TP2 - Créer un playbook de déploiement d'application flask
TP3 - Structurer le projet avec des roles
TP4 - Orchestration, Serveur de contrôle et Cloud
Bibliographie
0 - Introduction à Docker
1 - Manipulation des conteneurs
TP 1 - Installer Docker et jouer avec
2 - Images et conteneurs
TP 2 - Images et conteneurs
3 - Volumes et réseaux
TP 3 - Réseaux
TP 3bis - Volumes
4 - Créer une application multiconteneur
TP 4 - Créer une application multiconteneur
5 - Orchestration et clustering
TP 5 - Orchestration et clustering
Conclusion
TP 6 (bonus) - Intégration continue avec Gitlab
TP 7 (bonus) - Docker et les reverse proxies
QCM Docker
Cours 1 - Présentation de Kubernetes
Cours 2 - Mettre en place un cluster Kubernetes
TP1 - Installation et configuration de Kubernetes
Cours 3 - Concepts de Kubernetes
Cours 4 - Objets Kubernetes - Partie 1
TP 2 - Déployer Wordpress rapidement
Cours 5 - Le réseau dans Kubernetes
TP 3 - Déployer des conteneurs de A à Z
Rappels Docker
Cours 6 - Objets Kubernetes - Partie 2
Cours 7 - Helm, le gestionnaire de paquets Kubernetes
TP 4 - Déployer Wordpress avec Helm
TP 5 - Cloud Azure

TD opt. - StatefulSets et bases de données
TP opt. - Les ingresses
TP opt. - Le RBAC
Conclusion
Redimensionner le disque d'une machine virtualbox
TP7 - Stratégies de déploiement et monitoring
Traduire des documents

Introduction

Introduction

DevOps

Le nouveau paradigme de l’informatique

Introduction au DevOps

La culture et la pratique du DevOps

A propos de moi

Hadrien Pélissier

- Ingénieur DevOps (Ansible / Docker / Kubernetes / Gitlab CI) / sécurité / développeur Python et Elixir
- Formateur DevOps et sécurité informatique

A propos de vous

- “Profil” : votre environnement technique initial
- Besoins : ce que vous aimeriez faire, avez besoin de savoir faire
- Attentes de cette formation

DevOps : définition

“Le DevOps est **un mouvement** qui s’attaque au conflit existant structurellement entre le développement de logiciels et les opérations. Ce conflit résulte d’objectifs et de motivations divergents. Le DevOps améliore la collaboration entre les départements du développement et des opérations et rationalise l’ensemble de l’organisation. (Citation de

Hütterman 2012 - Devops for developers)"

L'agilité en informatique

- Traditionnellement la qualité logicielle provient :
  - d'une conception détaillée en amont = création d'un spécification détaillée
  - d'un contrôle de qualité humain avant chaque livraison logicielle basé sur une processus = vérification du logiciel par rapport à la spécification
- Problèmes historiques posé par trop de spécification et validation humaine :
  - Lenteur de livraison du logiciel (une version par an ?) donc aussi difficulté de fixer les bugs et problèmes de sécurité a temps
  - Le travail des développeur·euses est dominé par des `process` formels : ennuyeux et abstrait
  - difficulté commerciale : comment répondre à la concurence s'il faut 3 ans pour lancer un produit logiciel.

Solution : développer de façon agile c'est à dire itérative

- Sortir une version par semaine voir par jour
- Créer de petites évolution plutôt que de grosses évolution
- Confronter en permanence le logiciel aux retours clients et utilisateurs

Mais l'agilité traditionnelle ne concerne pas l'administration système.

La motivation au coeur du DevOps : La célérité

- La célérité est : la rapidité (itérative) non pas seulement dans le développement du logiciel mais plus largement dans la livraison du service au client:

Exemple : Netflix ou Spotify ou Facebook etc. déploient une nouvelle version mineure de leur logiciel par jour.

- Lorsque la concurrence peut déployer des innovations en continu il devient central de pouvoir le faire.

Le problème que cherche à résoudre le DevOps

La célérité et l'agrandissementest sont incompatibles avec une administration système traditionnelle:

Dans un DSI (département de service informatique) on organise ces activités d'admin sys en opérations:

- On a un planning d'opération avec les priorités du moment et les trucs moins urgents
- On prépare chaque opération au minimum quelques jours à l'avance.
- On suit un protocole pour pas oublier des étapes de l'opération (pas oublier de faire une sauvegarde avant par exemple)

La difficulté principale pour les Ops c'est qu'un système informatique est:

- Un système très complexe qu'il est quasi **impossible de complètement visualiser** dans sa tête.
- Les **évènements** qui se passe sur la machines sont **instantanés** et **invisibles**
- L'**état actuel** de la machine n'est **pas ou peu explicite** (combien d'utilisateur, machine pas connectée au réseau par exemple.)
- Les **interractions entre des problèmes** peu graves peuvent entrainer des erreurs critiques en cascades.

On peut donc constater que les opérations traditionnelles implique une culture de la **prudence**

- On s'organise à l'avance.
- On vérifie plusieurs fois chaque chose.
- On ne fait pas confiance au code que nous donnent les développeur·euses.
- On suit des procédures pour limiter les risques.
- On surveille l'état du système (on parle de monitoring)
- Et on reçoit même des SMS la nuit si ya un problème :S

Bilan

Les opérations "traditionnelles":

- Peuvent pas aller trop vite car il faut marcher sur des oeufs.
- Les Ops veulent pas déployer de nouvelles versions **trop souvent** car ça fait plein de boulot et ils prennent des risques (bugs / incompatilibités).
- Quand c'est **mal organisé** ou qu'on va **trop vite** il y a des **catastrophes** possibles.

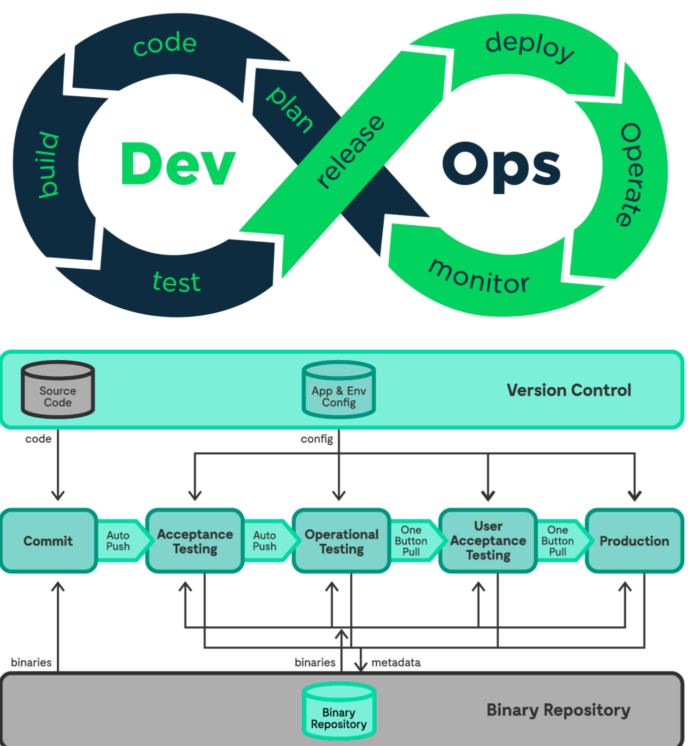
L'objectif technique idéal du DevOps : Intégration et déploiement continus (CI/CD)

Du côté des développeur·euses avec l'agilité on a déjà depuis des années une façon d'automatiser pleins d'opérations sur le code à chaque fois qu'on valide une modification.

- Chaque modification du code est validée dans le gestionnaire de version **Git**.
- Ensuite est envoyée sur le dépot de code commun.
- Des tests logiciels se lancent automatiquement pour s'assurer qu'il n'y a pas de bugs ou de failles.
- Les développeur·euses sont avertis des problèmes.

C'est ce qu'on appelle l'intégration continue.

Le principe central du DevOps est d'automatiser également les opérations de déploiement et de maintenance en se basant sur le même modèle.



Mais pour que ça fonctionne il faut résoudre des défi techniques nouveau => innovations

Renforcer la collaboration

Équipes transversales

Dans le cadre d'un produit logiciel, les administrateurs systèmes sont rassemblées avec le développement et le chef produit : tout le monde fait les réunions ensemble pour se parler et se comprendre.

Culture de la polyvalence

- Les développeur·euses peuvent plus facilement créer un environnement réaliste pour jouer avec et comprendre comment fonctionne l'infrastructure de production (ils progressent dans l'administration système et la compréhension des enjeux opérationnels).
- Les adminsys apprennent à programmer leurs opérations de façon puissante il deviennent donc plus proche de la logique des développeur·euses. (grace à l'Infrastructure as Code)

Le profil DevOps

Par abus de langage on dit un ou une DevOps pour parler d'un métier spécifique dans une entreprise. Je dis que je suis DevOps sur mon CV par exemple.

Vous pouvez retenir :

Un·e DevOps c’est un·e Administrateur·ice Système qui programme ses outils.

Le profil DevOps

Il faut être polyvalent : bien connaître l’administration système Linux mais aussi un peu la programmation et le développement.

Il faut connaître les nouvelles bonnes pratiques et les nouveaux outils cités précédemment.

En résumé

- Un profil ? Un hybride de dev et d’ops…
- Une méthode ? Infra-as-Code, *continuous integration and delivery* (CI/CD), conteneurisation
- Une façon de virer des adminsys… ?

Réancrer les programmes dans la réalité de leur utilisation

“Machines ain’t smart. You are!” Comment dire correctement aux machines quoi faire ?

Solutions techniques

Quelques expressions que vous allez beaucoup entendre:

- Technologies de Cloud (infrastructures à la demande)
- CI / CD
- Infrastructure as Code
- Containerisation

Le cloud

Plutôt que d'**installer manuellement** de nouveaux serveurs linux pour faire tourner des logiciels on peut utiliser des outils pour **faire apparaître de nouveaux serveurs à la demande**.

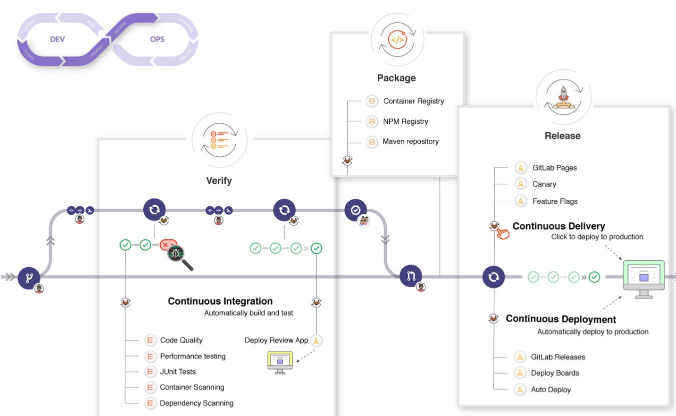
Du coup on peut agrandir sans effort l'infrastructure de production pour délivrer une nouvelle version

C'est ce qu'on appelle le IaaS (Infrastructure as a service)

CI / CD

(intégration continue et déploiement continu)

- Accélérer la livraison des nouvelles versions du logiciel.
- Des tests systématiques et automatisés pour ne pas se reposer sur la vérification humaine.
- Un déploiement progressif en parallèle (Blue/Green) pour pouvoir automatiser le Rollback et être serein.
- A chaque étape le code passe dans un **Pipeline** de validation automatique.



Infrastructure as code

- Permet de régler un problème de l'administration système : Difficultée l'état du système à un instant T ce qui augmente les risques.
- Plutôt que d'appliquer des commandes puis d'oublier si on les a appliqué, On **décrit** le système d'exploitation (l'état du linux) dans un fichier et on utilise un système qui applique cette configuration explicite à tout moment.
- Permet aux Ops/AdminSys de travailler comme des développeur·euses (avec une usine logicielle et ses outils)

Infrastructure As Code

Un mouvement d'informatique lié au DevOps et au cloud :

- Rapprocher la production logicielle et la gestion de l'infrastructure
  - Rapprocher la configuration de dev et de production (+ staging)
  - Assumer le côté imprévisible de l'informatique en ayant une approche expérimentale

- Aller vers de l’intégration et du déploiement continu et automatisé.

Une façon de définir une infrastructure dans un fichier descriptif et ainsi de créer dynamiquement des services.

- Du code qui décrit **l’état désiré** d’un système.
- Arrêtons de faire de l’admin-sys ad-hoc !

Avantages :

- **Descriptif** : on peut lire facilement l'**état actuel** de l’infra
- Git ! Gérer les versions de l’infrastructure et collaborer facilement comme avec du code.
- Tester les instrastructure pour éviter les régressions/bugs
- Facilite l’intégration et le déploiement continus = vélocité = versions testées puis mises en prod' progressivement et automatiquement dans le *cycle DevOps*
- Pas de surprise = possibilité d’agrandir les clusters sans souci !
    - On peut multiplier les machines (une machine ou 100 machines identiques c’est pareil).

Assez différent de l’administration système sur mesure (= méthode de résolution plus ou moins rigoureuse à chaque nouveau bug)

Infrastructure As Code

Concepts proches

- Infrastructure as a Service (commercial et logiciel)
    - Amazon Web Services, Azure, Google Cloud, DigitalOcean
    - = des VM ou des serveurs dédiés
- Plateform as a Service - Heroku, cluster Kubernetes Avec une offre d’hébergement de conteneurs, on parle la plupart du temps de Platform as a Service.

L’infrastructure as code

Il s’agit comme son nom l’indique de gérer les infrastructures en tant que code c’est-à-dire des fichiers textes avec une logique algorithmique/de données et suivis grâce à un gestionnaire de version (git).

Le problème identifié que cherche a résoudre l’IaC est un écheveau de difficulées pratiques rencontrée dans l’administration système traditionnelle:

1. Connaissance limité de l’état courant d’un système lorsqu’on fait de l'**administration**

ad-hoc (manuelle avec des commandes unix/dos).

- Dérive progressive de l'état des systèmes et difficultés à documenter leur états.
- Fiabilité limitée et risques peu maîtrisés lors de certaines opérations transversales (si d'autres méchanismes de fiabilisation n'ont pas été mis en place).
- Problème de communication dans les grandes équipes car l'information est détenue implicitement par quelques personnes.

2. Faible reproductibilité des systèmes et donc difficultée/lenteur du passage à l'échelle (horizontal scaling).

- Multiplier les serveurs identiques est difficile si leur état est le résultat d'un processus manuel partiellement documenté.
- Difficulté à reproduire/simuler l'état précis de l'infrastructure de production dans les contextes de tests logiciels.

3. Difficultés du travail collaboratif dans de grandes équipes avec plusieurs culture (Dev vs Ops) lorsque les rythmes et les modes de travail diffèrent

- L'IaC permet de tout gérer avec git et des commits.
- L'IaC permet aux Ops qui ne le faisait pas de se mettre au code et aux développeur·euses de se confronter plus facilement.
- L'IaC permet d'accélérer la transformation des infrastructures pour l'aligner sur la livraison logicielle quotidienne (idéalement ;))

Containerisation

Les conteneurs (Docker et Kubernetes)

Faire des boîtes isolées avec nos logiciels:

- Un façon standard de packager un logiciel
- Cela permet d'assembler de grosses applications comme des legos
- Cela réduit la complexité grâce:
  - à l'intégration de toutes les dépendance déjà dans la boîte
  - au principe d'immutabilité qui implique de jeter les boîtes (automatiser pour lutter contre la culture prudence). Rend l'infra prédictible.

Docker (et un peu LXC)

Il s'agit de mettre en quelques sortes les logiciels dans des boîtes :

- Avec tout ce qu'il faut pour qu'ils fonctionnent (leurs dépendances).
- Ces boîtes sont fermées (on peut ne peux plus les modifier). On parle d'**immutabilité**.
- Si on a besoin d'un nouvelle version on fait **un nouveau modèle** de boîte. (on dit une nouvelle image docker)

- Cette nouvelle image permet de **créer autant d'instances que nécessaire**.

Containerisation - Pourquoi ?

- **L'isolation des containers permet d'éviter que les logiciels s'emmêlent entre eux. (Les dépendances ne rentrent pas en conflit)**
- Les conteneurs non modifiables permettent de savoir exactement l'état de ce qu'on exécute sur l'ordinateur

> Le risque de bug diminue énormément : **fiabilisation**

- L'agrandissement d'un infrastructure logiciel est beaucoup pour facile lorsqu'on a des boîtes autonomes qu'on peut multiplier.

Préparation

Un peu de logistique

- **Les supports de présentation et les TD sont disponibles à l'adresse https://cours.hadrienpelissier.fr**
- Pour exporter les TD utilisez la fonction d'impression pdf de google chrome.

⚠️ **Pour l'anglais, si un texte ne vous paraît pas clair, quelques liens :**

- Pour les textes : https://www.deepl.com/translator
- Pour les pages web : https://translate.google.com/
- Pour les mots : https://linguee.fr/

Se connecter au lab via Apache Guacamole

- Les TP sont réalisables dans une VM disponible depuis votre navigateur, en allant sur https://lab.hadrienpelissier.fr
- Se connecter avec `votreprenom` (en minuscules) et le mot de passe donné.
- Puis cliquez sur la machine `vnc-votreprenom` (si besoin, le mot de passe dans la VM est le même que celui pour accéder au lab)
- Ouvrez un autre onglet et cliquez aussi sur la machine appelée `vnc-formateur-...`
- Pour faire un **copier-coller** depuis l'extérieur à votre VM, il faut appuyer sur les touches `Ctrl+Alt+Maj` , puis coller ce que l'on veut dans le presse-papier, et refermer la sidebar avec `Ctrl+Alt+Maj` .

Installer quelques logiciels

- Installez VSCode avec le gestionnaire de paquet `snap install code --classic`
- En ligne de commande (`apt`) installez `git` , `htop` , `ncdu`

Explorer Ubuntu Focal Fossa (20.04) : Démo

Explorer l'éditeur VSCode : Démo

Ansible

Module 1

Ansible

Découvrir le couteau suisse de l'automatisation et de l'infrastructure as code.

Plan

Module 1 : Installer ansible, configurer la connexion et commandes ad hoc ansible

Installation

- créer un lab avec LXD
- configurer SSH et python pour utiliser ansible

configurer ansible

- /etc ou ansible.cfg
- configuration de la connexion
- connexion SSH et autres plugins de connection
- versions de Python et d'Ansible

L'inventaire ansible

- gérer des groupes de machines
- L'inventaire est la source d'information principale pour Ansible

Ansible ad-hoc et les modules de base

- la commande `ansible` et ses options
- explorer les nombreux modules d'Ansible
- idempotence des modules
- exécuter correctement des commandes shell avec Ansible
- le check mode pour controller l'état d'une ressource

TP1: Installation, configuration et prise en main avec des commandes ad-hoc

Module 2 : Les playbooks pour déployer une application web

syntaxe yaml des playbooks

- structure d'un playbook

modules de déploiement et configuration

- Templates de configuration avec Jinja2
- gestion des paquets, utilisateurs et fichiers, etc.

Variable et structures de controle

- explorer les variables
- syntaxe jinja des variables et lookups
- facts et variables spéciales
- boucles et conditions

Idempotence d'un playbook

- handlers
- contrôler le statut de retour des tâches
- gestion de l'idempotence des commandes Unix

debugging de playbook

- verbosite
- directive de debug
- gestion des erreurs à l'exécution

TP2: Écriture d'un playbook simple de déploiement d'une application web flask en python.

Module 3 : Structurer un projet, utiliser les rôles

Complexifier notre lab en ajoutant de nouvelles machines dans plusieurs groupes.

- modules de provisionning de machines pour Ansible
- organisation des variables de l'inventaire

- la commande ansible-inventory

Les rôles

- Ansible Galaxy pour installer des rôles.
- Architecture d'un rôle et bonnes pratiques de gestion des rôles.

Écrire un rôle et organiser le projet

- Imports et includes réutiliser du code.
- Bonne pratiques d'organisation d'un projet Ansible
- Utiliser des modules personnalisés et des plugins pour étendre Ansible
- gestion de version du code Ansible

TP3: Transformation de notre playbook en rôle et utilisation de rôles ansible galaxy pour déployer une infrastructure multitiers.

Module 4 : Orchester Ansible dans un contexte de production

Intégration d'Ansible

- Intégrer ansible dans le cloud un inventaire dynamique et Terraform
- Différents type d'intégration Ansible

Orchestration

- Stratégies : Parallélisme de l'exécution
- Délégation de tâche
- Réalisation d'un rolling upgrade de notre application web grace à Ansible
- Inverser des tâches Ansible - stratégies de rollback
- Exécution personnalisée avec des tags

Sécurité

- Ansible Vault : gestion des secrets pour l'infrastructure as code
- desctiver les logs des tâches sensibles
- Renforcer le mode de connexion ansible avec un bastion SSH

Exécution d'Ansible en production

- Intégration et déploiement avec Gitlab
- Gérer une production Ansible découvrir TOWER/AWX
- Tester ses rôles et gérer de multiples versions

TP4: Refactoring de notre code pour effectuer un rolling upgrade et déploiement dans le cloud + AWX

Cours 1 - Présentation

Présentation d’Ansible

Ansible

Ansible est un **gestionnaire de configuration** et un **outil de déploiement et d’orchestration** très populaire et central dans le monde de l'**infrastructure as code** (IaC).

Il fait donc également partie de façon centrale du mouvement **DevOps** car il s’apparente à un véritable **couteau suisse** de l’automatisation des infrastructures.

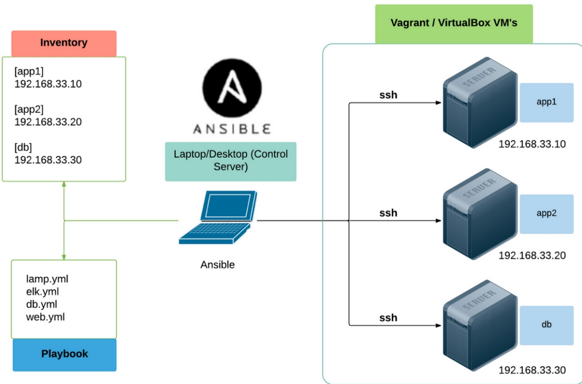
Histoire

Ansible a été créé en **2012** (plus récent que ses concurrents Puppet et Chef) autour d’une recherche de **simplicité** et du principe de configuration **agentless**.

Très orienté linux/opensource et versatile il obtient rapidement un franc succès et s’avère être un couteau suisse très adapté à l’automatisation DevOps et Cloud dans des environnements hétérogènes.

Red Hat rachète Ansible en 2015 et développe un certain nombre de produits autour (Ansible Tower, Ansible container avec Openshift).

Architecture : simplicité et portabilité avec ssh et python



Ansible est **agentless** c’est à dire qu’il ne nécessite aucun service/daemon spécifique sur les machines à configurer.
La simplicité d’Ansible provient également du fait qu’il s’appuie sur des technologies linux omniprésentes et devenues universelles.

- **ssh** : connexion et authentification classique avec les comptes présents sur les machines.
- **python** : multiplateforme, un classique sous linux, adapté à l’admin sys et à tous les usages.

De fait Ansible fonctionne efficacement sur toutes les distributions linux, debian, centos, ubuntu en particulier (et maintenant également sur Windows).

Ansible pour la configuration

Ansible est **semi-déclaratif** c'est à dire qu'il s'exécute **séquentiellement** mais idéalement de façon **idempotente**.

Il permet d'avoir un état descriptif de la configuration:

- qui soit **auditable**
- qui peut **évoluer progressivement**
- qui permet d'**éviter** que celle-ci ne **dérive** vers un état inconnu

Ansible pour le déploiement et l'orchestration

Peut être utilisé pour des **opérations ponctuelles** comme le **déploiement**:

- vérifier les dépendances et l'état requis d'un système
- récupérer la nouvelle version d'un code source
- effectuer une migration de base de données (si outil de migration)
- tests opérationnels (vérifier qu'un service répond)

Ansible à différentes échelles

Les cas d'usages d'Ansible vont de ...:

- petit:
    - ... un petit playbook (~script) fournit avec le code d'un logiciel pour déployer en mode test.
    - ... la configuration d'une machine de travail personnelle.
    - etc.
- moyen:
    - ... faire un lab avec quelques machines.
    - ... déployer une application avec du code, une runtime (php/jav etc) et une base de données à migrer.
    - etc.
- grand:
    - ... gestion de plusieurs DC avec des produits multiples.
    - ... gestion multi-équipes et logging de toutes les opérations grâce à Ansible Tower.
    - etc.

Ansible et Docker

Ansible est très complémentaire à docker:

- Il permet de provisionner des machines avec docker ou kubernetes installé pour ensuite déployer des conteneurs.

- Il permet une orchestration simple des conteneur avec le module `docker_container` .

Plus récemment avec l'arrivé d' `Ansible container` il est possible de construire et déployer des conteneurs docker avec du code ansible. Cette solution fait partie de la stack Red Hat Openshift. Concrêtement le langage ansible remplace (avantageusement ?) le langage Dockerfile pour la construction des images Docker.

Partie 1, Installation, configuration et commandes ad hoc.

Pour l'installation plusieurs options sont possibles:

- Avec le gestionnaire de paquet de la distribution ou homebrew sur OSX:
  - version généralement plus ancienne (2.4 ou 2.6)
  - facile à mettre à jour avec le reste du système
  - Pour installer une version récente on il existe des dépots spécifique à ajouter: exemple sur ubuntu: `sudo apt-add-repository --yes --update ppa:ansible/ansible`
- Avec `pip` le gestionnaire de paquet du langage python: `sudo pip3 install`
  - installe la dernière version stable (2.8 actuellement)
  - commande d'upgrade spécifique `sudo pip3 install ansible --upgrade`
  - possibilité d'installer facilement une version de développement pour tester de nouvelles fonctionnalité ou anticiper les migrations.

Pour voir l'ensemble des fichier installé par un paquet `pip3` :

`pip3 show -f ansible | less`

Pour tester la connexion aux serveurs on utilise la commande ad hoc suivante. `ansible all -m ping`

Les inventaires statiques

Il s'agit d'une liste de machines sur lesquelles vont s'exécuter les modules Ansible. Les machines de cette liste sont:

- Classées par groupe et sous groupes pour être désignables collectivement (exp executer telle opération sur)
- La méthode connexion est précisée soit globalement soit pour chaque machine.
- Des variables peuvent être définies pour chaque machine ou groupe pour contrôler dynamiquement par la suite la configuration ansible.

Exemple :

```
[all:vars]
ansible_ssh_user=elie
ansible_python_interpreter=/usr/bin/python3
```

```
[awx_nodes]
awxnode1 node_state=started ansible_host=10.164.210.101 container_image=cei

[dbservers]
pgnode1 node_state=started ansible_host=10.164.210.111 container_image=cen
pgnode2 node_state=started ansible_host=10.164.210.112 container_image=cen

[appservers]
appnode1 node_state=started ansible_host=10.164.210.121 container_image=cei
appnode2 node_state=started ansible_host=10.164.210.122 container_image=cei
```

Les inventaires peuvent également être au format YAML (plus lisible mais pas toujours intuitif) ou JSON (pour les machines).

Configuration

Ansible se configure classiquement au niveau global dans le dossier `/etc/ansible/` dans lequel on retrouve en autre l'inventaire par défaut et des paramètre de configuration.

Ansible est très fortement configurable pour s'adapter à des environnement contraints. Liste des paramètre de configuration:

Alternativement on peut configurer ansible par projet avec un fichier `ansible.cfg` présent à la racine. Toute commande ansible lancée à la racine du projet récupère automatiquement cette configuration.

La commande `ansible`

- version minimale : `ansible <groupe_machine> -m <module> -a <arguments_module>`
- `ansible all -m ping` : Permet de tester si les hotes sont joignables et ansible utilisable (SSH et python sont présents et configurés).
- version plus complète : `ansible <groupe_machine> --inventory <fichier_inventaire> --become -m <module> -a <arguments_module>`

Les modules Ansible

Ansible fonctionne grâce à des modules python téléversés sur sur l'hôte à configurer puis exécutés. Ces modules sont conçus pour être cohérents et versatiles et rendre les tâches courantes d'administration plus simples.

Il en existe pour un peu toute les tâches raisonnablement courantes : un slogan Ansible "Batteries included" ! Plus de 1300 modules sont intégrés par défaut.

- `ping` : un module de test Ansible (pas seulement réseau comme la commande ping)
- `yum/apt` : pour gérer les paquets sur les distributions basées respectivement sur Red Hat ou Debian.

`... -m yum -a "name=openssh-server state=present"`

- `systemd` (ou plus générique `service`): gérer les services/daemons d'un système.

```
... -m systemd -a "name=openssh-server state=started"
```

- `user` : créer des utilisateurs et gérer leurs options/permission/groupes
- `file` : pour créer, supprimer, modifier, changer les permission de fichiers, dossier et liens.
- `shell` : pour exécuter des commandes unix grace à un shell

Option et documentation des modules

La documentation des modules Ansible se trouve à l'adresse https://docs.ansible.com/ansible/latest/modules/file_module.html

Chaque module propose de nombreux arguments pour personnaliser son comportement:

exemple: le module `file` permet de gérer de nombreuses opérations avec un seul module en variant les arguments.

Il est également à noter que la plupart des arguments sont facultatifs.

- cela permet de garder les appel de modules très succints pour les tâches par défaut
- il est également possible de rendre des paramètres par défaut explicites pour augmenter la clarté du code.

Exemple et bonne pratique: toujours préciser `state: present` même si cette valeur est presque toujours le défaut implicite.

Commençons le TP1

Cours 2 - Les playbooks Ansible

Pour rappel, les avantages du code Ansible, qui donne tout son intérêt à l'Infrastructure-as-Code sont :

- texte descriptif écrit une fois pour toute
- logique lisible et auditable
- versionnable avec Git
- reproductible et incrémental

La dimension incrémentale du code rend en particulier plus aisé de construire une infrastructure progressivement en la complexifiant au fur et à mesure plutôt que de devoir tout plannifier à l'avance.

Le `playbook` est une sorte de script Ansible, c'est-à-dire du code. Le nom provient du football américain : il s'agit d'un ensemble de stratégies qu'une équipe a travaillé pour

répondre aux situations du match. Elle insiste sur la versatilité de l'outil.

Syntaxe YAML

Les playbooks ansible sont écrits au format **YAML**.

- YAML est basé sur les identations à base d'espaces (2 espaces par indentation en général). Comme le langage python.
- C'est un format assez lisible et simple à écrire bien que les indentations soient parfois difficiles à lire.
- C'est un format assez flexible avec des types liste et dictionnaires qui peuvent s'imbriquer.
- Le YAML est assez proche du JSON (leur structures arborescentes typées sont dites isomorphes, en gros faciles à convertir de l'un vers l'autre) mais plus facile à écrire.

A quoi ça ressemble ?

Une liste

```
- 1
- Poire
- "Message à caractère informatif"
```

Un dictionnaire

```
clé1: valeur1
clé2: valeur2
clé3: 3
```

Un exemple imbriqué plus complexe

```
marché: # début du dictionnaire global "marché"
  lieu: Crimée Curial
  jour: dimanche
  horaire:
    unité: "heure"
    min: 9
    max: 14 # entier
  fruits: # liste de dictionnaires décrivant chaque fruit
    - nom: pomme
      couleur: "verte"
      pesticide: avec # les chaines sont avec ou sans " ou '
    - nom: poires
      couleur: jaune
      pesticide: sans
  légumes: # liste de 3 éléments
    - courgettes
    - salade
# on peut sauter des lignes sans interrompre la liste ou le dictionnaire e
```

```
    - potiron
# fin du dictionnaire global
```

Pour mieux visualiser l'imbrication des dictionnaires et des listes en YAML on peut utiliser un convertisseur YAML -> JSON : https://www.json2yaml.com/.

Notre marché devient:

```
{
  "marché": {
    "lieu": "Crimée Curial",
    "jour": "dimanche",
    "horaire": {
      "unité": "heure",
      "min": 9,
      "max": 14
    },
    "fruits": [
      {
        "nom": "pomme",
        "couleur": "verte",
        "pesticide": "avec"
      },
      {
        "nom": "poires",
        "couleur": "jaune",
        "pesticide": "sans"
      }
    ],
    "légumes": [
      "courgettes",
      "salade",
      "potiron"
    ]
  }
}
```

Observez en particulier la syntaxe assez condensée de la liste "fruits" en YAML qui est une liste de dictionnaires.

Structure d'un playbook

```
---
- name: premier play # une liste de play (chaque play commence par un tire
  hosts: serveur_web # un premier play
  become: yes
  gather_facts: false # récupérer le dictionnaires d'informations (facts)

  vars:
    logfile_name: "auth.log"
```

```yaml
    var_files:
      - mesvariables.yml

    pre_tasks:
      - name: dynamic variable
        set_fact:
          mavariable: "{{ inventory_hostname + 'prod' }}" #guillemets obliga

    rôles:
      - flaskapp

    tasks:
      - name: installer le serveur nginx
        apt: name=nginx state=present # syntaxe concise proche des commandes

      - name: créer un fichier de log
        file: # syntaxe yaml extensive : conseillée
          path: /var/log/{{ logfile_name }} #guillemets facultatifs
          mode: 755

      - import_tasks: mestaches.yml

    handlers:
      - systemd:
          name: nginx
          state: "reloaded"

  - name: un autre play
    hosts: dbservers
    tasks:
      ...
```

- Un playbook commence par un tiret car il s'agit d'une liste de plays.
- Un play est un dictionnaire yaml qui décrit un ensemble de tâches ordonnées en plusieurs sections. Un play commence par préciser sur quelles machines il s'applique puis précise quelques paramètres faculatifs d'exécution comme `become: yes` pour l'élévation de privilège (section `hosts`).
- La section `hosts` est obligatoire. Toutes les autres sections sont **facultatives** !
- La section `tasks` est généralement la section principale car elle décrit les tâches de configuration à appliquer.
- La section `tasks` peut être remplacée ou complétée par une section `roles` et des sections `pre_tasks` `post_tasks`
- Les `handlers` sont des tâches conditionnelles qui s'exécutent à la fin (post traitements conditionnels comme le redémarrage d'un service)

Ordre d'exécution

1. `pre_tasks`
2. `roles`
3. `tasks`

4. `post_tasks`
5. `handlers`

Les rôles ne sont pas des tâches à proprement parler mais un ensemble de tâches et ressources regroupées dans un module, un peu comme une librairie dans le développement. Nous explorerons les rôles au cours 3.

Bonnes pratiques de syntaxe

- Indentation de deux espaces.
- Toujours mettre un `name:` qui décrit lors de l'exécution la tâche en cours : un des principes de l'Infrastructure-as-Code est l'intelligibilité des opérations.
- Utiliser les arguments au format YAML (sur plusieurs lignes) pour la lisibilité, sauf s'il y a peu d'arguments

Pour valider la syntaxe il est possible d'installer et utiliser `ansible-linter` sur les fichiers YAML.

Élévation de privilège

L'élévation de privilège est nécessaire lorsqu'on a besoin d'être `root` pour exécuter une commande ou plus généralement qu'on a besoin d'exécuter une commande avec un utilisateur différent de celui utilisé pour la connexion on peut utiliser:

- Au moment de l'exécution l'argument `--become` en ligne de commande avec `ansible`, `ansible-console` ou `ansible-playbook`.
- La section `become: yes`
  - au début du play (après `hosts`) : toutes les tâches seront executée avec cette élévation par défaut.
  - après n'importe quelle tâche : l'élévation concerne uniquement la tâche cible.
- Pour executer une tâche avec un autre utilisateur que root (become simple) ou celui de connexion (sans become) on le précise en ajoutant à `become: yes`, `become_user: username`

Variables Ansible

Ansible utilise en arrière plan un dictionnaire contenant de nombreuses variables.

Pour s'en rendre compte on peut lancer : `ansible <hote_ou_groupe> -m debug -a "msg={{ hostvars }}"`

Ce dictionnaire contient en particulier:

- des variables de configuration ansible (`ansible_user` par exemple)
- des facts c'est à dire des variables dynamiques caractérisant les systèmes cible (par exemple `ansible_os_family`) et récupéré au lancement d'un playbook.
- des variables personnalisées (de l'utilisateur) que vous définissez avec vos propre nom généralement en **snake_case**.

Jinja2 et variables dans les playbooks et rôles (fichiers de code)

La plupart des fichiers Ansible (sauf l'inventaire) sont traités avec le moteur de template python JinJa2.

Ce moteur permet de créer des valeurs dynamiques dans le code des playbooks, des rôles, et des fichiers de configuration.

- Les variables écrites au format `{{ mavariable }}` sont remplacées par leur valeur provenant du dictionnaire d'exécution d'Ansible.
- Des filtres (fonctions de transformation) permettent de transformer la valeur des variables: exemple : `{{ hostname | default('localhost') }}` (Voir plus bas)

Jinja2 et les variables dans les fichiers de templates

Les fichiers de templates (.j2) utilisés avec le module template, généralement pour créer des fichiers de configuration peuvent **contenir des variables** et des **filtres** comme les fichier de code (voir au dessus) **mais également** d'autres constructions jinja2 comme:

- Des `if` : `{% if nginx_state == 'present' %}...{% endif %}` .
- Des boucles `for` : `{% for host in groups['appserver'] %}...{% endfor %}` .
- Des inclusions de templates `{% include 'autre_fichier_template.j2' %}`

Définition des variables

On peut définir et modifier la valeur des variables à différents endroits du code ansible:

- La section `vars:` du playbook.
- Un fichier de variables appelé avec `var_files:`
- L'inventaire : variables pour chaque machine ou pour le groupe.
- Dans des dossier extension de l'inventaire `group_vars` , `host_bars`
- Dans le dossier `defaults` des rôles (cf partie sur les rôles)
- Dans une tâche avec le module `set_facts` .
- A runtime au moment d'appeler la CLI ansible avec `--extra-vars "version=1.23.45 other_variable=foo"`

Lorsque définies plusieurs fois, les variables ont des priorités en fonction de l'endroit de définition. L'ordre de priorité est plutôt complexe: `https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable`

En résumé la règle peut être exprimée comme suit: les variables de runtime sont prioritaires sur les variables dans un playbook qui sont prioritaires sur les variables de l'inventaire qui sont prioritaires sur les variables par défaut d'un rôle.

- Bonne pratique: limiter les redéfinitions de variables en cascade (au maximum une

valeur par défaut, une valeur contextuelle et une valeur runtime) pour éviter que le playbook soit trop complexe et difficilement compréhensible et donc maintenable.

Remarques de syntaxe

- `groups.all` et `groups['all']` sont deux syntaxes équivalentes pour désigner les éléments d'un dictionnaire.

Variables spéciales

https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html

Les plus utiles:

- `hostvars` : dictionaire de toute les variables rangées par hote de l'inventaire.
- `ansible_host` : information utilisée pour la connexion (ip ou domaine).
- `inventory_hostname` : nom de la machine dans l'inventaire.
- `groups` : dictionnaire de tous les groupes avec la liste des machines appartenant à chaque groupe.

Pour explorer chacune de ces variables vous pouvez utiliser le module `debug` en mode adhoc ou dans un playbook:

```
ansible <hote_ou_groupe> -m debug -a "msg={{ ansible_host }}"
```

ou encore:

```
ansible <hote_ou_groupe> -m debug -a "msg={{ groups.all }}"
```

Facts

Les facts sont des valeurs de variables récupérées au début de l'exécution durant l'étape **gather_facts** et qui décrivent l'état courant de chaque machine.

- Par exemple, `ansible_os_family` est un fact/variable décrivant le type d'OS installé sur la machine. Elle n'existe qu'une fois les facts récupérés.

! Lors d'une **commande adhoc** ansible les **facts** ne sont pas récupérés : la variable `ansible_os_family` ne sera pas disponible.

La liste des facts peut être trouvée dans la documentation et dépend des plugins utilisés pour les récupérés:
https://docs.ansible.com/ansible/latest/user_guide/playbooks_vars_facts.html

Structures de controle Ansible (et non Jinja2)

La directive `when`

Elle permet de rendre une tâche conditionnelle (une sorte de `if`)

```yaml
- name: start nginx service
```

```yaml
    systemd:
      name: nginx
      state: started
    when: ansible_os_family == 'RedHat'
```

Sinon la tâche est sautée (skipped) durant l'exécution.

La directive `loop:`

Cette directive permet d'executer une tâche plusieurs fois basée sur une liste de valeur:

https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html

exemple:

```yaml
- hosts: localhost
  tasks:
    - name: exemple de boucle
      debug:
        msg: "{{ item }}"
      loop:
        - message1
        - message2
        - message3
```

On peut également controler cette boucle avec quelques paramètres:

```yaml
- hosts: localhost
  vars:
    messages:
      - message1
      - message2
      - message3

  tasks:
    - name: exemple de boucle
      debug:
        msg: "message numero {{ num }} : {{ message }}"
      loop: "{{ messages }}"
      loop_control:
        loop_var: message
        index_var: num
```

Cette fonctionnalité de boucle était anciennement accessible avec le mot clé `with_items:` qui est maintenant déprécié.

Filtres Jinja

Pour transformer la valeur des variables à la volée lors de leur appel on peut utiliser des filtres (jinja2) :

- par exemple on peut fournir une valeur par défaut pour une variable avec filtre default: `{{ hostname | default('localhost') }}`
- Un autre usage courant des filtres est de reformater et filtrer des listes et dictionnaires de paramètre. Ces syntaxes sont peut intuitives. Vous pouvez vous entrainer en regardant ces tutoriels:
  - https://www.tailored.cloud/devops/how-to-filter-and-map-lists-in-ansible/
  - https://www.tailored.cloud/devops/advanced-list-operations-ansible/

La liste complète des filtres ansible se trouve ici : https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html

Debugger un playbook.

Avec Ansible on dispose d'au moins trois manières de debugger un playbook:

- Rendre la sortie verbeuse (mode debug) avec `-vvv` .
- Utiliser une tâche avec le module `debug` : `debug msg="{{ mavariable }}"` .
- Utiliser la directive `debugger: always` ou `on_failed` à ajouter à la fin d'une tâche. L'exécution s'arrête alors après l'exécution de cette tâche et propose un interpreteur de debug.

Les commandes et l'usage du debugger sont décrits dans la documentation: https://docs.ansible.com/ansible/latest/user_guide/playbooks_debugger.html

Cours 3 - Organiser un projet

Organisation d'un dépôt de code Ansible

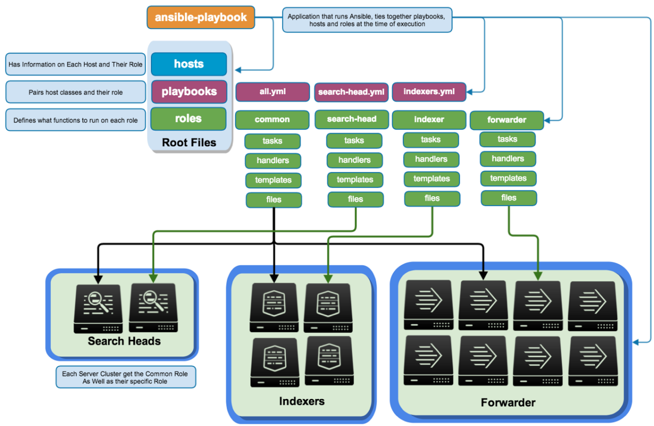
Voici, extrait de la documentation Ansible sur les "Best Practice", l'une des organisations de référence d'un projet ansible de configuration d'une infrastructure:

```
production                # inventory file for production servers
staging                   # inventory file for staging environment

group_vars/
   group1.yml             # here we assign variables to particular groups
   group2.yml
host_vars/
   hostname1.yml          # here we assign variables to particular systems
   hostname2.yml


site.yml                  # master playbook
webservers.yml            # playbook for webserver tier
dbservers.yml             # playbook for dbserver tier

roles/
    common/               # this hierarchy represents a "role"
```

```
        ...                   # role code

    webtier/                  # same kind of structure as "common" was above, (
    monitoring/               # ""
    fooapp/                   # ""
```

Plusieurs remarques:

- Chaque environnement (staging, production) dispose d'un inventaire, ce qui permet de préciser au runtime quel environnement cibler avec l'option `--inventory production` .
- Chaque groupe de serveurs (tier) dispose de son playbook
    - qui s'applique sur le groupe en question.
    - éventuellement définit quelques variables spécifiques (mais il vaut mieux les mettre dans l'inventaire ou les dossiers cf suite).
    - Idéalement contient un minimum de tâches et plutôt des rôles
- Pour limiter la taille de l'inventaire principal on range les variables communes dans des dossiers `group_vars` et `host_vars` . On met à l'intérieur un fichier `<nom_du_groupe>.yml` qui contient un dictionnaire de variables.
- On cherche à modulariser au maximum la configuration dans des rôles, c'est-à-dire des groupes de tâches rendues génériques et specifiques à un objectif de configuration.
- Ce modèle d'organisation correspond plutôt à la **configuration** de base d'une infrastructure (playbooks à exécuter régulièrement) qu'à l'usage de playbooks ponctuels comme pour le déploiement. Mais, bien sûr, on peut ajouter un dossier `playbooks` ou `operations` pour certaines opérations ponctuelles. (cf cours 4)
- Si les modules de Ansible (complétés par les commandes bash) ne suffisent pas on peut développer ses propres modules Ansible.
    - Il s'agit de programmes Python plus ou moins complexes
    - On les range alors dans le dossier `library` du projet ou d'un rôle et on le précise éventuellement dans `ansible.cfg` .
- Observons le rôle `common` : il est utilisé ici pour rassembler les tâches de base et communes à toutes les machines. Par exemple s'assurer que les clés ssh de l'équipe sont présentes, que les dépôts spécifiques sont présents, etc.



Rôles Ansible

Objectif:

- Découper les tâches de configuration en sous-ensembles réutilisables (une suite d'étapes de configuration).
- Ansible est une sorte de langage de programmation et l'intérêt du code est de pouvoir créer des fonctions regroupées en librairies et les composer. Les rôles sont les "librairies" Ansible en quelque sorte.
- Comme une fonction, un rôle prend généralement des paramètres qui permettent de personnaliser son comportement.
- Tout le nécessaire doit y être (fichiers de configurations, archives et binaires à déployer, modules personnels dans `library` etc.)
- Remarque : ne pas confondre **modules** et **roles**. `file` est un module, `geerlingguy.docker` est un rôle. On **doit** écrire des rôles pour coder correctement en Ansible, on **peut** écrire des modules mais c'est largement facultatif car la plupart des actions existent déjà.
- Présentation d'un exemple de rôle : https://github.com/geerlingguy/ansible-role-docker
  - Dans la philosophie Ansible on recherche la généricité des rôles. On cherche à ajouter des paramètres pour que le rôle s'adapte à différents cas (comme notre playbook flask app).
  - Une bonne pratique: préfixer le nom des paramètres par le nom du rôle. Exemple : `docker_edition`.
  - La généricité est nécessaire quand on veut distribuer le rôle ou construire des outils spécifiques qui servent à plusieurs endroit de l'infrastructure
    - mais elle augmente la complexité
    - donc pour les rôles internes on privilégie la simplicité, et le rôle fait sur mesure, plutôt que générique à de nombreux contexte
  - Les rôles contiennent idéalement un fichier `README` pour en décrire l'usage et un fichier `meta/main.yml` qui décrit la compatibilité et les dépendances, en plus de la licence et l'auteur.
  - Il doivent idéalement être versionnés dans des dépôts à part et installés avec la commande `ansible-galaxy`.

Structure d'un rôle

Un rôle est un dossier avec des sous-dossiers conventionnels:

```
roles/
    my_role/               # hiérarchie du rôle "my_role"
        tasks/            #
            main.yml      #  <-- le fichier de tâches exécuté par défaut
        handlers/         #
            main.yml      #  <-- les handlers
        templates/        #  <-- dossier des templates
```

```
        ntp.conf.j2    #  <------- les templates finissent par .j2
    files/             #
        foo.sh         #  <-- d'autres fichiers si nécessaire
    vars/              #
        main.yml       #  <-- variables internes du rôle
    defaults/          #
        main.yml       #  <-- variables par défaut pour le rôle
    meta/              #
        main.yml       #  <-- informations sur le rôle
```

On constate que les noms des sous-dossiers correspondent souvent à des sections du playbook. En fait le principe de base est d'extraire les différentes listes de tâches ou de variables dans des sous-dossiers.

- Remarque : les fichiers de liste **doivent nécessairement** s'appeler **main.yml**" (pas très intuitif)
- Remarque 2 : `main.yml` peut en revanche importer d'autres fichiers aux noms personnalisés (ex: rôle docker de geerlingguy)
- Le dossier `defaults` contient les valeurs par défaut des paramètres du rôle. Ces valeurs ne sont jamais prioritaires (elles sont écrasées par n'importe quelle autre définition de la même variable ailleurs dans le code Ansible)
- Le fichier `meta/main.yml` est facultatif mais conseillé et contient des informations sur le rôle
    - auteur.ice
    - licence
    - compatibilité
    - version
    - dépendances à d'autres rôles.
- Le dossier `files` contient les fichiers qui ne sont pas des templates (pour les module `copy` ou `sync` , `script` etc).

Ansible Galaxy

C'est le store de rôles officiel d'Ansible : https://galaxy.ansible.com/

C'est également le nom d'une commande `ansible-galaxy` qui permet d'installer des rôles et leurs dépendances depuis internet. Un sorte de gestionnaire de paquets pour Ansible.

Elle est utilisée généralement sour la forme `ansible install -r roles/requirements.yml -p roles <nom_role>` , ou plus simplement `ansible-galaxy install <role>` (mais installe dans `/etc/ansible/roles` dans ce cas).

Tous les rôles Ansible sont communautaires (pas de rôles officiels) et généralement stockés sur Github ou Gitlab.

Mais on peut voir la popularité (étoiles Github), et la présence de tests (avec un outil Ansible appelé *Molecule*), qui garantissement la plus ou moins grande fiabilité et qualité du rôle.

Il existe des rôles pour installer un peu n'importe quelle application serveur courante aujourd'hui. Passez du temps à explorer le web avant de développer quelque chose avec Ansible.

Installer des rôles avec `requirements.yml`

Conventionnellement on utilise un fichier `requirements.yml` situé dans `roles` pour décrire la liste des rôles nécessaires à un projet.

```
- src: geerlingguy.repo-epel
- src: geerlingguy.haproxy
- src: geerlingguy.docke
# from GitHub, overriding the name and specifying a specific tag
- src: https://github.com/bennojoy/nginx
  version: master
  name: nginx_role
```

- Ensuite pour les installer on lance: `ansible-galaxy install -r roles/requirements.yml -p roles` .

Imports et includes

Il est possible d'importer le contenu d'autres fichiers dans un playbook:

- `import_tasks` : importe une liste de tâches (atomiques)
- `import_playbook` : importe une liste de play contenus dans un playbook.

Les deux instructions précédentes désignent un import **statique** qui est résolu avant l'exécution.

Au contraire, `include_tasks` permet d'intégrer une liste de tâche **dynamiquement** pendant l'exécution.

Par exemple :

```
vars:
  apps:
    - app1
    - app2
    - app3

tasks:
  - include_tasks: install_app.yml
    loop: "{{ apps }}"
```

Ce code indique à Ansible d'exécuter une série de tâches pour chaque application de la liste. On pourrait remplacer cette liste par une liste dynamique. Comme le nombre d'imports ne peut pas facilement être connu à l'avance on **doit** utiliser `include_tasks` .

Savoir si on doit utiliser `include` ou `import` se fait selon les cas et avec tâtonnement

le plus souvent.

Cours 4 - Sécurité et Cloud

Sécurité

Les problématiques de sécurité Linux ne sont pas du tout résolues magiquement par Ansible. Tous le travail de réflexion et de sécurisation reste identique mais peut, comme le reste, être mieux controllé grâce à l'approche déclarative de l'infrastructure as code.

Si cette problématique des liens entre Ansible et sécurité vous intéresse, il existe un livre appelé `Security automation with Ansible` .

Il est à noter tout de même qu'Ansible est généralement apprécié d'un point de vue sécurité car il n'augmente pas (vraiment) la surface d'attaque de vos infrastructures : il est basé sur ssh qui est éprouvé et ne nécessite généralement pas de réorganisation des infrastructures.

Pour les cas plus spécifiques, Ansible est relativement agnostique du mode de connexion grâce aux plugins de connexions (voir ci-dessous).

Authentification et SSH

Un bonne pratique : changer le port de connexion ssh pour un port atypique. Vous pourrez ajouter la variable `ansible_ssh_port=17728` dans l'inventaire.

Il faut idéalement éviter de créer un seul compte Ansible de connexion pour toutes les machines :

- difficile à bouger
- responsabilité des connexions pas auditable (auth.log + syslog)

Il faut utiliser comme nous avons fait dans les TP des logins ssh avec des utilisateurs aux noms correspondant aux usages ou aux humains derrière, et des clés ssh. C'est-à-dire le même modèle d'authentification que l'administration traditionnelle.

Les autres modes de connexion

Le mode de connexion par défaut de Ansible est SSH, cependant il est possible d'utiliser de nombreux autres modes de connexion spécifiques :

- Pour afficher la liste des plugins disponible lancez `ansible-doc -t connection -l` .
- Une autre connexion courante est `ansible_connection=local` qui permet de configurer la machine locale sans avoir besoin d'installer un serveur ssh.
- Citons également les connexions `ansible_connexion=docker` et

`ansible_connexion=lxd` pour configurer des conteneurs linux ainsi que `ansible_connexion=winrm` pour les serveurs windows
- Les questions de sécurités de la connexion se posent bien sûr différemment selon le mode de connexion utilisé (port, authentification, etc.)
- Pour débugger les connexions et diagnotiquer leur sécurité on peut afficher les détails de chaque connexion ansible avec le mode de verbosité maximal en utilisant le paramètre `-vvvv` .

Variables et secrets

Le principal risque de sécurité lié à Ansible comme avec Docker et l'infrastructure-as-code en général consiste à laisser trainer des secrets (mot de passe, identités de clients, tokens d'API, secrets de chiffrement / migration etc.) dans le code (ou sur les serveurs à des endroits non prévus).

Attention : les dépôts git peuvent cacher des secrets dans leur historique. Pour nettoyer un secret dans un dépôt Git, l'outil le plus courant est BFG : https://rtyley.github.io/bfg-repo-cleaner/

Désactiver le logging des informations sensibles

Ansible propose une directive `no_log: yes` qui permet de désactiver l'affichage des valeurs d'entrée et de sortie d'une tâche.

Il est ainsi possible de limiter la prolifération de données sensibles.

Par exemple, si une tâche change une entrée en base qui contient un mot de passe, `no_log: yes` est tout indiqué.

Ansible Vault

Pour éviter de divulguer des secrets par inadvertance, il est possible de gérer les secrets avec des variables d'environnement ou avec un fichier variable externe au projet qui échappera au versionning git, mais ce n'est pas idéal.

Ansible intègre un trousseau de secrets appelé **Ansible Vault**. Il permet de chiffrer des valeurs **variables par variables** ou via des **fichiers complets**. Les valeurs stockées dans le trousseau sont déchiffrées à l'exécution après déverrouillage du trousseau.

- `ansible-vault create /var/secrets.yml`
- `ansible-vault edit /var/secrets.yml` ouvre `$EDITOR` pour changer le fichier de variables.
- `ansible-vault encrypt_file /vars/secrets.yml` pour chiffrer un fichier existant
- `ansible-vault encrypt_string monmotdepasse` permet de chiffrer une valeur avec un mot de passe. le résultat peut être ensuite collé dans un fichier de variables par ailleurs en clair.

Pour déchiffrer il est ensuite nécessaire d'ajouter l'option `--ask-vault-pass` au moment de l'exécution de `ansible` ou `ansible-playbook`

Il existe également un mode pour gérer plusieurs mots de passe associés à des identifiants.

Ansible dans le cloud

L'automatisation Ansible fait d'autant plus sens dans un environnement dynamique d'infrastructures :

- L'agrandissement horizontal implique de résinstaller régulièrement des machines identiques
- L'automatisation et la gestion des configurations permet de mieux contrôler des environnements de plus en plus complexes.

Il existe de nombreuses solutions pour intégrer Ansible avec les principaux providers de cloud (modules Ansible, plugins d'API, intégration avec d'autre outils d'Infrastructure-as-Code cloud comme Terraform ou Cloudformation).

Inventaires dynamiques

Les inventaires que nous avons utilisés jusqu'ici implique d'affecter à la main les adresses IP des différents noeuds de notre infrastructure. Cela devient vite ingérable si celle-ci change souvent.

La solution Ansible pour ne pas gérer les IP et les groupes à la main est appelée *inventaire dynamique* ou `inventory plugin` . Un inventaire dynamique est simplement un programme qui renvoie un JSON respectant le format d'inventaire JSON Ansible, généralement en contactant l'API du cloud provider ou une autre source.

```
$ ./inventory_terraform.py
{
  "_meta": {
    "hostvars": {
      "balancer0": {
        "ansible_host": "104.248.194.100"
      },
      "balancer1": {
        "ansible_host": "104.248.204.222"
      },
      "awx0": {
        "ansible_host": "104.248.204.202"
      },
      "appserver0": {
        "ansible_host": "104.248.202.47"
      }
    }
  },
  "all": {
    "children": [],
    "hosts": [
      "appserver0",
      "awx0",
      "balancer0",
```

```
        "balancer1"
      ],
      "vars": {}
    },
    "appservers": {
      "children": [],
      "hosts": [
        "balancer0",
        "balancer1"
      ],
      "vars": {}
    },
    "awxnodes": {
      "children": [],
      "hosts": [
        "awx0"
      ],
      "vars": {}
    },
    "balancers": {
      "children": [],
      "hosts": [
        "appserver0"
      ],
      "vars": {}
    }
  }%
```

On peut ensuite appeler `ansible-playbook` en utilisant ce programme plutôt qu'un fichier statique d'inventaire: `ansible-playbook -i inventory_terraform.py configuration.yml`

Étendre et intégrer Ansible

La bonne pratique : utiliser un plugin d'inventaire pour alimenter

Bonne pratique : Normalement l'information de configuration Ansible doit provenir au maximum de l'inventaire. Ceci est conforme à l'orientation plutôt déclarative d'Ansible et à son exécution descendante (master -> nodes). La méthode à privilégier pour intégrer Ansible à des sources d'information existantes est donc d'utiliser ou développer un **plugin d'inventaire**.

https://docs.ansible.com/ansible/latest/plugins/inventory.html

On peut cependant alimenter le dictionnaire de variable Ansible au fur et à mesure de l'exécution, en particulier grâce à la directive `register` et au module `set_fact` .

Exemple:

```
# this is just to avoid a call to |default on each iteration
- set_fact:
```

```
      postconf_d: {}

  - name: 'get postfix default configuration'
    command: 'postconf -d'
    register: postconf_result
    changed_when: false

  # the answer of the command give a list of lines such as:
  # "key = value" or "key =" when the value is null
  - name: 'set postfix default configuration as fact'
    set_fact:
      postconf_d: >
              {{ postconf_d | combine(dict([ item.partition('=')[::2]map'tri
    loop: postconf_result.stdout_lines
```

On peut explorer plus facilement la hiérarchie d'un inventaire statique ou dynamique avec la commande:

```
ansible-inventory --inventory <inventory> --graph
```

Principaux types de plugins possibles pour étendre Ansible

https://docs.ansible.com/ansible/latest/dev_guide/developing_plugins.html

- Ansible modules
- Inventory plugins
- Connection plugins

Intégration Ansible et AWS

Pour les VPS de base Amazon EC2 : utiliser un plugin d'inventaire AWS et les modules adaptés.

- Module EC2: https://docs.ansible.com/ansible/latest/modules/ec2_module.html.
- Plugin d'inventaire:
  https://docs.ansible.com/ansible/latest/plugins/inventory/aws_ec2.html.

Intégration Ansible Nagios

Possibilité 1 : Gérer l'exécution de tâches Ansible et le monitoring Nagios séparément, utiliser le module nagios pour désactiver les alertes Nagios lorsqu'on manipule les ressources monitorées par Nagios.

Possibilité 2 : Laisser le contrôle à Nagios et utiliser un plugin pour que Nagios puisse lancer des plays Ansible en réponse à des évènements sur les sondes.

TP1 - Mise en place et Ansible ad-hoc

Installation de Ansible

- Installez Ansible au niveau du système avec `apt` en lançant:

```
$ sudo apt update
$ sudo apt install software-properties-common
$ sudo apt-add-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible
```

- Affichez la version pour vérifier que c'est bien la dernière stable.

```
ansible --version
=> 2.8.x
```

- Traditionnellement lorsqu'on veut vérifier le bon fonctionnement d'une configuration on utilise `ansible all -m ping`. Que signifie-t-elle ?

Réponse :

- Lancez la commande précédente. Que ce passe-t-il ?

Réponse :

- Utilisez en plus l'option `-vvv` pour mettre en mode très verbeux. Ce mode est très efficace pour **debugger** lorsqu'une erreur inconnue se présente. Que se passe-t-il avec l'inventaire ?

Réponse :

- Testez l'installation avec la commande `ansible` en vous connectant à votre machine `localhost` et en utilisant le module `ping`.

Réponse :

- Ajoutez la ligne `hotelocal ansible_host=127.0.0.1` dans l'inventaire par défaut (le chemin est `/etc/ansible/hosts`). Et pinguer hotelocal.

Réponse :

Explorer LXD

LXD est une technologie de conteneurs actuellement promue par canonical (ubuntu) qui permet de faire des conteneur linux orientés systèmes plutôt qu'application. Par exemple `systemd` est disponible à l'intérieur des conteneurs contrairement aux conteneurs Docker.

LXD est déjà installé et initialisé sur notre ubuntu (sinon `apt install snapd` + `snap install lxd` + ajouter votre utilisateur courant au group unix `lxd`).

Il faut cependant l'initialiser avec : `lxd init`

- Cette commande vous pose un certain nombre de questions pour la configuration et vous pouvez garder TOUTES les valeurs par défaut en fait ENTER simplement à chaque question.
- Affichez la liste des conteneurs avec `lxc list` . Aucun conteneur ne tourne.
- Maintenant lançons notre premier conteneur `centos` avec `lxc launch images:centos/7/amd64 centos1` .
- Listez à nouveau les conteneurs lxc.
- Ce conteneur est un centos minimal et n'a donc pas de serveur SSH pour se connecter. Pour lancez des commandes dans le conteneur on utilise une commande LXC pour s'y connecter `lxc exec <non_conteneur> -- <commande>` . Dans notre cas nous voulons lancer bash pour ouvrir un shell dans le conteneur : `lxc exec centos1 -- bash` .
- Nous pouvons installer des logiciels dans le conteneur comme dans une VM. Pour sortir du conteneur on peut simplement utiliser `exit` .
- Un peu comme avec Docker, LXC utilise des images modèles pour créer des conteneurs. Affichez la liste des images avec `lxc image list` . Trois images sont disponibles l'image centos vide téléchargée et utilisée pour créer centos1 et deux autres images préconfigurée `ubuntu_ansible` et `centos_ansible` . Ces images contiennent déjà la configuration nécessaire pour être utilisée avec ansible (SSH + Python + Un utilisateur + une clé SSH).
- Supprimez la machine centos1 avec `lxc stop centos1 && lxc delete centos1`

Récupérer les images pré-configurées

Pour avoir tous les mêmes images de base générons-les depuis un script pré-installé, dans un terminal lancez :

```
bash /opt/lxd.sh
```

Lancer et tester les conteneurs

Créons à partir des images du remotes un conteneur ubuntu et un autre centos:

```
lxc launch ubuntu_ansible ubu1
lxc launch centos_ansible centos1
```

- Pour se connecter en SSH nous allons donc utiliser une clé SSH appelée `id_stagiaire` qui devrait être présente dans votre dossier `~/.ssh/` . Vérifiez cela en lançant `ls -l /home/stagiaire/.ssh` .
- Déverrouillez cette clé ssh avec `ssh-add ~/.ssh/id_stagiaire` et le mot de passe `devops101` (le ssh-agent doit être démarré dans le shell pour que cette commande

fonctionne si ce n'est pas le cas `eval $(ssh-agent)`).
- Essayez de vous connecter à `ubu1` et `centos1` en ssh pour vérifier que la clé ssh est bien configurée et vérifiez dans chaque machine que le sudo est configuré sans mot de passe avec `sudo -i` .

Créer un projet de code Ansible

Lorsqu'on développe avec Ansible il est conseillé de le gérer comme un véritable projet de code :

- versionner le projet avec Git
- Ajouter tous les paramètres nécessaires dans un dossier pour être au plus proche du code. Par exemple utiliser un inventaire `inventory.cfg` ou `hosts` et une configuration locale au projet `ansible.cfg`

Nous allons créer un tel projet de code pour la suite du tp1

- Créez un dossier projet `tp1` sur le Bureau.

Facultatif :

- Ouvrez Visual Studio Code.
- Installez l'extension Ansible dans VSCode.
- Ouvrez le dossier du projet avec `Open Folder...`

Un projet Ansible implique généralement une configuration Ansible spécifique décrite dans un fichier `ansible.cfg`

- Ajoutez à la racine du projet un tel fichier `ansible.cfg` avec à l'intérieur:

```
[defaults]
inventory = ./inventory.cfg
roles_path = ./roles
host_key_checking = false # nécessaire pour les labs où l'on créé et suppr
```

- Créez le fichier d'inventaire spécifié dans `ansible.cfg` et ajoutez à l'intérieur notre nouvelle machine `hote1` .

Créez et complétez le fichier `inventory.cfg` d'après ce modèle:

```
ubu1 ansible_host=<ip>

[all:vars]
ansible_user=<votre_user>
```

Contacter nos nouvelles machines

Ansible cherche la configuration locale dans le dossier courant. Conséquence : on **lance généralement** toutes les commandes Ansible depuis **la racine de notre projet**.

- Dans le dossier du projet, essayez de relancer la commande ad-hoc `ping` sur cette machine.
- Ansible implique le cas échéant (login avec clé ssh) de déverrouiller la clé ssh pour se connecter à **chaque** hôte. Lorsqu'on en a plusieurs il est donc nécessaire de la déverrouiller en amont avec l'agent ssh pour ne pas perturber l'exécution des commandes ansible. Pour cela : `ssh-add` .
- Créez un groupe `adhoc_lab` et ajoutez les deux machines `ubu1` et `centos1` .

Réponse :

- Lancez `ping` sur les deux machines.

Réponse :

- Nous avons jusqu'à présent utilisé une connexion ssh par clé et précisé l'utilisateur de connexion dans le fichier `ansible.cfg` . Cependant on peut aussi utiliser une connexion par mot de passe et préciser l'utilisateur et le mot de passe dans l'inventaire ou en lançant la commande.

En précisant les paramètres de connexion dans le playbook il et aussi possible d'avoir des modes de connexion différents pour chaque machine.

Installons nginx avec quelques modules et commandes ad-hoc

- Modifiez l'inventaire pour créer deux sous-groupes de `adhoc_lab` , `centos_hosts` et `ubuntu_hosts` avec deux machines dans chacun. (utilisez pour cela `[adhoc_lab:children]`)

```
[all:vars]
ansible_user=<votre_user>

[ubuntu_hosts]
ubu1 ansible_host=<ip>

[centos_hosts]
centos1 ansible_host=<ip>

[adhoc_lab:children]
ubuntu_hosts
centos_hosts
```

Dans un inventaire ansible on commence toujours par créer les plus petits sous groupes puis on les rassemble en plus grands groupes.

- Pinguer chacun des 3 groupes avec une commande ad hoc.

Nous allons maintenant installer `nginx` sur les 2 machines. Il y a plusieurs façons d'installer des logiciels grâce à Ansible: en utilisant le gestionnaire de paquets de la distribution ou un gestionnaire spécifique comme `pip` ou `npm` . Chaque méthode

dispose d'un module ansible spécifique.

- Si nous voulions installer nginx avec la même commande sur des machines centos et ubuntu à la fois impossible d'utiliser `apt` car centos utilise `yum` . Pour éviter ce problème on peut utiliser le module `package` qui permet d'uniformiser l'installation (pour les cas simples).
    - Allez voir la documentation de ce module
    - utilisez `--become` pour devenir root avant d'exécuter la commande (cf élévation de privilège dans le cours2)
    - Utilisez le pour installer `nginx`

Réponse :

- Pour résoudre le problème installez `epel-release` sur la machine centos.

Réponse :

- Relancez la commande d'installation de `nginx` . Que remarque-t-on ?

Réponse :

- Utiliser le module `systemd` et l'option `--check` pour vérifier si le service `nginx` est démarré sur chacune des 2 machines. Normalement vous constatez que le service est déjà démarré (par défaut) sur la machine ubuntu et non démarré sur la machine centos.

Réponse :

- L'option `--check` à vérifier l'état des ressources sur les machines mais sans modifier la configuration`. Relancez la commande précédente pour le vérifier. Normalement le retour de la commande est le même (l'ordre peu varier).
- Lancez la commande avec `state=stopped` : le retour est inversé.
- Enlevez le `--check` pour vous assurer que le service est démarré sur chacune des machines.
- Visitez dans un navigateur l'ip d'un des hôtes pour voir la page d'accueil nginx.

Ansible et les commandes unix

Il existe trois façon de lancer des commandes unix avec ansible:

- le module `command` utilise python pour lancez la commande.
    - les pipes et syntaxes bash ne fonctionnent pas.
    - il peut executer seulement les binaires.
    - il est cependant recommandé quand c'est possible car il n'est pas perturbé par l'environnement du shell sur les machine et donc plus prévisible.
- le module `shell` utilise un module python qui appelle un shell pour lancer une commande.
    - fonctionne comme le lancement d'une commande shell mais utilise un module python.

- le module `raw` .
  - exécute une commande ssh brute.
  - ne nécessite pas python sur l'hote : on peut l'utiliser pour installer python justement.
  - ne dispose pas de l'option `creates` pour simuler de l'idempotence.
- Créez un fichier dans `/tmp` avec `touch` et l'un des modules précédents.
- Relancez la commande. Le retour est toujours `changed` car ces modules ne sont pas idempotents.
- Relancer l'un des modules `shell` ou `command` avec `touch` et l'option `creates` pour rendre l'opération idempotente. Ansible détecte alors que le fichier témoin existe et n'exécute pas la commande.

```
ansible adhoc_lab --become -m "command touch /tmp/file" -a "creates=/tmp/f
```

TP2 - Créer un playbook de déploiement d'application flask

Création du projet

- Créez un nouveau dossier `tp2_flask_deployment` .
- Créez le fichier `ansible.cfg` comme précédemment.

```
[defaults]
inventory = ./inventory.cfg
roles_path = ./roles
host_key_checking = false
```

- Créez l'inventaire statique `inventory.cfg` .

```
[all:vars]
ansible_user=<user>

[appservers]
app1 ansible_host=10.x.y.z
app2 ansible_host=10.x.y.z
```

- Ajoutez à l'intérieur les deux machines dans un groupe `appservers` .
- Pinguez les machines.

```
ansible all -m ping
```

Facultatif :

Premier playbook : installer les dépendances

Le but de ce projet est de déployer une application flask, c'est a dire une application web python. Le code (très minimal) de cette application se trouve sur github à l'adresse: https://github.com/e-lie/flask_hello_ansible.git.

- N'hésitez pas consulter extensivement la documentation des modules avec leur exemple ou d'utiliser la commande de doc `ansible-doc <module>`
- Créons un playbook : ajoutez un fichier `flaskhello_deploy.yml` avec à l'intérieur:

```yaml
- hosts: <hotes_cible>

  tasks:
    - name: ping
      ping:
```

- Lancez ce playbook avec la commande `ansible-playbook <nom_playbook>` .
- Commençons par installer les dépendances de cette application. Tous nos serveurs d'application sont sur ubuntu. Nous pouvons donc utiliser le module `apt` pour installer les dépendances. Il fournit plus d'option que le module `package` .
- Avec le module `apt` installez les applications: `python3-dev` , `python3-pip` , `python3-virtualenv` , `virtualenv` , `nginx` , `git` . Donnez à cette tâche le nom: `ensure basic dependencies are present` . Ajoutez, pour devenir root, la directive `become: yes` au début du playbook.

```yaml
    - name: Ensure apt dependencies are present
      apt:
        name:
          - python3-dev
          - python3-pip
          - python3-virtualenv
          - virtualenv
          - nginx
          - git
        state: present
```

- Lancez ce playbook sans rien appliquer avec la commande `ansible-playbook <nom_playbook> --check --diff` . La partie `--check` indique à Ansible de ne faire aucune modification. La partie `--diff` nous permet d'afficher ce qui changerait à l'application du playbook.
- Relancez bien votre playbook à chaque tâche : comme Ansible est idempotent il n'est pas grave en situation de développement d'interrompre l'exécution du playbook et de reprendre l'exécution après un échec.
- Ajoutez une tâche `systemd` pour s'assurer que le service `nginx` est démarré.

```yaml
  - name: Ensure nginx service started
    systemd:
      name: nginx
      state: started
```

- Ajoutez une tâche pour créer un utilisateur `flask` et l'ajouter au groupe `www-data`. Utilisez bien le paramètre `append: yes` pour éviter de supprimer des groupes à l'utilisateur.

```yaml
  - name: Add the user running webapp
    user:
      name: "flask"
      state: present
      append: yes # important pour ne pas supprimer les groupes d'un uti
      groups:
        - "www-data"
```

Récupérer le code de l'application

- Pour déployer le code de l'application deux options sont possibles.
    - Télécharger le code dans notre projet et le copier sur chaque serveur avec le module `sync` qui fait une copie rsync.
    - Utiliser le module `git`.
- Nous allons utiliser la deuxième option (`git`) qui est plus cohérente pour le déploiement et la gestion des versions logicielles. Allez voir la documentation comment utiliser ce module.
- Utilisez le pour télécharger le code source de l'application (branche `master`) dans le dossier `/home/flask/hello` mais en désactivant la mise à jour (au cas ou le code change).

```yaml
  - name: Git clone/update python hello webapp in user home
    git:
      repo: "https://github.com/e-lie/flask_hello_ansible.git"
      dest: /home/flask/hello
      version: "master"
      clone: yes
      update: no
```

- Lancez votre playbook et allez vérifier sur une machine en ssh que le code est bien téléchargé.

Installez les dépendances python de l'application

Le langage python a son propre gestionnaire de dépendances `pip` qui permet d'installer facilement les librairies d'un projet. Il propose également un méchanisme d'isolation des paquets installés appelé `virtualenv`. Normalement installer les

dépendances python nécessite 4 ou 5 commandes shell.

- La liste de nos dépendances est listée dans le fichier `requirements.txt` à la racine du dossier d'application.
- Nous voulons installer ces dépendances dans un dossier `venv` également à la racine de l'application.
- Nous voulons installer ces dépendances en version python3 avec l'argument `virtualenv_python: python3` .

Avec ces informations et la documentation du module `pip` installez les dépendances de l'application.

Réponse :

Changer les permissions sur le dossier application

Notre application sera executée en tant qu'utilisateur flask pour des raisons de sécurité. Pour cela le dossier doit appartenir à cet utilisateur or il a été créé en tant que root (à cause du `become: yes` de notre playbook).

- Créez une tâche `file` qui change le propriétaire du dossier de façon récursive.

```
- name: Change permissions of app directory
  file:
    path: /home/flask/hello
    state: directory
    owner: "flask"
    recurse: true
```

Module Template : configurer le service qui fera tourner l'application

Notre application doit tourner comme c'est souvent le cas en tant que service (systemd). Pour cela nous devons créer un fichier service adapté `hello.service` dans le le dossier `/etc/systemd/system/` .

Ce fichier est un fichier de configuration qui doit contenir le texte suivant:

```
[Unit]
Description=Gunicorn instance to serve hello
After=network.target

[Service]
User=flask
Group=www-data
WorkingDirectory=/home/flask/hello
Environment="PATH=/home/flask/hello/venv/bin"
ExecStart=/home/flask/hello/venv/bin/gunicorn --workers 3 --bind unix:hell

[Install]
```

```
WantedBy=multi-user.target
```

Pour gérer les fichier de configuration on utilise généralement le module `template` qui permet à partir d'un fichier modèle situé dans le projet ansible de créer dynamiquement un fichier de configuration adapté sur la machine distante.

- Créez un dossier `templates` , avec à l'intérieur le fichier `app.service.j2` contenant le texte précédent.
- Utilisez le module `template` pour le copier au bon endroit avec le nom `hello.service` .
- Utilisez ensuite `systemd` pour démarrer ce service (`state: restarted` ici pour le cas ou le fichier à changé).

Configurer nginx

- Comme précédemment créez un fichier de configuration `hello.test.conf` dans le dossier `/etc/nginx/sites-available` à partir du fichier modèle:

`nginx.conf.j2`

```
server {
    listen 80;

    server_name hello.test;

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/flask/hello/hello.sock;
    }
}
```

- Utilisez `file` pour créer un lien symbolique de ce fichier dans `/etc/nginx/sites-enabled` (avec l'option `force:yes` pour écraser le cas échéant).
- Ajoutez une tâche pour supprimer le site `/etc/nginx/sites-enabled/default` .
- Ajouter une tâche de redémarrage de nginx.
- Ajoutez `hello.test` dans votre fichier `/etc/hosts` pointant sur l'ip d'un des serveur d'application.
- Visitez l'application dans un navigateur et debugger le cas échéant.

Correction intermédiaire

`flaskhello_deploy.yml`

Code de correction :
Facultatif :

Améliorer notre playbook avec des variables.

Variables

Ajoutons des variables pour gérer dynamiquement les paramètres de notre déploiement:

- Ajoutez une section `vars:` avant la section `tasks:` du playbook.
- Mettez dans cette section la variable suivante (dictionnaire):

```yaml
app:
  name: hello
  user: flask
  domain: hello.test
```

- Remplacez dans le playbook précédent et les deux fichiers de template:
  - toutes les occurence de la chaine `hello` par `{{ app.name }}`
  - toutes les occurence de la chaine `flask` par `{{ app.user }}`
  - toutes les occurence de la chaine `hello.test` par `{{ app.domain }}`
- Relancez le playbook : toutes les tâches devraient renvoyer `ok` à part les "restart" car les valeurs sont identiques.

Facultatif :

- Pour la correction clonez le dépôt de base à l'adresse https://github.com/e-lie/ansible_tp_corrections.
- Renommez le clone en tp2_before_handlers.
- ouvrez le projet avec VSCode.
- Activez la branche `tp2_before_handlers_correction` avec `git checkout tp2_before_handlers_correction` .

Le dépôt contient également les corrigés du TP3 et TP4 dans d'autre branches.

Vous pouvez consultez la correction également directement sur le site de github.

Ajouter un handler pour nginx et le service

Pour le moment dans notre playbook, les deux tâches de redémarrage de service sont en mode `restarted` c'est à dire qu'elles redémarrent le service à chaque exécution (résultat: `changed`) et ne sont donc pas idempotentes. En imaginant qu'on lance ce playbook toutes les 15 minutes dans un cron pour stabiliser la configuration, on aurait un redémarrage de nginx 4 fois par heure sans raison.

On désire plutôt ne relancer/recharger le service que lorsque la configuration correspondante a été modifiée. c'est l'objet des tâches spéciales nommées `handlers` .

Ajoutez une section `handlers:` à la suite

- Déplacez la tâche de redémarrage/reload de `nginx` dans cette section et mettez comme nom `reload nginx` .
- Ajoutez aux deux tâches de modification de la configuration la directive `notify:`

`<nom_du_handler>` .

- Testez votre playbook. Il devrait être idempotent sauf le restart de `hello.service` .
- Testez le handler en ajoutant un commentaire dans le fichier de configuration `nginx.conf.j2` .

```
    - name: template nginx site config
      template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx/sites-available/{{ app.domain }}.conf
      notify: reload nginx

      ...

  handlers:
    - name: reload nginx
      systemd:
        name: "nginx"
        state: reloaded

# => penser aussi à supprimer la tâche de restart de nginx précédente
```

Rendre le playbook dynamique avec une boucle.

Plutôt qu'une variable `app` unique on voudrait fournir au playbook une liste d'application à installer (liste potentiellement définie durant l'exécution).

- Identifiez dans le playbook précédent les tâches qui sont exactement communes aux deux installations.

  Réponse :
- Créez un nouveau fichier `deploy_app_tasks.yml` et copier à l'intérieur la liste de toutes les autres tâches mais sans les handlers que vous laisserez à la fin du playbook.
-

Réponse :

- Ce nouveau fichier n'est pas à proprement parler un `playbook` mais une liste de tâches. utilisez `include_tasks:` pour importer cette liste de tâche à l'endroit ou vous les avez supprimées.
- Vérifiez que le playbook fonctionne et est toujours idempotent.
- Ajoutez une tâche `debug: msg={{ app }}` au début du playbook pour visualiser le contenu de la variable.
- Ensuite remplacez la variable `app` par une liste `flask_apps` de deux dictionnaires (avec `name` , `domain` , `user` différents les deux dictionnaires et `repository` et `version` identiques).

```
flask_apps:
  - name: hello
```

```yaml
    domain: "hello.test"
    user: "flask1"
    version: master
    repository: https://github.com/e-lie/flask_hello_ansible.git

  - name: hello2
    domain: "hello2.test"
    user: "flask2"
    version: master
    repository: https://github.com/e-lie/flask_hello_ansible.git
```

- Utilisez les directives `loop` et `loop_control` + `loop_var` sur la tâche `include_tasks` pour inclure les tâches pour chacune des deux applications.
- Créez le dossier `group_vars` et déplacez le dictionnaire `flask_apps` dans un fichier `group_vars/appservers.yml` . Comme son nom l'indique ce dossier permet de définir les variables pour un groupe de serveurs dans un fichier externe.
- Testez en relançant le playbook que le déplacement des variables est pris en compte correctement.

Correction

- Pour la correction clonez le dépôt de base à l'adresse https://github.com/e-lie/ansible_tp_corrections.
- Renommez le clone en tp2.
- ouvrez le projet avec VSCode.
- Activez la branche `tp2_correction` avec `git checkout tp2_correction` .

Le dépôt contient également les corrigés du TP3 et TP4 dans d'autre branches.

Vous pouvez consultez la correction également directement sur le site de github.

Bonus

Pour ceux ou celles qui sont allé-es vite, vous pouvez tenter de créer une nouvelle version de votre playbook portable entre CentOS et ubuntu. Pour cela utilisez la directive `when: ansible_os_family == 'Debian'` ou `RedHat` .

Bonus 2 pour pratiquer

Essayez de déployer une version plus complexe d'application flask avec une base de donnée mysql: https://github.com/miguelgrinberg/microblog/tree/v0.17

Il s'agit de l'application construite au fur et à mesure dans un super tutoriel Python sur Flask. Ce chapitre indique comment déployer l'application sur linux.

TP3 - Structurer le projet avec des roles

Ajouter un provisionneur d'infra maison pour créer les machines automatiquement

- Clonez la correction du TP2 (lien à la fin du TP2) et renommez là en `tp3_provisionner_roles` .
- Chargez ce dossier dans VSCode (vous pouvez fermer le tp2).

Dans notre infra virtuelle, nous avons trois machines dans deux groupes. Quand notre lab d'infra grossit il devient laborieux de créer les machines et affecter les ip à la main. En particulier détruire le lab et le reconstruire est pénible. Nous allons pour cela introduire un playbook de provisionning qui va créer les conteneurs lxd en définissant leur ip à partir de l'inventaire.

- modifiez l'inventaire comme suit:

```
[all:vars]
ansible_user=<votre_user>

[appservers]
app1 ansible_host=10.x.y.121 container_image=ubuntu_ansible node_state=sta
app2 ansible_host=10.x.y.122 container_image=ubuntu_ansible node_state=sta

[dbservers]
db1 ansible_host=10.x.y.131 container_image=ubuntu_ansible node_state=star
```

- Remplacez `x` et `y` dans l'adresse IP par celle fournies par votre réseau virtuel lxd (faites `lxc list` et copier simplement les deux chiffres du milieu des adresses IP)
- Ajoutez un playbook `provision_lxd_infra.yml` dans un dossier `provisionners` contenant:

```
- hosts: localhost
  connection: local

  tasks:
    - name: Setup linux containers for the infrastructure simulation
      lxd_container:
        name: "{{ item }}"
        state: "{{ hostvars[item]['node_state'] }}"
        source:
          type: image
          alias: "{{ hostvars[item]['container_image'] }}"
        profiles: ["default"]
        config:
          security.nesting: 'true'
          security.privileged: 'false'
        devices:
```

```
          # configure network interface
          eth0:
            type: nic
            nictype: bridged
            parent: lxdbr0
            # get ip address from inventory
            ipv4.address: "{{ hostvars[item].ansible_host }}"

        # Comment following line if you installed lxd using apt
        url: unix:/var/snap/lxd/common/lxd/unix.socket
        wait_for_ipv4_addresses: true
        timeout: 600

      register: containers
      loop: "{{ groups['all'] }}"


    # Uncomment following if you want to populate hosts file pour containe
    # AND launch playbook with --ask-become-pass option

    # - name: Config /etc/hosts file accordingly
    #   become: yes
    #   lineinfile:
    #     path: /etc/hosts
    #     regexp: ".*{{ item }}$"
    #     line: "{{ hostvars[item].ansible_host }}    {{ item }}"
    #     state: "present"
    #   loop: "{{ groups['all'] }}"
```

- Etudions le playbook (explication démo).
- Lancez le playbook avec `sudo` car `lxd` se contrôle en root sur localhost: `sudo ansible-playbook provision_lxd_infra` (c'est le seul cas exceptionnel ou ansible-playbook doit être lancé avec sudo, pour les autre playbooks ce n'est pas le cas)
- Lancez `lxc list` pour afficher les nouvelles machines de notre infra et vérifier que le serveur de base de données a bien été créé.

Ajouter une machine MySQL simple avec un rôle externe

Transformer notre playbook en rôle

- Si ce n'est pas fait, créez à la racine du projet le dossier `roles` dans lequel seront rangés tous les rôles (c'est une convention Ansible à respecter).
- Créer un dossier `flaskapp` dans `roles` .
- Ajoutez à l'intérieur l'arborescence:

```
flaskapp
├── defaults
│   └── main.yml
```

```
├── handlers
│   └── main.yml
├── tasks
│   ├── deploy_app_tasks.yml
│   └── main.yml
└── templates
    ├── app.service.j2
    └── nginx.conf.j2
```

- Les templates et les listes de handlers/tasks sont à mettre dans les fichiers correspondants (voir plus bas)
- Le fichier `defaults/main.yml` permet de définir des valeurs par défaut pour les variables du rôle. Mettez à l'intérieur une application par défaut :

```yaml
flask_apps:
  - name: defaultflask
    domain: defaultflask.test
    repository: https://github.com/e-lie/flask_hello_ansible.git
    version: master
    user: defaultflask
```

Ces valeurs seront écrasées par celles fournies dans le dossier `group_vars` (la liste de deux applications du TP2). Elle est présente pour que le rôle fonctionne même en l'absence de variable (valeurs de fallback).

- Copiez les tâches (juste la liste de tirets sans l'intitulé de section `tasks:`) contenues dans le playbook `appservers` dans le fichier `tasks/main.yml` .
- De la même façon, copiez le handler dans `handlers/main.yml` sans l'intitulé `handlers:` .
- Copiez également le fichier `deploy_flask_tasks.yml` dans le dossier `tasks` .
- Déplacez vos deux fichiers de template dans le dossier `templates` du rôle (et non celui à la racine que vous pouvez supprimer).
- Pour appeler notre nouveau rôle, supprimez les sections `tasks:` et `handlers:` du playbook `appservers.yml` et ajoutez à la place:

```yaml
roles:
  - flaskapp
```

- Votre rôle est prêt : lancez `appservers.yml` et debuggez le résultat le cas échéant.

Facultatif: Ajouter un paramètre d'exécution à notre rôle pour mettre à jour l'application

Facultatif :

Correction

- Pour la correction, clonez le dépôt de base à l'adresse https://github.com/e-lie/ansible_tp_corrections.
- Renommez le clone en tp3.
- Ouvrez le projet avec VSCode.
- Activez la branche `tp3_correction` avec `git checkout tp3_correction` .

Il contient également les corrigés du TP2 et TP4 dans d'autres branches.

Bonus

Essayez différents exemples de projets de Geerlingguy accessibles sur GitHub à l'adresse https://github.com/geerlingguy/ansible-for-devops.

TP4 - Orchestration, Serveur de contrôle et Cloud

Cloner le projet modèle

- Pour simplifier le démarrage, clonez le dépôt de base à l'adresse https://github.com/e-lie/ansible_tp_corrections.
- Renommez le clone en tp4.
- ouvrez le projet avec VSCode.
- Activez la branche `tp4_correction` avec `git checkout tp4_correction` .

Facultatif: infrastructure dans le cloud avec Terraform et Ansible

Facultatif :

Infrastructure multi-tiers avec load balancer

Pour configurer notre infrastructure:

- Installez les rôles avec `ansible-galaxy install -r roles/requirements.yml -p roles` .
- Si vous n'avez pas fait la partie Terraform:
    - complétez l'inventaire statique (inventory.cfg)
    - changer dans ansible.cfg l'inventaire en `./inventory.cfg` comme pour les TP précédents
    - Supprimez les conteneurs app1 et app2 du TP précédent puis lancez le playbook de provisionning lxd : `sudo ansible-playbook provisionner/provision_lxd_infra.yml`

- Lancez le playbook global `site.yml`
- Utilisez la commande `ansible-inventory --graph` pour afficher l'arbre des groupes et machines de votre inventaire
- Utilisez la de même pour récupérer l'ip du `balancer0` (ou `balancer1`) avec : `ansible-inventory --host=balancer0`
- Ajoutez `hello.test` et `hello2.test` dans `/etc/hosts` pointant vers l'ip de `balancer0` .
- Chargez les pages `hello.test` et `hello2.test` .
- Observons ensemble l'organisation du code Ansible de notre projet.
  - Nous avons rajouté à notre infrastructure un loadbalancer installé à l'aide du fichier `balancers.yml`
  - Le playbook `upgrade_apps.yml` permet de mettre à jour l'application en respectant sa haute disponibilité. Il s'agit d'une opération d'orchestration simple en les 3 serveurs de notre infrastructure.
  - Cette opération utilise en particulier `serial` qui permet de d'exécuter séquentiellement un play sur un fraction des serveurs d'un groupe (ici 1 à la fois parmis les 2).
  - Notez également l'usage de `delegate` qui permet d'exécuter une tâche sur une autre machine que le groupe initialement ciblé. Cette directive est au coeur des possibilités d'orchestration Ansible en ce qu'elle permet de contacter un autre serveur (déplacement latéral et non pas master -> node) pour récupérer son état ou effectuer une modification avant de continuer l'exécution et donc de coordonner des opérations.
  - notez également le playbook `exclude_backend.yml` qui permet de sortir un backend applicatif du pool. Il s'utilise avec des variables en ligne de commande
- Désactivez le noeud qui vient de vous servir la page en utilisant le playbook `exclude_backend.yml` :

```
ansible-playbook --extra-vars="backend_name=<noeud a desactiver> backend_s
```

- Rechargez la page: vous constatez que c'est l'autre backend qui a pris le relais.
- Nous allons maintenant mettre à jour

Facultatif : ajoutons un serveur de control AWX (/ Ansible Tower)

Facultatif :

Facultatif : Versionner le projet et utiliser la CI gitlab avec Ansible pour automatiser le déploiement

Facultatif :

Bibliographie

Ansible

- Jeff Geerling - Ansible for DevOps - Leanpub

Pour aller plus loin :

- Keating2017 - Mastering Ansible - Second Edition - Packt

Ansible pour des thématiques sépcifiques

- Ratan2017 - Practical Network Automation: Leverage the power of Python and Ansible to optimize your network
- Madhu, Akash2017 - Security automation with Ansible 2
- https://iac.goffinet.org/ansible-network/

Cheatsheet

- https://www.digitalocean.com/community/cheatsheets/how-to-use-ansible-cheat-sheet-guide

Docker

Module 2

Docker

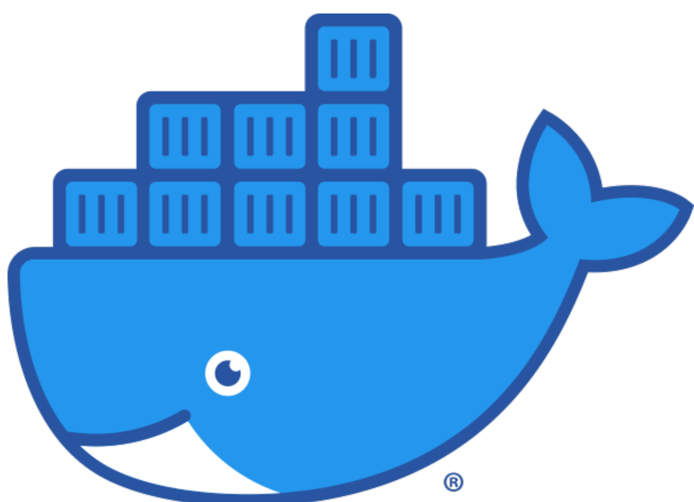
Créer et manipuler des conteneurs

- 0 - Introduction à Docker
- 1 - Manipulation des conteneurs
- TP 1 - Installer Docker et jouer avec
- 2 - Images et conteneurs
- TP 2 - Images et conteneurs
- 3 - Volumes et réseaux
- TP 3 - Réseaux
- TP 3bis - Volumes
- 4 - Créer une application multiconteneur
- TP 4 - Créer une application multiconteneur

- 5 - Orchestration et clustering
- TP 5 - Orchestration et clustering
- Conclusion
- TP 6 (bonus) - Intégration continue avec Gitlab
- TP 7 (bonus) - Docker et les reverse proxies
- QCM Docker

0 - Introduction à Docker

Modularisez et maîtrisez vos applications



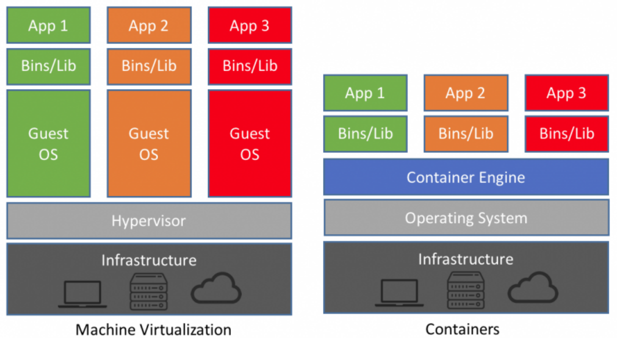
Introduction

- ## La métaphore docker : “box it, ship it”

- Une abstraction qui ouvre de nouvelles possibilités pour la manipulation logicielle.
- Permet de standardiser et de contrôler la livraison et le déploiement.

Retour sur les technologies de virtualisation

On compare souvent les conteneurs aux machines virtuelles. Mais ce sont de grosses simplifications parce qu’on en a un usage similaire : isoler des programmes dans des “contextes”. Une chose essentielle à retenir sur la différence technique : **les conteneurs utilisent les mécanismes internes du _kernel de l’OS Linux_ tandis que les VM tentent de communiquer avec l’OS (quel qu’il soit) pour directement avoir accès au matériel de l’ordinateur.**



- **VM** : une abstraction complète pour simuler des machines
  - un processeur, mémoire, appels systèmes, carte réseau, carte graphique, etc.
- **conteneur** : un découpage dans Linux pour séparer des ressources (accès à des dossiers spécifiques sur le disque, accès réseau).

Les deux technologies peuvent utiliser un système de quotas pour l'accès aux ressources matérielles (accès en lecture/écriture sur le disque, sollicitation de la carte réseau, du processeur)

Si l'on cherche la définition d'un conteneur :

C'est un groupe de *processus* associé à un ensemble de permissions.

L'imaginer comme une "boîte" est donc une allégorie un peu trompeuse, car ce n'est pas de la virtualisation (= isolation au niveau matériel).

Docker Origins : genèse du concept de conteneur

Les conteneurs mettent en œuvre un vieux concept d'isolation des processus permis par la philosophie Unix du "tout est fichier".

`chroot` , `jail` , les 6 `namespaces` et les `cgroups`

`chroot`

- Implémenté principalement par le programme `chroot` [*change root* : changer de racine], présent dans les systèmes UNIX depuis longtemps (1979 !) :

  > "Comme tout est fichier, changer la racine d'un processus, c'est comme le faire changer de système".

`jail`

- `jail` est introduit par FreeBSD en 2002 pour compléter `chroot` et qui permet

pour la première fois une **isolation réelle (et sécurisée) des processus**.

- `chroot` ne s'occupait que de l'isolation d'un process par rapport au système de fichiers :
  - ce n'était pas suffisant, l'idée de "tout-est-fichier" possède en réalité plusieurs exceptions
  - un process *chrooté* n'est pas isolé du reste des process et peut agir de façon non contrôlée sur le système sur plusieurs aspects
- En 2005, Sun introduit les **conteneurs Solaris** décrits comme un « chroot sous stéroïdes » : comme les *jails* de FreeBSD

Les *namespaces* (espaces de noms)

- Les ***namespaces***, un concept informatique pour parler simplement de…
  - groupes séparés auxquels on donne un nom, d'ensembles de choses sur lesquelles on colle une étiquette
  - on parle aussi de **contextes**
- `jail` était une façon de *compléter* `chroot` , pour FreeBSD.
- Pour Linux, ce concept est repris via la mise en place de **namespaces Linux**
  - Les *namespaces* sont inventés en 2002
  - popularisés lors de l'inclusion des 6 types de *namespaces* dans le **noyau Linux** (3.8) en **2013**
- Les conteneurs ne sont finalement que **plein de fonctionnalités Linux saucissonnées ensemble de façon cohérente**.
- Les *namespaces* correspondent à autant de types de **compartiments** nécessaires dans l'architecture Linux pour isoler des processus.

Pour la culture, 6 types de *namespaces* :

- **Les namespaces PID** : "fournit l'isolation pour l'allocation des identifiants de processus (PIDs), la liste des processus et de leurs détails. Tandis que le nouvel espace de nom est isolé de ses adjacents, les processus dans son espace de nommage « parent » voient toujours tous les processus dans les espaces de nommage enfants — quoique avec des numéros de PID différent."
- **Network namespace** : "isole le contrôleur de l'interface réseau (physique ou virtuel), les règles de pare-feu iptables, les tables de routage, etc."
- **Mount namespace** : "permet de créer différents modèles de systèmes de fichiers, ou de créer certains points de montage en lecture-seule"
- **User namespace** : isolates the user IDs between namespaces (dernière pièce du puzzle)
- "UTS" namespace : permet de changer le nom d'hôte.
- IPC namespace : isole la communication inter-processus entre les espaces de nommage.

Les *cgroups* : derniers détails pour une vraie isolation

- Après, il reste à s'occuper de limiter la capacité d'un conteneur à agir sur les ressources matérielles :
  - usage de la mémoire
  - du disque
  - du réseau
  - des appels système
  - du processeur (CPU)
- En 2005, Google commence le développement des **cgroups** : une façon de *tagger* les demandes de processeur et les appels systèmes pour les grouper et les isoler.

Exemple : bloquer le système hôte depuis un simple conteneur

> ```
> :(){ : | :& }; :
> ```

Ceci est une *fork bomb*. Dans un conteneur **non privilégié**, on bloque tout Docker, voire tout le système sous-jacent, en l'empêchant de créer de nouveaux processus.

Pour éviter cela il faudrait limiter la création de processus via une option kernel.

Ex: `docker run -it --ulimit nproc=3 --name fork-bomb bash`

L'isolation des conteneurs n'est donc ni magique, ni automatique, ni absolue ! Correctement paramétrée, elle est tout de même assez **robuste, mature et testée**.

Les conteneurs : définition

On revient à notre définition d'un **conteneur** :

Un conteneur est un groupe de *processus* associé à un ensemble de permissions sur le système.

> 1 container = 1 groupe de *process* Linux
>
> - des *namespaces* (séparation entre ces groups)
> - des *cgroups* (quota en ressources matérielles)

LXC (LinuX Containers)

- En 2008 démarre le projet LXC qui chercher à rassembler :
  - les **cgroups**
  - le **chroot**
  - les **namespaces**.
- Originellement, Docker était basé sur **LXC**. Il a depuis développé son propre assemblage de ces 3 mécanismes.

Docker et LXC

- En 2013, Docker commence à proposer une meilleure finition et une interface simple qui facilite l'utilisation des conteneurs **LXC**.
- Puis il propose aussi son cloud, le **Docker Hub** pour faciliter la gestion d'images toutes faites de conteneurs.
- Au fur et à mesure, Docker abandonne le code de **LXC** (mais continue d'utiliser le **chroot**, les **cgroups** et **namespaces**).
- Le code de base de Docker (notamment **runC**) est open source : l'**Open Container Initiative** vise à standardiser et rendre robuste l'utilisation de containers.

Bénéfices par rapport aux machines virtuelles

Docker permet de faire des "quasi-machines" avec des performances proches du natif.

- Vitesse d'exécution.
- Flexibilité sur les ressources (mémoire partagée).
- Moins **complexe** que la virtualisation
- Plus **standard** que les multiples hyperviseurs
  - notamment moins de bugs d'interaction entre l'hyperviseur et le noyau

Bénéfices par rapport aux machines virtuelles

VM et conteneurs proposent une flexibilité de manipulation des ressources de calcul mais les machines virtuelles sont trop lourdes pour être multipliées librement :

- elles ne sont pas efficaces pour isoler **chaque application**
- elles ne permettent pas la transformation profonde que permettent les conteneurs :
  - le passage à une architecture **microservices**
  - et donc la **scalabilité** pour les besoins des services cloud

Avantages des machines virtuelles

- Les VM se rapprochent plus du concept de “boite noire”: l’isolation se fait au niveau du matériel et non au niveau du noyau de l’OS.
- même si une faille dans l’hyperviseur reste possible car l’isolation n’est pas qu’uniquement matérielle
- Les VM sont-elles “plus lentes” ? Pas forcément.
  - La RAM est-elle un facteur limite ? Non elle n’est pas cher
  - Les CPU pareil : on est rarement bloqués par la puissance du CPU
  - Le vrai problème c’est l’I/O : l’accès en entrée-sortie au disque et au réseau
    - en réalité Docker peut être bien plus lent pour l’implémentation de la sécurité réseau (usage du NAT et du bridging)
    - pareil pour l’accès au disque : la technologie d'*overlay* (qui a une place centrale dans Docker) s’améliore mais reste lente.

La comparaison VM / conteneurs est un thème extrêmement vaste et complexe.

Pourquoi utiliser Docker ?

Docker est pensé dès le départ pour faire des **conteneurs applicatifs** :

- **isoler** les modules applicatifs.
- gérer les **dépendances** en les embarquant dans le conteneur.
- se baser sur l'**immutabilité** : la configuration d’un conteneur n’est pas faite pour être modifiée après sa création.
- avoir un **cycle de vie court** -> logique DevOps du “bétail vs. animal de compagnie”

Pourquoi utiliser Docker ?

Docker modifie beaucoup la **“logistique”** applicative.

- **uniformisation** face aux divers langages de programmation, configurations et briques logicielles
- **installation sans accroc** et **automatisation** beaucoup plus facile
- permet de simplifier l'**intégration continue**, la **livraison continue** et le **déploiement continu**
- **rapproche le monde du développement** des **opérations** (tout le monde utilise la même technologie)
- Permet l’adoption plus large de la logique DevOps (notamment le concept *d’infrastructure as code*)

Infrastructure as Code

Résumé

- on décrit en mode code un état du système. Avantages :
  - pas de dérive de la configuration et du système (immutabilité)
  - on peut connaître de façon fiable l'état des composants du système
  - on peut travailler en collaboration plus facilement (grâce à Git notamment)
  - on peut faire des tests
  - on facilite le déploiement de nouvelles instances

Docker : positionnement sur le marché

- Docker est la technologie ultra-dominante sur le marché de la conteneurisation
  - La simplicité d'usage et le travail de standardisation (un conteneur Docker est un conteneur OCI : format ouvert standardisé par l'Open Container Initiative) lui garantissent légitimité et fiabilité
  - La logique du conteneur fonctionne, et la bonne documentation et l'écosystème aident !
- **LXC** existe toujours et est très agréable à utiliser, notamment avec **LXD** (développé par Canonical, l'entreprise derrière Ubuntu).
  - Il a cependant un positionnement différent : faire des conteneurs pour faire tourner des OS Linux complets.
- **Apache Mesos** : un logiciel de gestion de cluster qui permet de se passer de Docker, mais propose quand même un support pour les conteneurs OCI (Docker) depuis 2016.
- **Podman** : une alternative à Docker qui utilise la même syntaxe que Docker pour faire tourner des conteneurs OCI (Docker) qui propose un mode *rootless* et *daemonless* intéressant.
- **systemd-nspawn** : technologie de conteneurs isolés proposée par systemd

1 - Manipulation des conteneurs

Terminologie et concepts fondamentaux

Deux concepts centraux :

- Une **image** : un modèle pour créer un conteneur

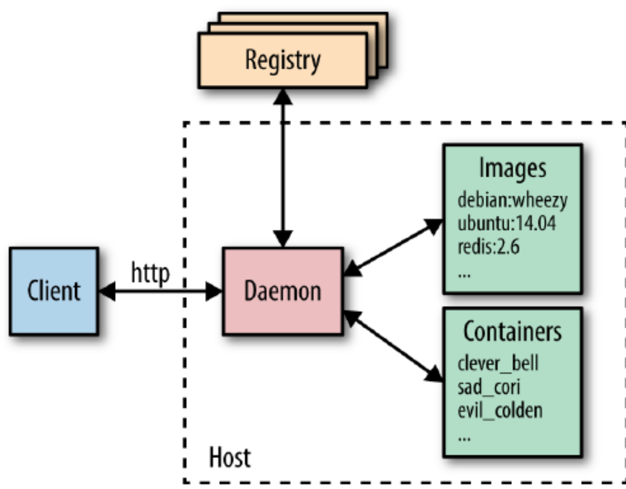
- Un **conteneur** : l'instance qui tourne sur la machine.

Autres concepts primordiaux :

- Un **volume** : un espace virtuel pour gérer le stockage d'un conteneur et le partage entre conteneurs.
- un **registry** : un serveur ou stocker des artefacts docker c'est à dire des images versionnées.
- un **orchestrateur** : un outil qui gère automatiquement le cycle de vie des conteneurs (création/suppression).

Visualiser l'architecture Docker

Daemon - Client - images - registry



L'écosystème Docker

- **Docker Compose** : Un outil pour décrire des applications multiconteneurs.
- **Docker Machine** : Un outil pour gérer le déploiement Docker sur plusieurs machines depuis un hôte.
- **Docker Hub** : Le service d'hébergement d'images proposé par Docker Inc. (le registry officiel)

L'environnement de développement

- Docker Engine pour lancer des commandes docker
- Docker Compose pour lancer des application multiconteneurs
- Portainer, un GUI Docker
- VirtualBox pour avoir une VM Linux quand on est sur Windows

Installer Docker sur Windows ou MacOS

Docker est basé sur le noyau Linux :

- En **production** il fonctionne nécessairement sur un **Linux** (virtualisé ou *bare metal*)
- Pour **développer et déployer**, il marche parfaitement sur **MacOS** et **Windows** mais avec une méthode de **virtualisation** :
  - virtualisation optimisée via un hyperviseur
  - ou virtualisation avec logiciel de virtualisation "classique" comme VMWare ou VirtualBox.

Installer Docker sur Windows

Quatre possibilités :

- Solution WSL2 : on utilise **Docker Desktop WSL2**:
  - Fonctionne avec Windows Subsystem for Linux : c'est une VM Linux très bien intégrée à Windows
  - Le meilleur des deux mondes ?
  - Workflow similaire à celui d'un serveur Linux
- Solution Windows : on utilise **Docker Desktop for Windows**:
  - Fonctionne avec Hyper-V (l'hyperviseur optimisé de Windows)
  - Casse VirtualBox/VMWare (incompatible avec la virtualisation logicielle)
  - Proche du monde Windows et de PowerShell
- Solution VirtualBox : on utilise **Docker Engine** dans une VM Linux
  - Utilise une VM Linux avec VirtualBox
  - Workflow identique à celui d'un serveur Linux
  - Proche de la réalité de l'administration système actuelle
- Solution *legacy* : on utilise **Docker Toolbox** pour configurer Docker avec le **driver VirtualBox** :
  - Change légèrement le workflow par rapport à la version Linux native
  - Marche sur les "vieux" Windows (sans hyperviseur)
  - Utilise une VM Linux avec bash

Installer Docker sous MacOS

- Solution standard : on utilise **Docker Desktop for MacOS** (fonctionne avec la bibliothèque HyperKit qui fait de l'hypervision)
- Solution Virtualbox / *legacy* : On utilise une VM Linux

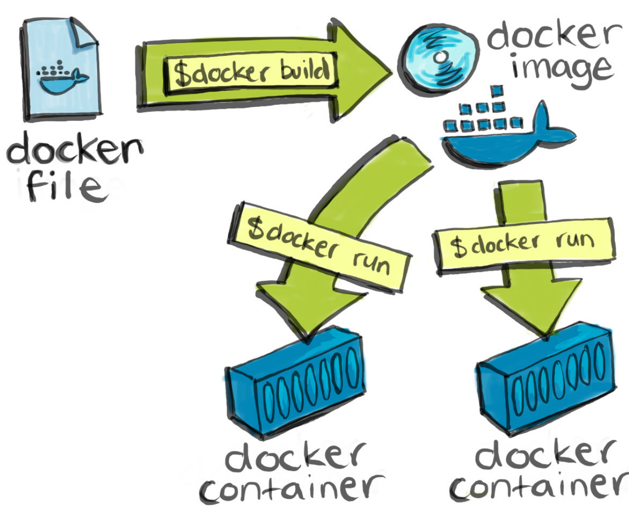
Installer Docker sur Linux

Pas de virtualisation nécessaire car Docker (le Docker Engine) utilise le noyau du système natif.

- On peut l'installer avec le gestionnaire de paquets de l'OS mais cette version peut être trop ancienne.
- Sur **Ubuntu** ou **CentOS** la méthode conseillée est d'utiliser les paquets fournis dans le dépôt officiel Docker (vous pouvez avoir des surprises avec la version *snap* d'Ubuntu).
  - Il faut pour cela ajouter le dépôt et les signatures du répertoire de packages Docker.
  - Documentation Ubuntu : https://docs.docker.com/install/linux/docker-ce/ubuntu/

Les images et conteneurs

Les images



Docker possède à la fois un module pour lancer les applications (runtime) et un **outil de build** d'application.

- Une image est le **résultat** d'un build :
  - on peut la voir un peu comme une boîte "modèle" : on peut l'utiliser plusieurs fois comme base de création de containers identiques, similaires ou différents.

Pour lister les images on utilise :

```
docker images
docker image ls
```

Les conteneurs

- Un conteneur est une instance en cours de fonctionnement ("vivante") d'une image.
  - un conteneur en cours de fonctionnement est un processus (et ses processus enfants) qui tourne dans le Linux hôte (mais qui est isolé de celui-ci)

Commandes Docker

Docker fonctionne avec des sous-commandes et propose de grandes quantités d'options pour chaque commande.

Utilisez `--help` au maximum après chaque commande, sous-commande ou sous-sous-commandes

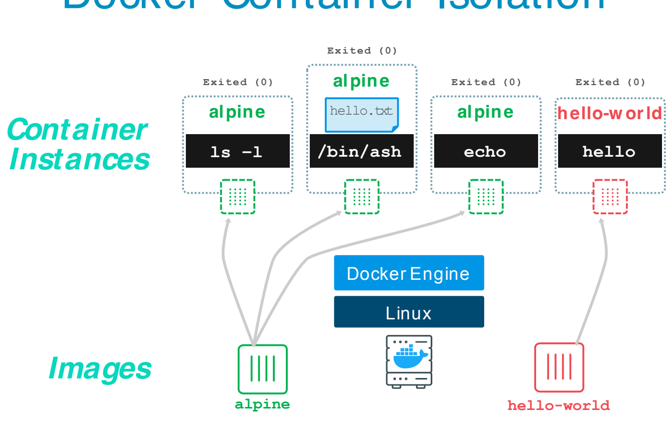
```
docker image --help
```

Pour vérifier l'état de Docker

- Les commandes de base pour connaître l'état de Docker sont :

```
docker info  # affiche plein d'information sur l'engine avec lequel vous ê
docker ps    # affiche les conteneurs en train de tourner
docker ps -a # affiche  également les conteneurs arrêtés
```

Créer et lancer un conteneur



- Un conteneur est une instance en cours de fonctionnement ("vivante") d'une image.

```
docker run [-d] [-p port_h:port_c] [-v dossier_h:dossier_c] <image> <comma
```

> créé et lance le conteneur

- **L'ordre des arguments est important !**
- **Un nom est automatiquement généré pour le conteneur à moins de fixer le nom avec `--name`**
- On peut facilement lancer autant d'instances que nécessaire tant qu'il n'y a **pas de collision** de **nom** ou de **port**.

Options docker run

- Les options facultatives indiquées ici sont très courantes.
  - `-d` permet* de lancer le conteneur en mode **daemon** ou **détaché** et libérer le terminal
  - `-p` permet de mapper un *port réseau* entre l'intérieur et l'extérieur du conteneur, typiquement lorsqu'on veut accéder à l'application depuis l'hôte.
  - `-v` permet de monter un *volume* partagé entre l'hôte et le conteneur.
  - `--rm` (comme *remove*) permet de supprimer le conteneur dès qu'il s'arrête.
  - `-it` permet de lancer une commande en mode *interactif* (un terminal comme `bash`).
  - `-a` (ou `--attach`) permet de se connecter à l'entrée-sortie du processus dans le container.

Commandes Docker

- Le démarrage d'un conteneur est lié à une **commande**.
- Si le conteneur n'a pas de commande, il s'arrête dès qu'il a fini de démarrer

```
docker run debian # s'arrête tout de suite
```

- Pour utiliser une commande on peut simplement l'ajouter à la fin de la commande run.

```
docker run debian echo 'attendre 10s' && sleep 10 # s'arrête après 10s
```

Stopper et redémarrer un conteneur

`docker run` créé un nouveau conteneur à chaque fois.

```
docker stop <nom_ou_id_conteneur> # ne détruit pas le conteneur
docker start <nom_ou_id_conteneur> # le conteneur a déjà été créé
docker start --attach <nom_ou_id_conteneur> # lance le conteneur et s'atta
```

Isolation des conteneurs

- Les conteneurs sont plus que des processus, ce sont des boîtes isolées grâce aux **namespaces** et **cgroups**
- Depuis l'intérieur d'un conteneur, on a l'impression d'être dans un Linux autonome.
- Plus précisément, un conteneur est lié à un système de fichiers (avec des dossiers `/bin` , `/etc` , `/var` , des exécutables, des fichiers…), et possède des métadonnées (stockées en `json` quelque part par Docker)
- Les utilisateurs Unix à l'intérieur du conteneur ont des UID et GID qui existent classiquement sur l'hôte mais ils peuvent correspondre à un utilisateur Unix sans droits sur l'hôte si on utilise les *user namespaces*.

Introspection de conteneur

- La commande `docker exec` permet d'exécuter une commande à l'intérieur du conteneur **s'il est lancé**.
- Une utilisation typique est d'introspecter un conteneur en lançant `bash` (ou `sh`).

```
docker exec -it <conteneur> /bin/bash
```

Docker Hub : télécharger des images

Une des forces de Docker vient de la distribution d'images :

- pas besoin de dépendances, on récupère une boîte autonome
- pas besoin de multiples versions en fonction des OS

Dans ce contexte un élément qui a fait le succès de Docker est le Docker Hub : hub.docker.com

Il s'agit d'un répertoire public et souvent gratuit d'images (officielles ou non) pour des milliers d'applications pré-configurées.

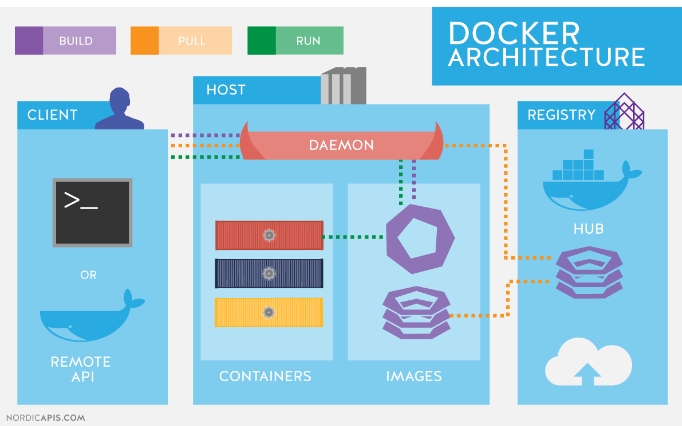
Docker Hub:

- On peut y chercher et trouver presque n'importe quel logiciel au format d'image Docker.
- Il suffit pour cela de chercher l'identifiant et la version de l'image désirée.
- Puis utiliser `docker run [<compte>/]<id_image>:<version>`
- La partie `compte` est le compte de la personne qui a poussé ses images sur le Docker Hub. Les images Docker officielles (`ubuntu` par exemple) ne sont pas liées à un compte : on peut écrire simplement `ubuntu:focal` .

- On peut aussi juste télécharger l'image : `docker pull <image>`

On peut également y créer un compte gratuit pour pousser et distribuer ses propres images, ou installer son propre serveur de distribution d'images privé ou public, appelé **registry**.

En résumé



TP 1 - Installer Docker et jouer avec

Premier TD : on installe Docker et on joue avec

Installer Docker sur la VM Ubuntu dans Guacamole

- Accédez à votre VM via l'interface Guacamole
- Pour accéder au copier-coller de Guacamole, il faut appuyer sur `Ctrl+Alt+Shift` et utiliser la zone de texte qui s'affiche (réappuyer sur `Ctrl+Alt+Shift` pour revenir à la VM).
- Pour installer Docker, suivez la documentation officielle pour installer Docker sur Ubuntu, depuis "Install using the repository" jusqu'aux deux commandes `sudo apt-get update` et `sudo apt-get install docker-ce docker-ce-cli containerd.io` .
  - Docker nous propose aussi une installation en une ligne (*one-liner*), moins sécurisée : `curl -sSL https://get.docker.com | sudo sh`
- Lancez `sudo docker run hello-world` . Bien lire le message renvoyé (le traduire sur Deepl si nécessaire). Que s'est-il passé ?
- Il manque les droits pour exécuter Docker sans passer par `sudo` à chaque fois.
  - Le daemon tourne toujours en `root`

- Un utilisateur ne peut accéder au client que s'il est membre du groupe `docker`
  - Ajoutez-le au groupe avec la commande `usermod -aG docker <user>` (en remplaçant `<user>` par ce qu'il faut)
  - Pour actualiser la liste de groupes auquel appartient l'utilisateur, redémarrez la VM avec `sudo reboot` puis reconnectez-vous avec Guacamole pour que la modification sur les groupes prenne effet.

Autocomplétion

- Pour vous faciliter la vie, ajoutez le plugin *autocomplete* pour Docker et Docker Compose à `bash` en copiant les commandes suivantes :

```
sudo apt update
sudo apt install bash-completion curl
sudo mkdir /etc/bash_completion.d/
sudo curl -L https://raw.githubusercontent.com/docker/docker-ce/master/com
sudo curl -L https://raw.githubusercontent.com/docker/compose/1.24.1/contr
```

Important: Vous pouvez désormais appuyer sur la touche pour utiliser l'autocomplétion quand vous écrivez des commandes Docker

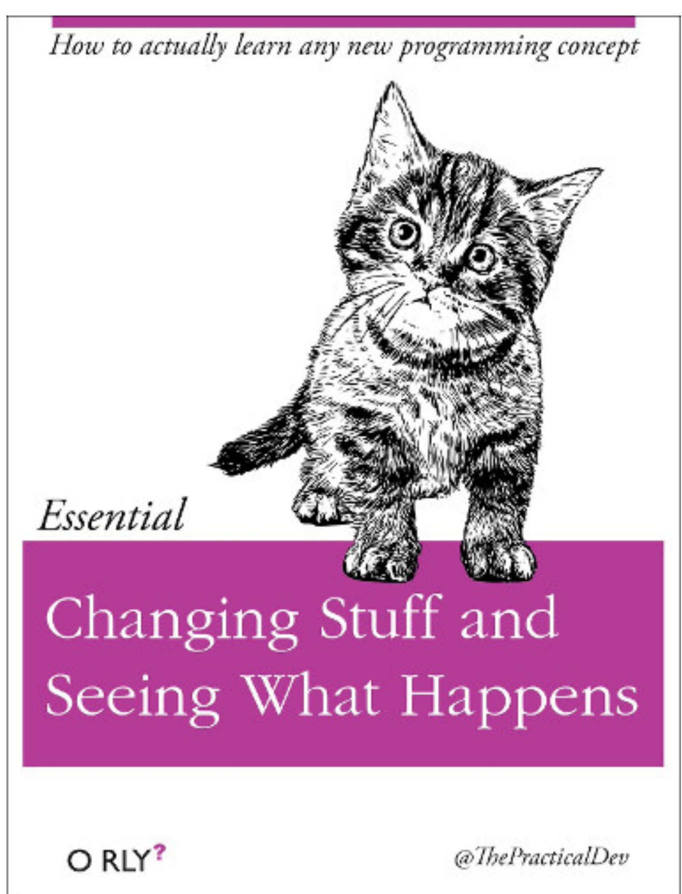
Pour vérifier l'installation

- Les commandes de base pour connaître l'état de Docker sont :

```
docker info  # affiche plein d'information sur l'engine avec lequel vous ê
docker ps    # affiche les conteneurs en train de tourner
docker ps -a # affiche  également les conteneurs arrêtés
```

Manipuler un conteneur

- **Commandes utiles :** https://devhints.io/docker
- **Documentation** `docker run` **:** https://docs.docker.com/engine/reference/run/

Mentalité : 

Il faut aussi prendre l'habitude de bien lire ce que la console indique après avoir passé vos commandes.

Avec l'aide du support et de `--help` , et en notant sur une feuille ou dans un fichier texte les commandes utilisées :

- Lancez simplement un conteneur Debian en mode *attached*. Que se passe-t-il ?

Résultat :

- Lancez un conteneur Debian (`docker run` puis les arguments nécessaires, cf. l'aide `--help`) en mode détaché avec la commande `echo "Debian container"` . Rien n'apparaît. En effet en mode détaché la sortie standard n'est pas connectée au terminal.
- Lancez `docker logs` avec le nom ou l'id du conteneur. Vous devriez voir le résultat de la commande `echo` précédente.

Résultat :

- Affichez la liste des conteneurs en cours d'exécution

Solution :

- Affichez la liste des conteneurs en cours d'exécution et arrêtés.

Solution :

- Lancez un conteneur debian **en mode détaché** avec la commande `sleep 3600`
- Réaffichez la liste des conteneurs qui tournent
- Tentez de stopper le conteneur, que se passe-t-il ?

```
docker stop <conteneur>
```

NB: On peut désigner un conteneur soit par le nom qu'on lui a donné, soit par le nom généré automatiquement, soit par son empreinte (toutes ces informations sont indiquées dans un `docker ps` ou `docker ps -a`). L'autocomplétion fonctionne avec les deux noms.

- Trouvez comment vous débarrasser d'un conteneur récalcitrant (si nécessaire, relancez un conteneur avec la commande `sleep 3600` en mode détaché).

Solution :

- Tentez de lancer deux conteneurs avec le nom `debian_container`

Solution :
Le nom d'un conteneur doit être unique (à ne pas confondre avec le nom de l'image qui est le modèle utilisé à partir duquel est créé le conteneur).

- Créez un conteneur avec le nom `debian2`

```
docker run debian -d --name debian2 sleep 500
```

- Lancez un conteneur debian en mode interactif (options `-i -t`) avec la commande `/bin/bash` et le nom `debian_interactif` .
- Explorer l'intérieur du conteneur : il ressemble à un OS Linux Debian normal.

Chercher sur Docker Hub

- Visitez hub.docker.com
- Cherchez l'image de Nginx (un serveur web), et téléchargez la dernière version (`pull`).

```
docker pull nginx
```

- Lancez un conteneur Nginx. Notez que lorsque l'image est déjà téléchargée le lancement d'un conteneur est quasi instantané.

```
docker run --name "test_nginx" nginx
```

Ce conteneur n'est pas très utile, car on a oublié de configurer un port ouvert.

- Trouvez un moyen d'accéder quand même au Nginx à partir de l'hôte Docker (indice : quelle adresse IP le conteneur possède-t-il ?).

Solution :

- Arrêtez le(s) conteneur(s) `nginx` créé(s).
- Relancez un nouveau conteneur `nginx` avec cette fois-ci le port correctement configuré dès le début pour pouvoir visiter votre Nginx en local.

```
docker run -p 8080:80 --name "test2_nginx" nginx # la syntaxe est : port_h
```

- En visitant l'adresse et le port associé au conteneur Nginx, on doit voir apparaître des logs Nginx dans son terminal car on a lancé le conteneur en mode *attached*.
- Supprimez ce conteneur. NB : On doit arrêter un conteneur avant de le supprimer, sauf si on utilise l'option "-f".

On peut lancer des logiciels plus ambitieux, comme par exemple Funkwhale, une sorte d'iTunes en web qui fait aussi réseau social :

```
docker run --name funky_conteneur -p 80:80 funkwhale/all-in-one:1.0.1
```

Vous pouvez visiter ensuite ce conteneur Funkwhale sur le port 80 (après quelques secondes à suivre le lancement de l'application dans les logs) ! Mais il n'y aura hélas pas de musique dedans :(

Attention à ne jamais lancer deux containers connectés au même port sur l'hôte, sinon cela échouera !

- Supprimons ce conteneur :

```
docker rm -f funky_conteneur
```

Facultatif : Wordpress, MYSQL et les variables d'environnement

- Lancez un conteneur Wordpress joignable sur le port `8080` à partir de l'image officielle de Wordpress du Docker Hub
- Visitez ce Wordpress dans le navigateur

Nous pouvons accéder au Wordpress, mais il n'a pas encore de base MySQL configurée. Ce serait un peu dommage de configurer cette base de données à la main. Nous allons configurer cela à partir de variables d'environnement et d'un deuxième conteneur créé à partir de l'image `mysql` .

Depuis Ubuntu:

- Il va falloir mettre ces deux conteneurs dans le même réseau (nous verrons plus tarde ce que cela implique), créons ce réseau :

```
docker network create wordpress
```

- Cherchez le conteneur `mysql` version 5.7 sur le Docker Hub.
- Utilisons des variables d'environnement pour préciser le mot de passe root, le nom de la base de données et le nom d'utilisateur de la base de données (trouver la documentation sur le Docker Hub).
- Il va aussi falloir définir un nom pour ce conteneur

Résultat :

- inspectez le conteneur MySQL avec `docker inspect`
- Faites de même avec la documentation sur le Docker Hub pour préconfigurer l'app Wordpress.
- En plus des variables d'environnement, il va falloir le mettre dans le même réseau, et exposer un port

Solution :

- regardez les logs du conteneur Wordpress avec `docker logs`
- visitez votre app Wordpress et terminez la configuration de l'application : si les deux conteneurs sont bien configurés, on ne devrait pas avoir à configurer la connexion à la base de données
- avec `docker exec`, visitez votre conteneur Wordpress. Pouvez-vous localiser le fichier `wp-config.php` ? Une fois localisé, utilisez `docker cp` pour le copier sur l'hôte.

Faire du ménage

- Lancez la commande `docker ps -aq -f status=exited`. Que fait-elle ?
- Combinez cette commande avec `docker rm` pour supprimer tous les conteneurs arrêtés (indice : en Bash, une commande entre les parenthèses de " `$()` " est exécutée avant et utilisée comme chaîne de caractère dans la commande principale)

Solution :

- S'il y a encore des conteneurs qui tournent (`docker ps`), supprimez un des conteneurs restants en utilisant l'autocomplétion et l'option adéquate
- Listez les images
- Supprimez une image
- Que fait la commande `docker image prune -a` ?

Décortiquer un conteneur

- En utilisant la commande `docker export votre_conteneur -o conteneur.tar`, puis `tar -C conteneur_decompresse -xvf conteneur.tar` pour décompresser un conteneur Docker, explorez (avec l'explorateur de fichiers par exemple) jusqu'à trouver l'exécutable principal contenu dans le conteneur.

Portainer

Portainer est un portail web pour gérer une installation Docker via une interface graphique. Il va nous faciliter la vie.

- Lancer une instance de Portainer :

```
docker volume create portainer_data
docker run --detach --name portainer \
    -p 9000:9000 \
    -v portainer_data:/data \
    -v /var/run/docker.sock:/var/run/docker.sock \
    portainer/portainer-ce
```

- Remarque sur la commande précédente : pour que Portainer puisse fonctionner et contrôler Docker lui-même depuis l'intérieur du conteneur il est nécessaire de lui donner accès au socket de l'API Docker de l'hôte grâce au paramètre `--mount` ci-dessus.
- Visitez ensuite la page http://localhost:9000 ou l'adresse IP publique de votre serveur Docker sur le port 9000 pour accéder à l'interface.
- il faut choisir l'option "local" lors de la configuration
- Créez votre user admin et choisir un mot de passe avec le formulaire.
- Explorez l'interface de Portainer.
- Créez un conteneur.

2 - Images et conteneurs

Créer une image en utilisant un Dockerfile

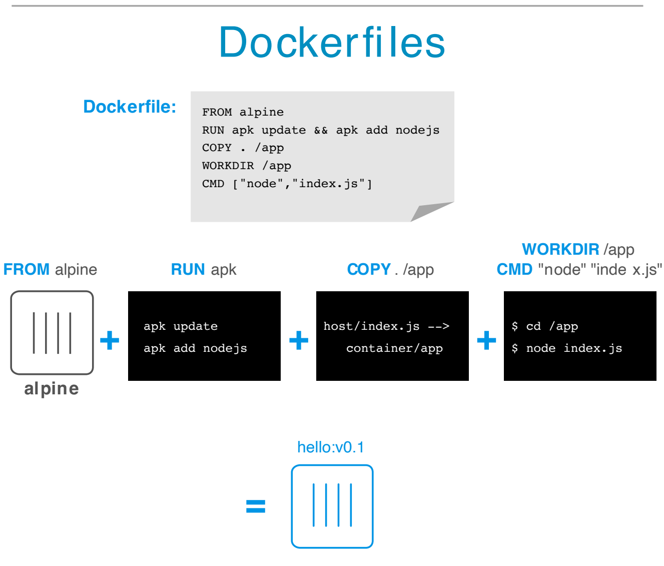
- Jusqu'ici nous avons utilisé des images toutes prêtes.
- Une des fonctionnalités principales de Docker est de pouvoir facilement construire des images à partir d'un simple fichier texte : **le Dockerfile**.

Le processus de build Docker

- Un image Docker ressemble un peu à une VM car on peut penser à un Linux "freezé" dans un état.
- En réalité c'est assez différent : il s'agit uniquement d'un système de fichier (par couches ou *layers*) et d'un manifeste JSON (des méta-données).
- Les images sont créés en empilant de nouvelles couches sur une image existante grâce à un système de fichiers qui fait du *union mount*.

- Chaque nouveau build génère une nouvelle image dans le répertoire des images (`/var/lib/docker/images`) (attention ça peut vite prendre énormément de place)

- On construit les images à partir d'un fichier `Dockerfile` en décrivant procéduralement (étape par étape) la construction.



Exemple de Dockerfile :

```
FROM debian:latest

RUN apt update && apt install htop

CMD ['sleep 1000']
```

- La commande pour construire l'image est :

```
docker build [-t tag] [-f dockerfile] <build_context>
```

- généralement pour construire une image on se place directement dans le dossier avec le `Dockerfile` et les élements de contexte nécessaire (programme, config, etc), le contexte est donc le caractère `.` , il est obligatoire de préciser un contexte.
- exemple : `docker build -t mondebian .`

- Le **Dockerfile** est un fichier procédural qui permet de décrire l'installation d'un logiciel (la configuration d'un container) en enchaînant des instructions Dockerfile (en MAJUSCULE).
- Exemple:

```
# our base image
FROM alpine:3.5

# Install python and pip
RUN apk add --update py2-pip
```

```
# upgrade pip
RUN pip install --upgrade pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000

# run the application
CMD ["python", "/usr/src/app/app.py"]
```

Instruction `FROM`

- L'image de base à partir de laquelle est construite l'image actuelle.

Instruction `RUN`

- Permet de lancer une commande shell (installation, configuration).

Instruction `ADD`

- Permet d'ajouter des fichier depuis le contexte de build à l'intérieur du conteneur.
- Généralement utilisé pour ajouter le code du logiciel en cours de développement et sa configuration au conteneur.

Instruction `CMD`

- Généralement à la fin du `Dockerfile` : elle permet de préciser la commande par défaut lancée à la création d'une instance du conteneur avec `docker run` . on l'utilise avec une liste de paramètres

```
CMD ["echo 'Conteneur démarré'"]
```

Instruction `ENTRYPOINT`

- Précise le programme de base avec lequel sera lancé la commande

```
ENTRYPOINT ["/usr/bin/python3"]
```

`CMD` et `ENTRYPOINT`

- Ne surtout pas confondre avec `RUN` qui exécute une commande Bash uniquement pendant la construction de l'image.

L'instruction `CMD` a trois formes :

- `CMD ["executable","param1","param2"]` (*exec form*, forme à préférer)
- `CMD ["param1","param2"]` (combinée à une instruction `ENTRYPOINT`)
- `CMD command param1 param2` (*shell form*)

Si l'on souhaite que notre container lance le même exécutable à chaque fois, alors on peut opter pour l'usage d' `ENTRYPOINT` en combination avec `CMD` .

Instruction `ENV`

- Une façon recommandée de configurer vos applications Docker est d'utiliser les variables d'environnement UNIX, ce qui permet une configuration "au *runtime*".

Instruction `HEALTHCHECK`

`HEALTHCHECK` permet de vérifier si l'app contenue dans un conteneur est en bonne santé.

```
HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit 1
```

Les variables

On peut utiliser des variables d'environnement dans les Dockerfiles. La syntaxe est `${...}` . Exemple :

```
FROM busybox
ENV FOO=/bar
WORKDIR ${FOO}   # WORKDIR /bar
ADD . $FOO       # ADD . /bar
COPY \$FOO /quux # COPY $FOO /quux
```

Se référer au mode d'emploi pour la logique plus précise de fonctionnement des variables.

Documentation

- Il existe de nombreuses autres instructions possibles très clairement décrites dans la

documentation officielle : https://docs.docker.com/engine/reference/builder/

Lancer la construction

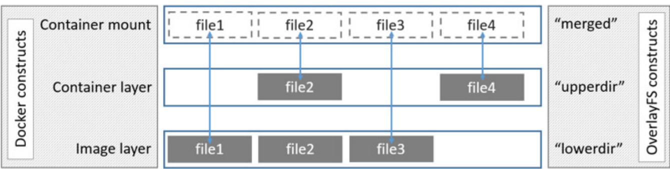
- La commande pour lancer la construction d'une image est :

```
docker build [-t <tag:version>] [-f <chemin_du_dockerfile>] <contexte_de_c
```

- Lors de la construction, Docker télécharge l'image de base. On constate plusieurs téléchargements en parallèle.
- Il lance ensuite la séquence des instructions du Dockerfile.
- Observez l'historique de construction de l'image avec `docker image history <image>`
- Il lance ensuite la série d'instructions du Dockerfile et indique un *hash* pour chaque étape.
    - C'est le *hash* correspondant à un *layer* de l'image

Les layers et la mise en cache

- **Docker construit les images comme une série de "couches" de fichiers successives.**
- On parle d'**Union Filesystem** car chaque couche (de fichiers) écrase la précédente.



- Chaque couche correspond à une instruction du Dockerfile.
- `docker image history <conteneur>` permet d'afficher les layers, leur date de construction et taille respectives.
- Ce principe est au coeur de l'**immutabilité** des images Docker.
- Au lancement d'un container, le Docker Engine rajoute une nouvelle couche de filesystem "normal" read/write par dessus la pile des couches de l'image.
- `docker diff <container>` permet d'observer les changements apportés au conteneur depuis le lancement.

Optimiser la création d'images

- Les images Docker ont souvent une taille de plusieurs centaines de **mégaoctets** voire parfois **gigaoctets**. `docker image ls` permet de voir la taille des images.

- Or, on construit souvent plusieurs dizaines de versions d'une application par jour (souvent automatiquement sur les serveurs d'intégration continue).
    - L'espace disque devient alors un sérieux problème.
- Le principe de Docker est justement d'avoir des images légères car on va créer beaucoup de conteneurs (un par instance d'application/service).
- De plus on télécharge souvent les images depuis un registry, ce qui consomme de la bande passante.

> La principale **bonne pratique** dans la construction d'images est de **limiter leur taille au maximum**.

Limiter la taille d'une image

- Choisir une image Linux de base **minimale**:
    - Une image `ubuntu` complète pèse déjà presque une soixantaine de mégaoctets.
    - mais une image trop rudimentaire (`busybox`) est difficile à débugger et peu bloquer pour certaines tâches à cause de binaires ou de bibliothèques logicielles qui manquent (compilation par exemple).
    - Souvent on utilise des images de base construites à partir de `alpine` qui est un bon compromis (6 mégaoctets seulement et un gestionnaire de paquets `apk`).
    - Par exemple `python3` est fourni en version `python:alpine` (99 Mo), `python:3-slim` (179 Mo) et `python:latest` (918 Mo).

Les multi-stage builds

Quand on tente de réduire la taille d'une image, on a recours à un tas de techniques. Avant, on utilisait deux `Dockerfile` différents : un pour la version prod, léger, et un pour la version dev, avec des outils en plus. Ce n'était pas idéal. Par ailleurs, il existe une limite du nombre de couches maximum par image (42 layers). Souvent on enchaînait les commandes en une seule pour économiser des couches (souvent, les commandes `RUN` et `ADD`), en y perdant en lisibilité.

Maintenant on peut utiliser les multistage builds.

Avec les multi-stage builds, on peut utiliser plusieurs instructions `FROM` dans un Dockerfile. Chaque instruction `FROM` utilise une base différente. On sélectionne ensuite les fichiers intéressants (des fichiers compilés par exemple) en les copiant d'un stage à un autre.

Exemple de `Dockerfile` utilisant un multi-stage build :

```
FROM golang:1.7.3 AS builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Créer des conteneurs personnalisés

- Il n'est pas nécessaire de partir d'une image Linux vierge pour construire un conteneur.
- On peut utiliser la directive `FROM` avec n'importe quelle image.
- De nombreuses applications peuvent être configurées en étendant une image officielle
- *Exemple : une image Wordpress déjà adaptée à des besoins spécifiques.*
- L'intérêt ensuite est que l'image est disponible préconfigurée pour construire ou mettre à jour une infrastructure, ou lancer plusieurs instances (plusieurs containers) à partir de cette image.
- C'est grâce à cette fonctionnalité que Docker peut être considéré comme un outil d'*infrastructure as code*.
- On peut également prendre une sorte de snapshot du conteneur (de son système de fichiers, pas des processus en train de tourner) sous forme d'image avec `docker commit <image>` et `docker push` .

Publier des images vers un registry privé

- Généralement les images spécifiques produites par une entreprise n'ont pas vocation à finir dans un dépôt public.
- On peut installer des **registries privés**.
- On utilise alors `docker login <adresse_repo>` pour se logger au registry et le nom du registry dans les `tags` de l'image.
- Exemples de registries :
  - **Gitlab** fournit un registry très intéressant car intégré dans leur workflow DevOps.

TP 2 - Images et conteneurs

Découverte d'une application web flask

- Récupérez d'abord une application Flask exemple en la clonant :

```
git clone https://github.com/uptime-formation/microblog/
```

- Ouvrez VSCode avec le dossier `microblog` en tapant `code microblog` ou bien en lançant VSCode avec `code` puis en cliquant sur `Open Folder` .
- Dans VSCode, vous pouvez faire `Terminal > New Terminal` pour obtenir un terminal en bas de l'écran.

- Observons ensemble le code dans VSCode.

Passons à Docker

Déployer une application Flask manuellement à chaque fois est relativement pénible. Pour que les dépendances de deux projets Python ne se perturbent pas, il faut normalement utiliser un environnement virtuel `virtualenv` pour séparer ces deux apps. Avec Docker, les projets sont déjà isolés dans des conteneurs. Nous allons donc construire une image de conteneur pour empaqueter l'application et la manipuler plus facilement. Assurez-vous que Docker est installé.

Pour connaître la liste des instructions des Dockerfiles et leur usage, se référer au manuel de référence sur les Dockerfiles.

- Dans le dossier du projet ajoutez un fichier nommé `Dockerfile` et sauvegardez-le
- Normalement, VSCode vous propose d'ajouter l'extension Docker. Il va nous faciliter la vie, installez-le. Une nouvelle icône apparaît dans la barre latérale de gauche, vous pouvez y voir les images téléchargées et les conteneurs existants. L'extension ajoute aussi des informations utiles aux instructions Dockerfile quand vous survolez un mot-clé avec la souris.
- Ajoutez en haut du fichier : `FROM ubuntu:latest` Cette commande indique que notre image de base est la dernière version de la distribution Ubuntu.

- Nous pouvons déjà contruire un conteneur à partir de ce modèle Ubuntu vide : `docker build -t microblog .`
- Une fois la construction terminée lancez le conteneur.
- Le conteneur s'arrête immédiatement. En effet il ne contient aucune commande bloquante et nous n'avons précisé aucune commande au lancement. Pour pouvoir observer le conteneur convenablement il fautdrait faire tourner quelque chose à l'intérieur. Ajoutez à la fin du fichier la ligne : `CMD ["/bin/sleep", "3600"]` Cette ligne indique au conteneur d'attendre pendant 3600 secondes comme au TP précédent.
- Reconstruisez l'image et relancez un conteneur

- Affichez la liste des conteneurs en train de fonctionner
- Nous allons maintenant rentrer dans le conteneur en ligne de commande pour observer. Utilisez la commande : `docker exec -it <id_du_conteneur> /bin/bash`
- Vous êtes maintenant dans le conteneur avec une invite de commande. Utilisez quelques commandes Linux pour le visiter rapidement (`ls` , `cd` ...).
- Il s'agit d'un Linux standard, mais il n'est pas conçu pour être utilisé comme un système complet, juste pour une application isolée. Il faut maintenant ajouter notre application Flask à l'intérieur. Dans le Dockerfile supprimez la ligne CMD, puis ajoutez :

```
RUN apt-get update -y
RUN apt-get install -y python3-pip
```

- Reconstruisez votre image. Si tout se passe bien, poursuivez.
- Pour installer les dépendances python et configurer la variable d'environnement Flask ajoutez:

```
COPY ./requirements.txt /requirements.txt
RUN pip3 install -r requirements.txt
ENV FLASK_APP microblog.py
```

- Reconstruisez votre image. Si tout se passe bien, poursuivez.
- Ensuite, copions le code de l'application à l'intérieur du conteneur. Pour cela ajoutez les lignes :

```
COPY ./ /microblog
WORKDIR /microblog
```

Cette première ligne indique de copier tout le contenu du dossier courant sur l'hôte dans un dossier `/microblog` à l'intérieur du conteneur. Nous n'avons pas copié les requirements en même temps pour pouvoir tirer partie des fonctionnalités de cache de Docker, et ne pas avoir à retélécharger les dépendances de l'application à chaque fois que l'on modifie le contenu de l'app.

Puis, dans la 2e ligne, le dossier courant dans le conteneur est déplacé à `/` .

- Reconstruisez votre image. **Observons que le build recommence à partir de l'instruction modifiée. Les layers précédents avaient été mis en cache par le Docker Engine.**
- Si tout se passe bien, poursuivez.
- Enfin, ajoutons la section de démarrage à la fin du Dockerfile, c'est un script appelé `boot.sh` :

```
CMD ["./boot.sh"]
```

- Reconstruisez l’image et lancez un conteneur basé sur l’image en ouvrant le port `5000` avec la commande : `docker run -p 5000:5000 microblog`
- Naviguez dans le navigateur à l’adresse `localhost:5000` pour admirer le prototype microblog.
- Lancez un deuxième container cette fois avec : `docker run -d -p 5001:5000 microblog`
- Une deuxième instance de l’app est maintenant en fonctionnement et accessible à l’adresse `localhost:5001`

Docker Hub

- Avec `docker login`, `docker tag` et `docker push`, poussez l’image `microblog` sur le Docker Hub. Créez un compte sur le Docker Hub le cas échéant.

Solution :

Améliorer le Dockerfile

Une image plus simple

- A l’aide de l’image `python:3-alpine` et en remplaçant les instructions nécessaires (pas besoin d’installer `python3-pip` car ce programme est désormais inclus dans l’image de base), repackagez l’app microblog en une image taggée `microblog:slim` ou `microblog:light`. Comparez la taille entre les deux images ainsi construites.

Faire varier la configuration en fonction de l’environnement

Le serveur de développement Flask est bien pratique pour debugger en situation de développement, mais n’est pas adapté à la production. Nous pourrions créer deux images pour les deux situations mais ce serait aller contre l’impératif DevOps de rapprochement du dev et de la prod.

Pour démarrer l’application, nous avons fait appel à un script de boot `boot.sh` avec à l’intérieur :

```
#!/bin/bash

# ...

set -e
if [ "$APP_ENVIRONMENT" = 'DEV' ]; then
    echo "Running Development Server"
    exec flask run -h 0.0.0.0
else
    echo "Running Production Server"
    exec gunicorn -b :5000 --access-logfile - --error-logfile - app_name:a
fi
```

- Déclarez maintenant dans le Dockerfile la variable d'environnement `APP_ENVIRONMENT` avec comme valeur par défaut `PROD` .
- Construisez l'image avec `build` .
- Puis, grâce aux bons arguments allant avec `docker run` , lancez une instance de l'app en configuration `PROD` et une instance en environnement `DEV` (joignables sur deux ports différents).
- Avec `docker ps` ou en lisant les logs, vérifiez qu'il existe bien une différence dans le programme lancé.

Exposer le port

- Ajoutons l'instruction `EXPOSE 5000` pour indiquer à Docker que cette app est censée être accédée via son port `5000` .
- NB : Publier le port grâce à l'option `-p port_de_l-hote:port_du_container` reste nécessaire, l'instruction `EXPOSE` n'est là qu'à titre de documentation de l'image.

Dockerfile amélioré

`Dockerfile` final :

L'instruction HEALTHCHECK

`HEALTHCHECK` permet de vérifier si l'app contenue dans un conteneur est en bonne santé.

- Dans un nouveau dossier ou répertoire, créez un fichier `Dockerfile` dont le contenu est le suivant :

```
FROM python:alpine

RUN apk add curl
RUN pip install flask==0.10.1

ADD /app.py /app/app.py
WORKDIR /app

HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit 1

CMD python app.py
```

- Créez aussi un fichier `app.py` avec ce contenu :

```
from flask import Flask

healthy = True

app = Flask(__name__)

@app.route('/health')
```

```python
def health():
    global healthy

    if healthy:
        return 'OK', 200
    else:
        return 'NOT OK', 500

@app.route('/kill')
def kill():
    global healthy
    healthy = False
    return 'You have killed your app.', 200


if __name__ == "__main__":
    app.run(host="0.0.0.0")
```

- Observez bien le code Python et la ligne `HEALTHCHECK` du `Dockerfile` puis lancez l'app. A l'aide de `docker ps` , relevez où Docker indique la santé de votre app.
- Visitez l'URL `/kill` de votre app dans un navigateur. Refaites `docker ps` . Que s'est-il passé ?
- *(Facultatif)* Rajoutez une instruction `HEALTHCHECK` au `Dockerfile` de notre app microblog.

Facultatif : Décortiquer une image

Une image est composée de plusieurs layers empilés entre eux par le Docker Engine et de métadonnées.

- Affichez la liste des images présentes dans votre Docker Engine.
- Inspectez la dernière image que vous venez de créez (`docker image --help` pour trouver la commande)
- Observez l'historique de construction de l'image avec `docker image history <image>`
- Visitons **en root** (`sudo su`) le dossier `/var/lib/docker/` sur l'hôte. En particulier, `image/overlay2/layerdb/sha256/` :
    - On y trouve une sorte de base de données de tous les layers d'images avec leurs ancêtres.
    - Il s'agit d'une arborescence.
- Vous pouvez aussi utiliser la commande `docker save votre_image -o image.tar` , et utiliser `tar -C image_decompressee/ -xvf image.tar` pour décompresser une image Docker puis explorer les différents layers de l'image.
- Pour explorer la hiérarchie des images vous pouvez installer `https://github.com/wagoodman/dive`

Facultatif : un Registry privé

- En récupérant la commande indiquée dans la doc officielle, créez votre propre registry.
- Puis trouvez comment y pousser une image dessus.
- Enfin, supprimez votre image en local et récupérez-la depuis votre registry.

Solution :

Facultatif : Faire parler la vache

Créons un nouveau Dockerfile qui permet de faire dire des choses à une vache grâce à la commande `cowsay`. Le but est de faire fonctionner notre programme dans un conteneur à partir de commandes de type :

- `docker run --rm cowsay Coucou !`
- `docker run --rm cowsay -f stegosaurus Yo!`
- `docker run --rm cowsay -f elephant-in-snake Un éléphant dans un boa.`
- Doit-on utiliser la commande `ENTRYPOINT` ou la commande `CMD` ? Se référer au manuel de référence sur les Dockerfiles si besoin.
- Pour information, `cowsay` s'installe dans `/usr/games/cowsay`.
- La liste des options (incontournables) de `cowsay` se trouve ici : https://debian-facile.org/doc:jeux:cowsay

Solution :

- L'instruction `ENTRYPOINT` et la gestion des entrées-sorties des programmes dans les Dockerfiles peut être un peu capricieuse et il faut parfois avoir de bonnes notions de Bash et de Linux pour comprendre (et bien lire la documentation Docker).
- On utilise parfois des conteneurs juste pour qu'ils s'exécutent une fois (pour récupérer le résultat dans la console, ou générer des fichiers). On utilise alors l'option `--rm` pour les supprimer dès qu'ils s'arrêtent.

Facultatif : Un multi-stage build

Transformez le `Dockerfile` de l'app `dnmonster` située à l'adresse suivante pour réaliser un multi-stage build afin d'obtenir l'image finale la plus légère possible : https://github.com/amouat/dnmonster/

La documentation pour les multi-stage builds est à cette adresse : https://docs.docker.com/develop/develop-images/multistage-build/

3 - Volumes et réseaux

Cycle de vie d'un conteneur

- Un conteneur a un cycle de vie très court: il doit pouvoir être créé et supprimé rapidement même en contexte de production.

Conséquences :

- On a besoin de mécanismes d'autoconfiguration, en particuler réseau car les IP des différents conteneur changent tout le temps.
- On ne peut pas garder les données persistantes dans le conteneur.

Solutions :

- Des réseaux dynamiques par défaut automatiques (DHCP mais surtout DNS automatiques)
- Des volumes (partagés ou non, distribués ou non) montés dans les conteneurs

Réseau

Gestion des ports réseaux (*port mapping*)

- L'instruction `EXPOSE` dans le Dockerfile informe Docker que le conteneur écoute sur les ports réseau au lancement. L'instruction `EXPOSE` **ne publie pas les ports**. C'est une sorte de **documentation entre la personne qui construit les images et la personne qui lance le conteneur à propos des ports que l'on souhaite publier**.
- Par défaut les conteneurs n'ouvrent donc pas de port même s'ils sont déclarés avec `EXPOSE` dans le Dockerfile.
- Pour publier un port au lancement d'un conteneur, c'est l'option `-p <port_host>:<port_guest>` de `docker run`.
- Instruction `port:` d'un compose file.

Bridge et overlay

- Un réseau bridge est une façon de créer un pont entre deux carte réseaux pour construire un réseau à partir de deux.
- Par défaut les réseaux docker fonctionne en bridge (le réseau de chaque conteneur est bridgé à un réseau virtuel docker)
- par défaut les adresses sont en 172.0.0.0/8, typiquement chaque hôte définit le bloc d'IP 172.17.0.0/16 configuré avec DHCP.

- Un réseau overlay est un réseau virtuel privé déployé par dessus un réseau existant (typiquement public). Pour par exemple faire un cloud multi-datacenters.

Le réseau Docker est très automatique

- Serveur DNS et DHCP intégré dans le "user-defined network" (c'est une solution IPAM)

- Donne un nom de domaine automatique à chaque conteneur.
- Mais ne pas avoir peur d'aller voir comment on perçoit le réseau de l'intérieur. Nécessaire pour bien contrôler le réseau.
- `ingress` : un loadbalancer automatiquement connecté aux nœuds d'un Swarm. Voir la doc sur les réseaux overlay.

Lier des conteneurs

- Aujourd'hui il faut utiliser un réseau dédié créé par l'utilisateur ("user-defined bridge network")
  - avec l'option `--network` de `docker run`
  - avec l'instruction `networks:` dans un docker composer
- On peut aussi créer un lien entre des conteneurs
  - avec l'option `--link` de `docker run`
  - avec l'instruction `link:` dans un docker composer
  - MAIS cette fonctionnalité est **obsolète** et déconseillée

Plugins réseaux

Il existe :

- les réseaux par défaut de Docker
- plusieurs autres solutions spécifiques de réseau disponibles pour des questions de performance et de sécurité
  - Ex. : **Weave Net** pour un cluster Docker Swarm
    - fournit une autoconfiguration très simple
    - de la sécurité
    - un DNS qui permet de simuler de la découverte de service
    - Du multicast UDP

Volumes

Les volumes Docker via la sous-commande `volume`

- `docker volume ls`
- `docker volume inspect`
- `docker volume prune`
- `docker volume create`
- `docker volume rm`

Bind mounting

Lorsqu'un répertoire hôte spécifique est utilisé dans un volume (la syntaxe `-v HOST_DIR:CONTAINER_DIR`), elle est souvent appelée **bind mounting** ("montage lié"). C'est quelque peu trompeur, car tous les volumes sont techniquement "bind mounted". La particularité, c'est que le point de montage sur l'hôte est explicite plutôt que caché dans un répertoire appartenant à Docker.

Exemple :

```
docker run -it -v /tmp/data:/data ubuntu /bin/bash

cd /data/
touch testfile
exit

ls /tmp/data/
```

L'instruction `VOLUME` dans un `Dockerfile`

L'instruction `VOLUME` dans un `Dockerfile` permet de désigner les volumes qui devront être créés lors du lancement du conteneur. On précise ensuite avec l'option `-v` de `docker run` à quoi connecter ces volumes. Si on ne le précise pas, Docker crée quand même un volume Docker au nom généré aléatoirement, un volume "caché".

Partager des données avec un volume

- Pour partager des données on peut monter le même volume dans plusieurs conteneurs.
- Pour lancer un conteneur avec les volumes d'un autre conteneur déjà montés on peut utiliser `--volumes-from <container>`
- On peut aussi créer le volume à l'avance et l'attacher après coup à un conteneur.
- Par défaut le driver de volume est `local` c'est-à-dire qu'un dossier est créé sur le disque de l'hôte.

```
docker volume create --driver local \
    --opt type=btrfs \
    --opt device=/dev/sda2 \
    monVolume
```

Plugins de volumes

On peut utiliser d'autres systèmes de stockage en installant de nouveau plugins de driver de volume. Par exemple, le plugin `vieux/sshfs` permet de piloter un volume distant via SSH.

Exemples:

- SSHFS (utilisation d'un dossier distant via SSH)
- NFS (protocole NFS)
- BeeGFS (système de fichier distribué générique)
- Amazon EBS (vendor specific)
- etc.

```
docker volume create -d vieux/sshfs -o sshcmd=<sshcmd> -o allow_other sshv
docker run -p 8080:8080 -v sshvolume:/path/to/folder --name test someimage
```

Ou via docker-compose :

```yaml
volumes:
  sshfsdata:
    driver: vieux/sshfs:latest
    driver_opts:
      sshcmd: "username@server:/location/on/the/server"
      allow_other: ""
```

Permissions

- Un volume est créé avec les permissions du dossier préexistant.

```dockerfile
FROM debian
RUN groupadd -r graphite && useradd -r -g graphite graphite
RUN mkdir -p /data/graphite && chown -R graphite:graphite /data/graphite
VOLUME /data/graphite
USER graphite
CMD ["echo", "Data container for graphite"]
```

Backups de volumes

- Pour effectuer un backup la méthode recommandée est d'utiliser un conteneur suplémentaire dédié
- qui accède au volume avec `--volume-from`
- qui est identique aux autres et donc normalement avec les mêmes UID/GID/permissions.
- permet de ne pas perdre bêtement le volume lors d'un `prune` car il reste un conteneur qui y est lié

TP 3 - Réseaux

Portainer

Si vous aviez déjà créé le conteneur Portainer, vous pouvez le relancer en faisant `docker start portainer` , sinon créez-le comme suit :

```
docker volume create portainer_data
docker run --detach --name portainer \
    -p 9000:9000 \
    -v portainer_data:/data \
    -v /var/run/docker.sock:/var/run/docker.sock \
    portainer/portainer-ce
```

Partie 1 : Docker networking

Pour expérimenter avec le réseau, nous allons lancer une petite application nodejs d'exemple (moby-counter) qui fonctionne avec une file (*queue*) redis (comme une base de données mais pour stocker des paires clé/valeur simples).

Récupérons les images depuis Docker Hub:

- `docker image pull redis:alpine`
- `docker image pull russmckendrick/moby-counter`
- Lancez la commande `ip a | tee /tmp/interfaces_avant.txt` pour lister vos interfaces réseau et les écrire dans le fichier

Pour connecter les deux applications créons un réseau manuellement:

- `docker network create moby-network`

Docker implémente ces réseaux virtuels en créant des interfaces. Lancez la commande `ip a | tee /tmp/interfaces_apres.txt` et comparez (`diff /tmp/interfaces_avant.txt /tmp/interfaces_apres.txt`). Qu'est-ce qui a changé ?

Maintenant, lançons les deux applications en utilisant notre réseau :

- `docker run -d --name redis --network <réseau> redis:alpine`
- `docker run -d --name moby-counter --network <réseau> -p 80:80 russmckendrick/moby-counter`
- Visitez la page de notre application. Qu'en pensez vous ? Moby est le nom de la mascotte Docker 🐳 😊. Faites un motif reconnaissable en cliquant.

Comment notre application se connecte-t-elle au conteneur redis ? Elle utilise ces instructions JS dans son fichier `server.js` :

```
var port = opts.redis_port || process.env.USE_REDIS_PORT || 6379;
var host = opts.redis_host || process.env.USE_REDIS_HOST || "redis";
```

En résumé par défaut, notre application se connecte sur l'hôte `redis` avec le port `6379`

Explorons un peu notre réseau Docker.

- Exécutez (`docker exec`) la commande `ping -c 3 redis` à l'intérieur de notre conteneur applicatif (`moby-counter` donc). Quelle est l'adresse IP affichée ?

```
docker exec moby-counter ping -c3 redis
```

- De même, affichez le contenu des fichiers `/etc/hosts` du conteneur (c'est la commande `cat` couplée avec `docker exec`). Nous constatons que Docker a automatiquement configuré l'IP externe **du conteneur dans lequel on est** avec l'identifiant du conteneur. De même, affichez `/etc/resolv.conf` : le résolveur DNS a été configuré par Docker. C'est comme ça que le conteneur connaît l'adresse IP de `redis` . Pour s'en assurer, interrogeons le serveur DNS de notre réseau `moby-network` en lançant la commande `nslookup redis 127.0.0.11` toujours grâce à `docker exec` : `docker exec moby-counter nslookup redis 127.0.0.11`
- Créez un deuxième réseau `moby-network2`
- Créez une deuxième instance de l'application dans ce réseau : `docker run -d --name moby-counter2 --network moby-network2 -p 9090:80 russmckendrick/moby-counter`
- Lorsque vous pingez `redis` depuis cette nouvelle instance `moby-counter2` , qu'obtenez-vous ? Pourquoi ?

Vous ne pouvez pas avoir deux conteneurs avec les mêmes noms, comme nous l'avons déjà découvert. Par contre, notre deuxième réseau fonctionne complètement isolé de notre premier réseau, ce qui signifie que nous pouvons toujours utiliser le nom de domaine `redis` . Pour ce faire, nous devons spécifier l'option `--network-alias` :

- Créons un deuxième redis avec le même domaine: `docker run -d --name redis2 --network moby-network2 --network-alias redis redis:alpine`
- Lorsque vous pingez `redis` depuis cette nouvelle instance de l'application, quelle IP obtenez-vous ?
- Récupérez comme auparavant l'adresse IP du nameserver local pour `moby-counter2` .
- Puis lancez `nslookup redis <nameserver_ip>` dans le conteneur `moby-counter2` pour tester la résolution de DNS.
- Vous pouvez retrouver la configuration du réseau et les conteneurs qui lui sont reliés avec `docker network inspect moby-network2` . Notez la section IPAM (IP Address Management).
- Arrêtons nos conteneurs : `docker stop moby-counter2 redis2` .
- Pour faire rapidement le ménage des conteneurs arrêtés lancez `docker container prune` .

- De même `docker network prune` permet de faire le ménage des réseaux qui ne sont plus utilisés par aucun conteneur.

TP 3bis - Volumes

Portainer

Si vous aviez déjà créé le conteneur Portainer, vous pouvez le relancer en faisant `docker start portainer` , sinon créez-le comme suit :

```
docker volume create portainer_data
docker run --detach --name portainer \
    -p 9000:9000 \
    -v portainer_data:/data \
    -v /var/run/docker.sock:/var/run/docker.sock \
    portainer/portainer-ce
```

Partie 2 : Volumes Docker

Introduction aux volumes

- Pour comprendre ce qu'est un volume, lançons un conteneur en mode interactif et associons-y le dossier `/tmp/data` de l'hôte au dossier `/data` sur le conteneur :

```
docker run -it -v /tmp/data:/data ubuntu /bin/bash
```

- Dans le conteneur, navigons dans ce dossier et créons-y un fichier :

```
cd /data/
touch testfile
```

- Sortons ensuite de ce conteneur avec la commande `exit`

```
exit
```

- Après être sorti·e du conteneur, listons le contenu du dossier **sur l'hôte** avec la commande suivante ou avec le navigateur de fichiers d'Ubuntu :

```
ls /tmp/data/
```

Le fichier `testfile` a été crée par le conteneur au dossier que l'on avait connecté grâce à `-v /tmp/data:/data`

L'app `moby-counter`, Redis et les volumes

Pour ne pas interférer avec la deuxième partie du TP :

- Stoppez tous les conteneurs redis et moby-counter avec `docker stop` ou avec Portainer.
- Supprimez les conteneurs arrêtés avec `docker container prune`
- Lancez `docker volume prune` pour faire le ménage de volume éventuellement créés dans les TPs précédent
- Lancez aussi `docker network prune` pour nettoyer les réseaux inutilisés

Passons à l'exploration des volumes:

- Recréez le réseau `moby-network` et les conteneurs `redis` et `moby-counter` à l'intérieur :

```
docker network create moby-network
docker run -d --name redis --network moby-network redis
docker run -d --name moby-counter --network moby-network -p 8000:80 russmcl
```

- Visitez votre application dans le navigateur. **Faites un motif reconnaissable en cliquant.**

Récupérer un volume d'un conteneur supprimé

- supprimez le conteneur `redis` : `docker stop redis` puis `docker rm redis`
- Visitez votre application dans le navigateur. Elle est maintenant déconnectée de son backend.
- Avons-nous vraiment perdu les données de notre conteneur précédent ? Non ! Le Dockerfile pour l'image officielle Redis ressemble à ça :

```
FROM alpine:3.5

RUN addgroup -S redis && adduser -S -G redis redis
RUN apk add --no-cache 'su-exec>=0.2'
ENV REDIS_VERSION 3.0.7
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-3.0.7.tar.g:
ENV REDIS_DOWNLOAD_SHA e56b4b7e033ae8dbf311f9191cf6fdf3ae974d1c
RUN set -x \
    && apk add --no-cache --virtual .build-deps \
        gcc \
        linux-headers \
```

```
        make \
        musl-dev \
        tar \
    && wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL" \
    && echo "$REDIS_DOWNLOAD_SHA *redis.tar.gz" | sha1sum -c - \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apk del .build-deps

RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data
COPY docker-entrypoint.sh /usr/local/bin/
RUN ln -s usr/local/bin/docker-entrypoint.sh /entrypoint.sh # backwards com
ENTRYPOINT ["docker-entrypoint.sh"]
EXPOSE 6379
CMD [ "redis-server" ]
```

Notez que, vers la fin du fichier, il y a une instruction `VOLUME` ; cela signifie que lorque notre conteneur a été lancé, un volume “caché” a effectivement été créé par Docker.

Beaucoup de conteneurs Docker sont des applications *stateful*, c'est-à-dire qui stockent des données. Automatiquement ces conteneurs créent des volument anonymes en arrière plan qu'il faut ensuite supprimer manuellement (avec rm ou prune).

- Inspectez la liste des volumes (par exemple avec Portainer) pour retrouver l'identifiant du volume caché. Normalement il devrait y avoir un volume `portainer_data` (si vous utilisez Portainer) et un volume anonyme avec un hash.
- Créez un nouveau conteneur redis en le rattachant au volume redis “caché” que vous avez retrouvé (en copiant l'id du volume anonyme) : `docker container run -d --name redis -v <volume_id>:/data --network moby-network redis:alpine`
- Visitez la page de l'application. Normalement un motif de logos *moby* d'une précédente session devrait s'afficher (après un délai pouvant aller jusqu'à plusieurs minutes)
- Affichez le contenu du volume avec la commande : `docker exec redis ls -lha /data`

Bind mounting

Finalement, nous allons recréer un conteneur avec un volume qui n'est pas anonyme.

En effet, la bonne façon de créer des volumes consiste à les créer manuellement (volumes nommés) : `docker volume create redis_data` .

- Supprimez l'ancien conteneur `redis` puis créez un nouveau conteneur attaché à ce volume nommé : `docker container run -d --name redis -v redis_data:/data --network moby-network redis:alpine`

Lorsqu'un répertoire hôte spécifique est utilisé dans un volume (la syntaxe `-v HOST_DIR:CONTAINER_DIR`), elle est souvent appelée **bind mounting**. C'est quelque peu trompeur, car tous les volumes sont techniquement "bind mounted". La différence, c'est que le point de montage est explicite plutôt que caché dans un répertoire géré par Docker.

- Lancez `docker volume inspect redis_data` .

Supprimer les volumes et réseaux

- Pour nettoyer tout ce travail, arrêtez d'abord les différents conteneurs `redis` et `moby-counter` .
- Lancez la fonction `prune` pour les conteneurs d'abord, puis pour les réseaux, et enfin pour les volumes.

Comme les réseaux et volumes n'étaient plus attachés à des conteneurs en fonctionnement, ils ont été supprimés.

Généralement, il faut faire beaucoup plus attention au prune de volumes (données à perdre) qu'au `prune` de conteneurs (rien à perdre car immutable et en général dans le registry).

Facultatif : utiliser `VOLUME` avec `microblog`

- Rendez-vous dans votre répertoire racine en tapant `cd` .
- Après être entré·e dans le repo `microblog` grâce à `cd microblog` , récupérez une version déjà dockerisée de l'app en chargeant le contenu de la branche Git `tp2-dockerfile` en faisant `git checkout tp2-dockerfile -- Dockerfile` .
- Si vous n'aviez pas encore le repo `microblog` :

```
git clone https://github.com/uptime-formation/microblog/
cd microblog
git checkout tp2-dockerfile
```

- Lire le `Dockerfile` de l'application `microblog` .

Un volume Docker apparaît comme un dossier à l'intérieur du conteneur. Nous allons faire apparaître le volume Docker comme un dossier à l'emplacement `/data` sur le conteneur.

- Pour que l'app Python soit au courant de l'emplacement de la base de données, ajoutez à votre `Dockerfile` une variable d'environnement `DATABASE_URL` ainsi (cette variable est lue par le programme Python) :

```
ENV DATABASE_URL=sqlite:////data/app.db
```

- Ajouter au `Dockerfile` une instruction `VOLUME` pour stocker la base de données SQLite de l'application.

Indice :
Solution :

- Créez un volume nommé appelé `microblog_db`, et lancez un conteneur l'utilisant, créez un compte et écrivez un message.
- Vérifier que le volume nommé est bien utilisé en branchant un deuxième conteneur `microblog` utilisant le même volume nommé.

Facultatif : Packagez votre propre app

Vous possédez tous les ingrédients pour packager l'app de votre choix désormais ! Récupérez une image de base, basez-vous sur un Dockerfile existant s'il vous inspire, et lancez-vous !

4 - Créer une application multiconteneur

Docker Compose

- Nous avons pu constater que lancer plusieurs conteneurs liés avec leur mapping réseau et les volumes liés implique des commandes assez lourdes. Cela devient ingérable si l'on a beaucoup d'applications microservice avec des réseaux et des volumes spécifiques.
- Pour faciliter tout cela et dans l'optique d'**Infrastructure as Code**, Docker introduit un outil nommé **docker-compose** qui permet de décrire de applications multiconteneurs grâce à des fichiers **YAML**.
- Pour bien comprendre qu'il ne s'agit que de convertir des options de commande Docker en YAML, un site vous permet de convertir une commande `docker run` en fichier Docker Compose : https://www.composerize.com/

A quoi ça ressemble, YAML ?

```yaml
- marché:
    lieu: Marché de la Défense
    jour: jeudi
    horaire:
      unité: "heure"
      min: 12
      max: 20
    fruits:
      - nom: pomme
        couleur: "verte"
        pesticide: avec
```

```yaml
    - nom: poires
      couleur: jaune
      pesticide: sans
  légumes:
    - courgettes
    - salade
    - potiron
```

Syntaxe

- Alignement ! (**2 espaces** !!)
- ALIGNEMENT !! (comme en python)
- **ALIGNEMENT !!!** (le défaut du YAML, pas de correcteur syntaxique automatique, c'est bête mais vous y perdrez forcément quelques heures !
- des listes (tirets)
- des paires **clé: valeur**
- Un peu comme du JSON, avec cette grosse différence que le JSON se fiche de l'alignement et met des accolades et des points-virgules
- **les extensions Docker et YAML dans VSCode vous aident à repérer des erreurs**

Un exemple de fichier Docker Compose

```yaml
version: 3

services:
  postgres:
    image: postgres:10
    environment:
      POSTGRES_USER: rails_user
      POSTGRES_PASSWORD: rails_password
      POSTGRES_DB: rails_db
    networks:
      - back_end

  redis:
    image: redis:3.2-alpine
    networks:
      - back_end

  rails:
    build: .
    depends_on:
      - postgres
      - redis
    environment:
      DATABASE_URL: "postgres://rails_user:rails_password@postgres:5432/ra
      REDIS_HOST: "redis:6379"
```

```
    networks:
      - front_end
      - back_end
    volumes:
      - .:/app

  nginx:
    image: nginx:latest
    networks:
      - front_end
    ports:
      - 3000:80
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf:ro

networks:
  front_end:
  back_end:
```

Un deuxième exemple :

```
version: "3.3"
services:

  mysql:
    container_name: mysqlpourwordpress
    environment:
      - MYSQL_ROOT_PASSWORD=motdepasseroot
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=monwordpress
    networks:
    - wordpress
    image: "mysql:5.7"

  wordpress:
    depends_on:
      - mysql
    container_name: wordpressavecmysql
    environment:
      - "WORDPRESS_DB_HOST=mysqlpourwordpress:3306"
      - WORDPRESS_DB_PASSWORD=monwordpress
      - WORDPRESS_DB_USER=wordpress
    networks:
    - wordpress
    ports:
      - "80:80"
    image: wordpress
    volumes:
      - wordpress_config:/var/www/html/

networks:
  wordpress:
```

```
volumes:
  wordpress_config:
```

Le workflow de Docker Compose

Les commandes suivantes sont couramment utilisées lorsque vous travaillez avec Compose. La plupart se passent d'explications et ont des équivalents Docker directs, mais il vaut la peine d'en être conscient·e :

- `up` démarre tous les conteneurs définis dans le fichier compose et agrège la sortie des logs. Normalement, vous voudrez utiliser l'argument `-d` pour exécuter Compose en arrière-plan.
- `build` reconstruit toutes les images créées à partir de Dockerfiles. La commande up ne construira pas une image à moins qu'elle n'existe pas, donc utilisez cette commande à chaque fois que vous avez besoin de mettre à jour une image (quand vous avez édité un Dockerfile). On peut aussi faire `docker-compose up --build`
- `ps` fournit des informations sur le statut des conteneurs gérés par Compose.
- `run` fait tourner un conteneur pour exécuter une commande unique. Cela aura aussi pour effet de faire tourner tout conteneur décrit dans `depends_on`, à moins que l'argument `--no-deps` ne soit donné.
- `logs` affiche les logs. De façon générale la sortie des logs est colorée et agrégée pour les conteneurs gérés par Compose.
- `stop` arrête les conteneurs sans les enlever.
- `rm` enlève les contenants à l'arrêt. N'oubliez pas d'utiliser l'argument `-v` pour supprimer tous les volumes gérés par Docker.
- `down` détruit tous les conteneurs définis dans le fichier Compose, ainsi que les réseaux

Le "langage" de Docker Compose

- N'hésitez pas à passer du temps à explorer les options et commandes de `docker-compose`.
- La documentation du langage (DSL) des compose-files est essentielle.
- Cette documentation indique aussi les différences entre les mots-clés supportés dans la version 2 et la version 3 des fichiers Docker Compose.
- il est aussi possible d'utiliser des variables d'environnement dans Docker Compose : se référer au mode d'emploi pour les subtilités de fonctionnement

Visualisation des applications microservice complexes

- Certaines applications microservice peuvent avoir potentiellement des dizaines de petits conteneurs spécialisés. Le service devient alors difficile à lire dans le compose file.

- Il est possible de visualiser l'architecture d'un fichier Docker Compose en utilisant docker-compose-viz
- Cet outil peut être utilisé dans un cadre d'intégration continue pour produire automatiquement la documentation pour une image en fonction du code.

TP 4 - Créer une application multiconteneur

Articuler deux images avec Docker compose

- Installez docker-compose avec `sudo apt install docker-compose` .

`identidock` : une application Flask qui se connecte à `redis`

- Démarrez un nouveau projet dans VSCode (créez un dossier appelé `identidock` et chargez-le avec la fonction *Add folder to workspace*)
- Dans un sous-dossier `app` , ajoutez une petite application python en créant ce fichier `identidock.py` :

```python
from flask import Flask, Response, request
import requests
import hashlib
import redis

app = Flask(__name__)
cache = redis.StrictRedis(host='redis', port=6379, db=0)
salt = "UNIQUE_SALT"
default_name = 'Joe Bloggs'

@app.route('/', methods=['GET', 'POST'])
def mainpage():

    name = default_name
    if request.method == 'POST':
        name = request.form['name']

    salted_name = salt + name
    name_hash = hashlib.sha256(salted_name.encode()).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
                Hello <input type="text" name="name" value="{0}">
                <input type="submit" value="submit">
                </form>
                <p>You look like a:
                <img src="/monster/{1}"/>
            '''.format(name, name_hash)
    footer = '</body></html>'
    return header + body + footer
```

```
@app.route('/monster/<name>')
def get_identicon(name):

    image = cache.get(name)

    if image is None:
        print ("Cache miss", flush=True)
        r = requests.get('http://dnmonster:8080/monster/' + name + '?size=8
        image = r.content
    cache.set(name, image)

    return Response(image, mimetype='image/png')

if __name__ == '__main__':
  app.run(debug=True, host='0.0.0.0', port=9090)
```

- `uWSGI` est un serveur python de production très adapté pour servir notre serveur intégré Flask, nous allons l'utiliser.
- Dockerisons maintenant cette nouvelle application avec le Dockerfile suivant :

```
FROM python:3.7

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask uWSGI requests redis
WORKDIR /app
COPY app/identidock.py /app

EXPOSE 9090 9191
USER uwsgi
CMD ["uwsgi", "--http", "0.0.0.0:9090", "--wsgi-file", "/app/identidock.py'
"--callable", "app", "--stats", "0.0.0.0:9191"]
```

- Observons le code du Dockerfile ensemble s'il n'est pas clair pour vous. Juste avant de lancer l'application, nous avons changé d'utilisateur avec l'instruction `USER` , pourquoi ?.
- Construire l'application, pour l'instant avec `docker build` , la lancer et vérifier avec `docker exec` , `whoami` et `id` l'utilisateur avec lequel tourne le conteneur.

Réponse :

Le fichier Docker Compose

- A la racine de notre projet `identidock` (à côté du Dockerfile), créez un fichier de déclaration de notre application appelé `docker-compose.yml` avec à l'intérieur :

```
version: "3.7"
services:
```

```yaml
  identidock:
    build: .
    ports:
      - "9090:9090"
```

- Plusieurs remarques :
  - la première ligne après `services` déclare le conteneur de notre application
  - les lignes suivantes permettent de décrire comment lancer notre conteneur
  - `build: .` indique que l'image d'origine de notre conteneur est le résultat de la construction d'une image à partir du répertoire courant (équivaut à `docker build -t identidock .`)
  - la ligne suivante décrit le mapping de ports entre l'extérieur du conteneur et l'intérieur.
- Lancez le service (pour le moment mono-conteneur) avec `docker-compose up` (cette commande sous-entend `docker-compose build`)
- Visitez la page web de l'app.
- Ajoutons maintenant un deuxième conteneur. Nous allons tirer parti d'une image déjà créée qui permet de récupérer une "identicon". Ajoutez à la suite du fichier Compose ***(attention aux indentations !)*** :

```yaml
dnmonster:
  image: amouat/dnmonster:1.0
```

Le `docker-compose.yml` doit pour l'instant ressembler à ça :

```yaml
version: "3.7"
services:
  identidock:
    build: .
    ports:
      - "9090:9090"

  dnmonster:
    image: amouat/dnmonster:1.0
```

Enfin, nous déclarons aussi un réseau appelé `identinet` pour y mettre les deux conteneurs de notre application.

- Il faut déclarer ce réseau à la fin du fichier (notez que l'on doit spécifier le driver réseau) :

```yaml
networks:
  identinet:
    driver: bridge
```

- Il faut aussi mettre nos deux services `identidock` et `dnmonster` sur le même réseau en ajoutant **deux fois** ce bout de code où c'est nécessaire ***(attention aux indentations !)*** :

```
networks:
  - identinet
```

- Ajoutons également un conteneur `redis` ***(attention aux indentations !)***. Cette base de données sert à mettre en cache les images et à ne pas les recalculer à chaque fois.

```
redis:
  image: redis
  networks:
    - identinet
```

`docker-compose.yml` final :

```
version: "3.7"
services:
  identidock:
    build: .
    ports:
      - "9090:9090"
    networks:
      - identinet

  dnmonster:
    image: amouat/dnmonster:1.0
    networks:
      - identinet

  redis:
    image: redis
    networks:
      - identinet

networks:
  identinet:
    driver: bridge
```

- Lancez l'application et vérifiez que le cache fonctionne en chercheant les `cache miss` dans les logs de l'application.
- N'hésitez pas à passer du temps à explorer les options et commandes de `docker-compose`, ainsi que la documentation officielle du langage des Compose files. Cette documentation indique aussi les différences entre la version 2 et la version 3 des fichiers Docker Compose.

D’autres services

Exercice de *google-fu* : un pad CodiMD

- Récupérez (et adaptez si besoin) à partir d’Internet un fichier `docker-compose.yml` permettant de lancer un pad CodiMD avec sa base de données. Je vous conseille de toujours chercher **dans la documentation officielle** ou le repository officiel (souvent sur Github) en premier. Attention, CodiMD avant s’appelait **HackMD**.
- Vérifiez que le pad est bien accessible sur le port donné.

Une stack Elastic

Centraliser les logs

L’utilité d’Elasticsearch est que, grâce à une configuration très simple de son module Filebeat, nous allons pouvoir centraliser les logs de tous nos conteneurs Docker. Pour ce faire, il suffit d’abord de télécharger une configuration de Filebeat prévue à cet effet :

```
curl -L -O https://raw.githubusercontent.com/elastic/beats/7.10/deploy/docl
```

Renommons cette configuration et rectifions qui possède ce fichier pour satisfaire une contrainte de sécurité de Filebeat :

```
mv filebeat.docker.yml filebeat.yml
sudo chown root filebeat.yml
```

Enfin, créons un fichier `docker-compose.yml` pour lancer une stack Elasticsearch :

```
version: "3"

services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.5.0
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    networks:
      - logging-network

  filebeat:
    image: docker.elastic.co/beats/filebeat:7.5.0
    user: root
    depends_on:
      - elasticsearch
    volumes:
      - ./filebeat.yml:/usr/share/filebeat/filebeat.yml:ro
      - /var/lib/docker/containers:/var/lib/docker/containers:ro
      - /var/run/docker.sock:/var/run/docker.sock:ro
```

```
    networks:
      - logging-network
    environment:
      - -strict.perms=false

  kibana:
    image: docker.elastic.co/kibana/kibana:7.5.0
    depends_on:
      - elasticsearch
    ports:
      - 5601:5601
    networks:
      - logging-network

networks:
  logging-network:
    driver: bridge
```

Il suffit ensuite de se rendre sur Kibana (port `5601`) et de configurer l'index en tapant `*` dans le champ indiqué, de valider et de sélectionner le champ `@timestamp`, puis de valider. L'index nécessaire à Kibana est créé, vous pouvez vous rendre dans la partie Discover à gauche (l'icône boussole 🧭) pour lire vos logs.

Facultatif : Utiliser Traefik

Vous pouvez désormais faire l'exercice 1 du TP7 pour configurer un serveur web qui permet d'accéder à vos services via des domaines.

5 - Orchestration et clustering

Orchestration

- Un des intérêts principaux de Docker et des conteneurs en général est de :
  - favoriser la modularité et les architectures microservice.
  - permettre la scalabilité (mise à l'échelle) des applications en multipliant les conteneurs.
- A partir d'une certaine échelle, il n'est plus question de gérer les serveurs et leurs conteneurs à la main.

Les nœuds d'un cluster sont les machines (serveurs physiques, machines virtuelles, etc.) qui font tourner vos applications (composées de conteneurs).

L'orchestration consiste à automatiser la création et la répartition des conteneurs à travers un cluster de serveurs. Cela peut permettre de :

- déployer de nouvelles versions d'une application progressivement.
- faire grandir la quantité d'instances de chaque application facilement.
- voire dans le cas de l'auto-scaling de faire grossir l'application automatiquement en fonction de la demande.

Docker Swarm

- Swarm est l'**outil de clustering et d'orchestration natif** de Docker (développé par Docker Inc.).
- Il s'intègre très bien avec les autres commandes docker (on a même pas l'impression de faire du clustering).
- Il permet de gérer de très grosses productions Docker.
- Swarm utilise l'API standard du Docker Engine (sur le port 2376) et sa propre API de management Swarm (sur le port 2377).
- Il a perdu un peu en popularité face à Kubernetes mais c'est très relatif (voir comparaison plus loin).

Architecture de Docker Swarm

Docker Enterprise Edition:
Swarm Architectur e

BYO TCP Load Balancer
Swarm Manager nodes
Quorum-based consensus gr oup (r aft)
node1 (leader)
Manager
Manager
Manager
• Usually 3, 5, or 7 managers, for HA and quorum status.
• Managers can also be work ers
node2
Worker
Worker
Worker
• Usually at least 1, but often many work ers.
• Workers cannot be managers (unless explicitly pr omoted)
Swarm W orker nodes
Our swarm so far...

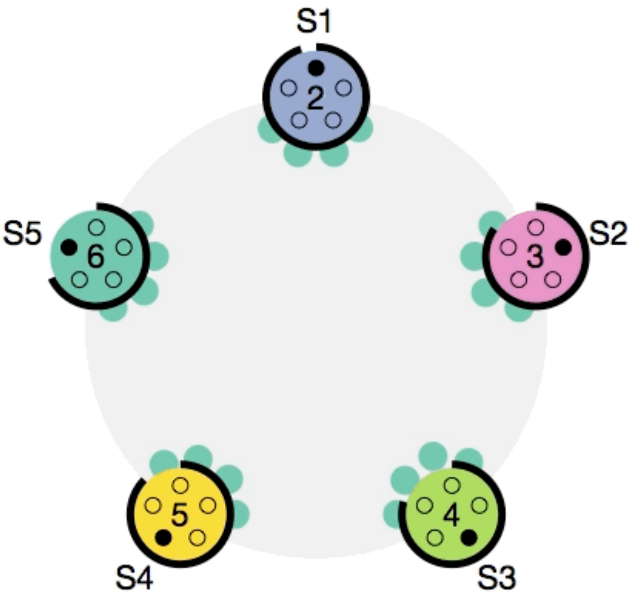
]

- Un ensemble de nœuds de contrôle pour gérer les conteneurs
- Un ensemble de nœuds worker pour faire tourner les conteneurs

- Les nœuds managers sont en fait aussi des workers et font tourner des conteneurs, c'est leur rôles qui varient.

Consensus entre managers Swarm

- L'algorithme Raft : http://thesecretlivesofdata.com/raft/



- Pas d'*intelligent balancing* dans Swarm
    - l’algorithme de choix est “spread”, c’est-à-dire qu’il répartit au maximum en remplissant tous les nœuds qui répondent aux contraintes données.

Docker Services et Stacks

- les **services** : la distribution **d’un seul conteneur en plusieurs exemplaires**
- les **stacks** : la distribution (en plusieurs exemplaires) **d’un ensemble de conteneurs (app multiconteneurs)** décrits dans un fichier Docker Compose

```
version: "3"
services:
  web:
    image: username/repo
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "4000:80"
    networks:
      - webnet
networks:
  webnet:
```

- Référence pour les options Swarm de Docker Compose : https://docs.docker.com/compose/compose-file/#deploy

- Le mot-clé `deploy` est lié à l'usage de Swarm
    - options intéressantes :
        - `update_config` : pour pouvoir rollback si l'update fail
        - `placement` : pouvoir choisir le nœud sur lequel sera déployé le service
        - `replicas` : nombre d'exemplaires du conteneur
        - `resources` : contraintes d'utilisation de CPU ou de RAM sur le nœud

Sous-commandes Swarm

- `swarm init` : Activer Swarm et devenir manager d'un cluster d'un seul nœud
- `swarm join` : Rejoindre un cluster Swarm en tant que nœud manager ou worker
- `service create` : Créer un service (= un conteneur en plusieurs exemplaires)
- `service inspect` : Infos sur un service
- `service ls` : Liste des services
- `service rm` : Supprimer un service
- `service scale` : Modifier le nombre de conteneurs qui fournissent un service
- `service ps` : Liste et état des conteneurs qui fournissent un service
- `service update` : Modifier la définition d'un service
- `docker stack deploy` : Déploie une stack (= fichier Docker compose) ou update une stack existante
- `docker stack ls` : Liste les stacks
- `docker stack ps` : Liste l'état du déploiement d'une stack
- `docker stack rm` : Supprimer une ou des stacks
- `docker stack services` : Liste les services qui composent une stack
- `docker node inspect` : Informations détaillées sur un nœud
- `docker node ls` : Liste les nœuds
- `docker node ps` : Liste les tâches en cours sur un nœud
- `docker node promote` : Transforme un nœud worker en manager
- `docker node demote` : Transforme un nœud manager en worker

Répartition de charge (load balancing)

- Un load balancer : une sorte d'**"aiguillage" de trafic réseau**, typiquement HTTP(S) ou TCP.
- Un aiguillage **intelligent** qui se renseigne sur plusieurs critères avant de choisir la direction.
- Cas d'usage :
    - Éviter la surcharge : les requêtes sont réparties sur différents backends pour éviter de les saturer.
- Haute disponibilité : on veut que notre service soit toujours disponible, même en cas

de panne (partielle) ou de maintenance.

- Donc on va dupliquer chaque partie de notre service et mettre les différentes instances derrière un load balancer.
- Le load balancer va vérifier pour chaque backend s'il est disponible (**healthcheck**) avant de rediriger le trafic.
- Répartition géographique : en fonction de la provenance des requêtes on va rediriger vers un datacenter adapté (+ ou - proche)

Le loadbalancing de Swarm est automatique

- Loadbalancer intégré : Ingress
- Permet de router automatiquement le trafic d'un service vers les nœuds qui l'hébergent et sont disponibles.
- Pour héberger une production il suffit de rajouter un loadbalancer externe qui pointe vers un certain nombre de nœuds du cluster et le trafic sera routé automatiquement à partir de l'un des nœuds.

Solutions de loadbalancing externe

- **HAProxy** : Le plus répandu en loadbalancing
- **Træfik** : Simple à configurer et fait pour l'écosystème Docker
- **NGINX** : Serveur web générique mais a depuis quelques années des fonctions puissantes de loadbalancing et de TCP forwarding.

Gérer les données sensibles dans Swarm avec les secrets Docker

- `echo "This is a secret" | docker secret create my_secret_data`
- `docker service create --name monservice --secret my_secret_data redis:alpine` => monte le contenu secret dans `/var/run/my_secret_data`

Docker Machine

- C'est l'outil de gestion d'hôtes Docker
- Il est capable de créer des serveurs Docker "à la volée"

- Concrètement, `docker-machine` permet de **créer automatiquement des machines** avec le **Docker Engine** et **ssh** configuré et de gérer les **certificats TLS** pour se connecter à l'API Docker des différents serveurs.
- Il permet également de changer le contexte de la ligne de commande Docker pour basculer sur l'un ou l'autre serveur avec les variables d'environnement adéquates.

- Il permet également de se connecter à une machine en ssh en une simple commande.

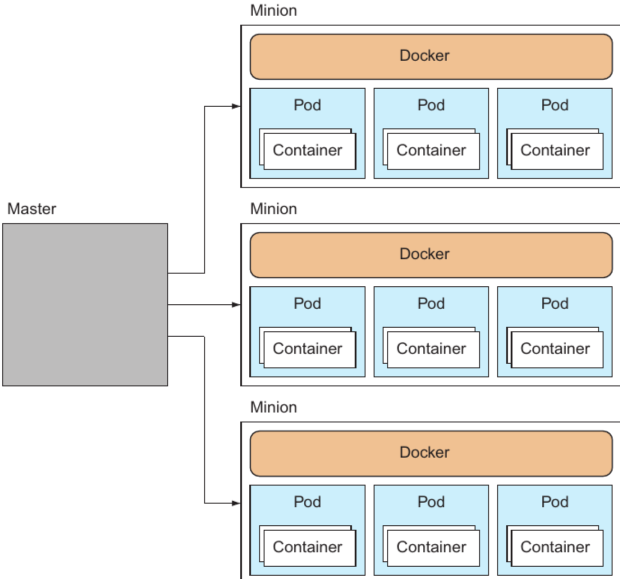
Exemple :

```
docker-machine create  --driver digitalocean \
     --digitalocean-ssh-key-fingerprint 41:d9:ad:ba:e0:32:73:58:4f:09:28:1
     --digitalocean-access-token "a94008870c9745febbb2bb84b01d16b6bf837b4
     hote-digitalocean
```

Pour basculer `eval $(docker env hote-digitalocean);`

- `docker run -d nginx:latest` créé ensuite un conteneur **sur le droplet digitalocean** précédemment créé.
- `docker ps -a` affiche le conteneur en train de tourner à distance.
- `wget $(docker-machine ip hote-digitalocean)` va récupérer la page nginx.

Présentation de Kubernetes



- Les **pods** Kubernetes servent à grouper des conteneurs en unités d'application (microservices ou non) fortement couplées (un peu comme les *stacks* Swarm)
- Les **services** sont des groupes de pods exposés à l'extérieur
-
- Les **deployments** sont une abstraction pour scaler ou mettre à jours des groupes de **pods** (un peu comme les *tasks* dans Swarm).

Présentation de Kubernetes

- Une autre solution très à la mode depuis 4 ans. Un buzz word du DevOps en France :)

- Une solution **robuste**, **structurante** et **open source** d'orchestration Docker.
- Au cœur du consortium **Cloud Native Computing Foundation** très influent dans le monde de l'informatique.
- Hébergeable de façon identique dans le cloud, on-premise ou en mixte.
- Kubernetes a un flat network (un overlay de plus bas niveau que Swarm) : https://neuvector.com/network-security/kubernetes-networking/

Comparaison Swarm et Kubernetes

- Swarm plus intégré avec la CLI et le workflow Docker.
- Swarm est plus fluide, moins structurant mais moins automatique que Kubernetes.
- Swarm groupe les containers entre eux par **stack**.
- Kubernetes au contraire crée des **pods** avec une meilleure isolation.
    - Kubernetes a une meilleure fault tolerance que Swarm
    - attention au contre-sens : un service Swarm est un seul conteneur répliqué, un service Kubernetes est un groupe de conteneurs (pod) répliqué, plus proche des Docker Stacks.

Comparaison Swarm et Kubernetes

- Kubernetes a plus d'outils intégrés. Il s'agit plus d'un écosystème qui couvre un large panel de cas d'usage.
- Swarm est beaucoup plus simple à mettre en œuvre qu'une stack Kubernetes.
- Swarm serait donc mieux pour les clusters moyen et Kubernetes pour les très gros

TP 5 - Orchestration et clustering

Introduction à Swarm

Initialisez Swarm avec `docker swarm init` .

Créer un service

A l'aide de `docker service create` , créer un service à partir de l'image `traefik/whoami` accessible sur le port `9999` et connecté au port `80` et avec 5 répliques.

Solution :

Accédez à votre service et actualisez plusieurs fois la page. Les informations affichées changent. Pourquoi ?

- Lancez une commande `service scale` pour changer le nombre de *replicas* de votre service et observez le changement avec `docker service ps hello`

La stack `example-voting-app`

- Cloner l'application `example-voting-app` ici : https://github.com/dockersamples/example-voting-app
- Lire le schéma d'architecture de l'app `example-voting-app` sur Github. A noter que le service `worker` existe en deux versions utilisant un langage de programmation différent (Java ou .NET), et que tous les services possèdent des images pour conteneurs Windows et pour conteneurs Linux. Ces versions peuvent être déployées de manière interchangeable et ne modifient pas le fonctionnement de l'application multi-conteneur. C'est une démonstration de l'utilité du paradigme de la conteneurisation et de l'architecture dite *"micro-service"*.
- Lire attentivement les fichiers `docker-compose.yml`, `docker-compose-simple.yml`, `docker-stack-simple.yml` et `docker-stack.yml`. Ce sont tous des fichiers Docker Compose classiques avec différentes options liées à un déploiement via Swarm. Quelles options semblent spécifiques à Docker Swarm ? Ces options permettent de configurer des fonctionnalités d'**orchestration**.
- Dessiner rapidement le schéma d'architecture associé au fichier `docker-compose-simple.yml`, puis celui associé à `docker-stack.yml` en indiquant bien à quel réseau quel service appartient.

- Avec `docker swarm init`, transformer son installation Docker en une installation Docker compatible avec Swarm. Lisez attentivement le message qui vous est renvoyé.
- Déployer la stack du fichier `docker-stack.yml` : `docker stack deploy --compose-file docker-stack.yml vote`
- `docker stack ls` indique 6 services pour la stack `vote`. Observer également l'output de `docker stack ps vote` et de `docker stack services vote`. Qu'est-ce qu'un service dans la terminologie de Swarm ?
- Accéder aux différents front-ends de la stack grâce aux informations contenues dans les commandes précédentes. Sur le front-end lié au vote, actualiser plusieurs fois la page. Que signifie la ligne `Processed by container ID […]` ? Pourquoi varie-t-elle ?
- Scaler la stack en ajoutant des *replicas* du front-end lié au vote avec l'aide de `docker service --help`. Accédez à ce front-end et vérifier que cela a bien fonctionné en actualisant plusieurs fois.

Clustering entre ami·es

Avec un service

- Se grouper par 2 ou 3 pour créer un cluster à partir de vos VM respectives (il faut

utiliser une commande Swarm pour récupérer les instructions nécessaires).

- Si grouper plusieurs des VM n'est pas possible, vous pouvez créer un cluster multi-nodes très simplement avec l'interface du site Play With Docker, il faut s'y connecter avec vos identifiants Docker Hub.
- Vous pouvez faire `docker swarm --help` pour obtenir des infos manquantes, ou faire `docker swarm leave --force` pour réinitialiser votre configuration Docker Swarm si besoin.
- N'hésitez pas à regarder dans les logs avec `systemctl status docker` comment se passe l'élection du nœud *leader*, à partir du moment où vous avez plus d'un manager.
- Lancez le service suivant : `docker service create --name whoami --replicas 5 --publish published=80,target=80 traefik/whoami`
- Accédez au service depuis un node, et depuis l'autre. Actualisez plusieurs fois la page. Les informations affichées changent. Lesquelles, et pourquoi ?

Avec la stack `example-voting-app`

- Si besoin, cloner de nouveau le dépôt de l'application `example-voting-app` avec `git clone https://github.com/dockersamples/example-voting-app` puis déployez la stack de votre choix.
- Ajouter dans le Compose file des instructions pour scaler différemment deux services (3 *replicas* pour le service *front* par exemple). N'oubliez pas de redéployer votre Compose file.
- puis spécifier quelques options d'orchestration exclusives à Docker Swarm : que fait `mode: global` ? N'oubliez pas de redéployer votre Compose file.
- Avec Portainer ou avec docker-swarm-visualizer, explorer le cluster ainsi créé (le fichier `docker-stack.yml` de l'app `example-voting-app` contient déjà un exemplaire de `docker-swarm-visualizer`).
- Trouver la commande pour déchoir et promouvoir l'un de vos nœuds de `manager` à `worker` et vice-versa.
- Puis sortir un nœud du cluster (`drain`) : `docker node update --availability drain <node-name>`

Facultatif : débugger la config Docker de `example-voting-app`

Vous avez remarqué ? Nous avons déployé une super stack d'application de vote avec succès mais, si vous testez le vote, vous verrez que ça ne marche pas, il n'est pas comptabilisé. Outre le fait que c'est un plaidoyer vivant contre le vote électronique, vous pourriez tenter de débugger ça maintenant (c'est plutôt facile).

Indice 1 :
Indice 2 :
Indice 3 :
Solution / explications :

Introduction à Kubernetes

Le fichier `kube-deployment.yml` de l'app `example-voting-app` décrit la même app pour un déploiement dans Kubernetes plutôt que dans Docker Compose ou Docker Swarm. Tentez de retrouver quelques équivalences entre Docker Compose / Swarm et Kubernetes en lisant attentivement ce fichier qui décrit un déploiement Kubernetes.

Facultatif : Utiliser Traefik avec Swarm

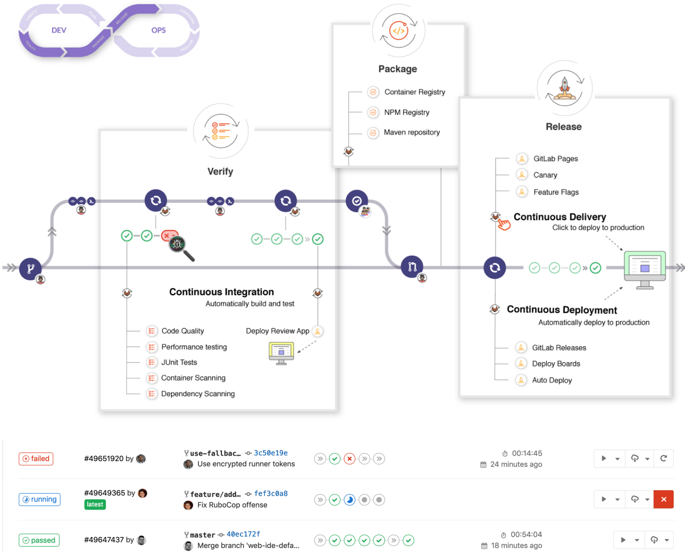
Vous pouvez désormais faire l'exercice 2 du TP 7 pour configurer un serveur web qui permet d'accéder à vos services Swarm via des domaines spécifiques.

Conclusion

Conclusions sur l'écosystème Docker

Configurer de la CI/CD

- La nature facile à déployer des conteneurs et l'intégration du principe d'Infrastructure-as-Code les rend indispensable dans de la CI/CD (intégration continue et déploiement continu).
- Les principaux outils de CI sont Gitlab, Jenkins, Github Actions, Travis CI…
  - Gitlab propose par défaut des runners préconfigurés qui utilisent des conteneurs Docker et tournent en général dans un cluster Kubernetes.
  - Gitlab propose aussi un registry d'images Docker, privé ou public, par projet.
- Les tests à l'intérieur des conteneurs peuvent aussi être faits de façon plus poussée, avec par exemple Ansible comme source de healthcheck ou comme suite pour les tests.
- Dans une autre catégorie, Gitpod base son workflow sur des images Docker permettant de configurer un environnement de développement

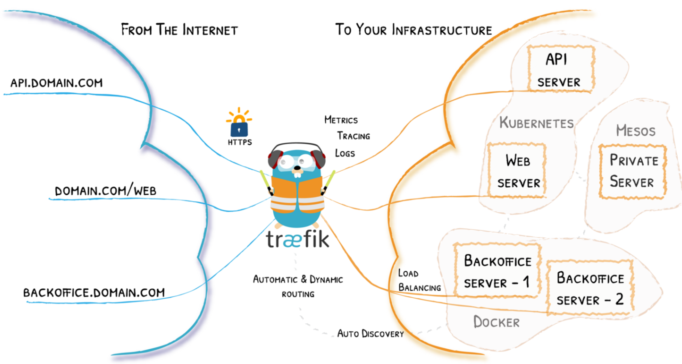


Gérer les logs des conteneurs

Avec Elasticsearch, Filebeat et Kibana... grâce aux labels sur les conteneurs Docker

Gérer le reverse proxy

Avec Traefik, aussi grâce aux labels sur les conteneurs Docker



Monitorer des conteneurs

- Avec Prometheus pour Docker et Docker Swarm
- Ou bien Netdata, un peu plus joli et configuré pour monitorer des conteneurs *out-of-the-box*

Tests sur des conteneurs

Ansible comme source de healthcheck

Bonnes pratiques et outils

Sécurité / durcissement

- **un conteneur privilégié est *root* sur la machine !**
- des *cgroups* correct : `ulimit -a`
- par défaut les *user namespaces* ne sont pas utilisés !
  - exemple de durcissement conseillé : https://medium.com/@mccode/processes-in-containers-should-not-run-as-root-2feae3f0df3b
- le benchmark Docker CIS : https://github.com/docker/docker-bench-security/
- La sécurité de Docker c'est aussi celle de la chaîne de dépendance, des images, des packages installés dans celles-ci : on fait confiance à trop de petites briques dont on ne vérifie pas la provenance ou la mise à jour
  - Clair : l'analyse statique d'images Docker
- docker-socket-proxy : protéger la *socket* Docker quand on a besoin de la partager à des conteneurs comme Traefik ou Portainer

Limites de Docker

Stateful

- les conteneurs stateless c'est bien beau mais avec une base de données, ça ne se gère pas magiquement du tout
  - quelques ressources sur le stateful avec Docker : https://container.training/swarm-selfpaced.yml.html#450

Configurer le réseau de façon plus complexe avec des plugins réseau

- Réseaux "overlay": IP in IP, VXLAN…
- …mais on a rapidement besoin de plugins exclusifs à Kubernetes : Calico, Flannel, Canal (Calico + Flannel), Cilium (qui utilise eBPF)

Volumes distribués

- problème des volumes partagés / répliqués
  - domaine à part entière
  - **Solution 1** : solutions applicatives robustes
    - pour MySQL/MariaDB : Galera
    - pour Postgres : on peut citer Citus ou pgpool, voir la comparaison de différentes solutions
    - Elasticsearch est distribué *out-of-the-box*
  - **Solution 2** : volume drivers avec Docker
    - Flocker, Convoy, visent à intégrer une technologie de réplication
    - c'est un moyen, pas une solution : reste un outil pour configurer ce que l'on souhaite

Aller plus loin

- Le livre *Mastering Docker*, de Russ McKendrick et Scott Gallagher
- les ressources présentes dans la bibliographie
- la liste de Awesome Docker

Dockercraft : administrez vos containers dans Minecraft

Retours

- Comment ça s'est passé ?
  - Difficulté : trop facile ? trop dur ? quoi en particulier ?
  - Vitesse : trop rapide ? trop lent ? lors de quoi en particulier ?
  - Attentes sur le contenu ? Les manipulations ?
  - Questions restées ouvertes ? Nouvelles questions ?
  - Envie d'utiliser Docker ? ou de le jeter à la poubelle ?

TP 6 (bonus) - Intégration continue avec Gitlab

Créer une pipeline de build d'image Docker avec les outils CI/CD Gitlab

1. Si vous n'en avez pas déjà un, créez un compte sur Gitlab.com : https://gitlab.com/users/sign_in#register-pane
2. Créez un nouveau projet et avec Git, le Web IDE Gitlab, ou bien en forkant une app existante depuis l'interface Gitlab, poussez-y l'app de votre choix (par exemple `microblog`, `dnmonster` ou l'app `healthcheck` vue au TP2).
3. Ajoutez un Dockerfile à votre repository ou vérifiez qu'il en existe bien un.
4. Créez un fichier `.gitlab-ci.yml` depuis l'interface web de Gitlab et choisissez "Docker" comme template. Observons-le ensemble attentivement.
5. Faites un commit de ce fichier.

6. Vérifiez votre CI : il faut vérifier sur le portail de Gitlab comment s'est exécutée la pipeline.
7. Vérifiez dans la section Container Registry que votre image a bien été push.

Ressources

- La section Quick Start de la documentation Gitlab-CI
- Vous pouvez trouver des exemples de CI dans la documentation Gitlab.
- La section dédiée à `docker build` de la documentation Gitlab
- La section de la documentation dédiée au Container Registry

Avec BitBucket

BitBucket propose aussi son outil de pipeline, à la différence qu'il n'a pas de registry intégré, le template par défaut propose donc de pousser son image sur le registry Docker Hub.

- Il suffit de créer un repo BitBucket puis d'y ajouter le template de CI Docker proposé (le template est caché derrière un bouton *See more*).
- Ensuite, il faut ajouter des *Repository variables* avec ses identifiants Docker Hub. Dans le template, ce sont les variables `DOCKERHUB_USERNAME` , `DOCKERHUB_PASSWORD` et `DOCKERHUB_NAMESPACE` (identique à l'username ici).

Ressources

- https://support.atlassian.com/bitbucket-cloud/docs/run-docker-commands-in-bitbucket-pipelines/

Conclusion

Déployer notre container ou notre projet Docker Compose

Nous avons fait la partie CI (intégration continue). Une étape supplémentaire est nécessaire pour ajouter le déploiement continu de l'app (CD) : si aucune étape précédente n'a échoué, la nouvelle version de l'app devra être déployée sur votre serveur, via une connexion SSH et `rsync` par exemple. Il faudra ajouter des variables secrètes au projet (clé SSH privée par exemple), cela se fait dans les options de Gitlab ou de BitBucket.

TP 7 (bonus) - Docker et les reverse proxies

Exercice 1 - Utiliser Traefik pour le routage

Traefik est un reverse proxy très bien intégré à Docker. Il permet de configurer un routage entre un point d'entrée (ports `80` et `443` de l'hôte) et des containers Docker, grâce aux informations du daemon Docker et aux `labels` sur chaque containers. Nous allons nous baser sur le guide d'introduction Traefik - Getting started.

- Avec l'aide de la documentation Traefik, ajoutez une section pour le reverse proxy Traefik pour dans un fichier Docker Compose de votre choix.

Solution :

- Explorez le dashboard Traefik accessible sur le port indiqué dans le fichier Docker Compose.
- Ajouter des labels à l'app web que vous souhaitez desservir grâce à Traefik à partir de l'exemple de la doc Traefik, grâce aux labels ajoutés dans le `docker-compose.yml` (attention à l'indentation).

  Solution :
- Avec l'aide de la documentation Traefik sur Let's Encrypt et Docker Compose, configurez Traefik pour qu'il crée un certificat Let's Encrypt pour votre container.
- Si vous avez une IP publique mais pas de domaine, vous pouvez utiliser le service gratuit [netlib.re] qui vous fournira un domaine en `*.netlib.re` .
- Vous aurez aussi besoin de configurer des DNS via `netlib.re` si vous voulez vérifier des sous-domaines (et non votre domaine principal) auprès de Let's Encrypt (de plus, si vous voulez un certificat avec *wildcard* pour tous vos sous-domaines, il faudra résoudre le `dnsChallenge` de Let's Encrypt de manière manuelle).

Solution :

Exercice 2 - Swarm avec Traefik

- Avec l'aide de la documentation Traefik sur Docker Swarm, configurez Traefik avec Swarm.

Solution :

QCM Docker

Entourez la bonne réponse

Question 1

Quelle est la principale différence entre une machine virtuelle (VM) et un conteneur ?

1. Un conteneur est une boîte qui contient un logiciel Windows alors qu'une VM fonctionne généralement sous Linux.
2. Un conteneur permet de faire des applications distribuées dans le cloud

contrairement aux machines virtuelles.
3. Un conteneur partage le noyau du système hôte alors qu'une machine virtuelle virtualise son propre noyau indépendant.

Question 2

En quoi Docker permet de faire de l'*Infrastructure as Code* ?

1. Comme Ansible, Docker se connecte en SSH à un Linux pour décrire des configurations.
2. Docker permet avec les Dockerfiles et les fichiers Compose de décrire l'installation d'un logiciel et sa configuration.

Question 3

Quels sont les principaux atouts de Docker ?

1. Il permet de rendre compatible tous les logiciels avec le cloud (AWS, etc.) et facilite l'IoT.
2. Il utilise le langage Go qui est de plus en plus populaire et accélère les logiciels qui l'utilise.
3. Il permet d'uniformiser les déploiements logiciels et facilite la construction d'application distribuées.

Question 4

Pour créer un conteneur Docker à partir du code d'un logiciel il faut d'abord :

1. Écrire un Dockerfile qui explique comment empaqueter le code puis construire l'image Docker avec docker build.
2. Créer un cluster avec docker-machine puis compiler le logiciel avec Docker Stack.

Question 5

Un volume Docker est :

1. Un espace de stockage connecté à un ou plusieurs conteneurs docker.
2. Une image fonctionnelle à partir de laquelle on crée des conteneurs identiques.
3. Un snapshot de l'application que l'on déploie dans un cluster comme Swarm.

Question 6

Indiquez la ou les affirmations vraies :

Comment configurer de préférence un conteneur à sa création (lancement avec `docker run`) ?

1. Reconstruire l'image à chaque fois à partir du Dockerfile avant.

2. Utiliser des variables d'environnement pour définir les paramètres à la volée.
3. Faire `docker exec` puis aller modifier les fichiers de configuration à l'intérieur
4. Associer le conteneur à un volume qui rassemble des fichiers de configuration

Question 7

Un *Compose file* ou fichier Compose permet :

1. D'installer Docker facilement sur des VPS et de contrôler un cluster.
2. D'alléger les images et de détecter les failles de sécurité dans le packaging d'une application.
3. De décrire une application multiconteneurs, sa configuration réseau et son stockage.

Question 8

Indiquez la ou les affirmations vraies :

La philosophie de Docker est basée sur :

1. L'immutabilité, c'est-à-dire le fait de jeter et recréer un conteneur pour le changer plutôt que d'aller modifier l'intérieur.
2. Le cloud, c'est-à-dire la vente de plateforme et de logiciel "as a service".
3. L'infrastructure-as-code, c'est-à-dire la description d'un état souhaité de l'infrastructure hébergeant application

Question 9

Indiquez la ou les affirmations vraies :

1. Docker est très pratique pour distribuer un logiciel mais tous les conteneurs doivent obligatoirement être exposés à Internet.
2. Docker utilise un cloud pour distribuer facilement des logiciels dans de nombreuses versions.
3. Docker est une catastrophe en terme de sécurité car les conteneurs sont peu isolés.

Question 10

Docker Swarm est :

1. Un cloud où pousser et récupérer les images Docker de la terre entière.
2. Une solution de clustering et d'orchestration intégrée directement avec Docker.
3. Un logiciel qui complète ce qu'offre Kubernetes en y ajoutant des fonctionnalités manquantes

Kubernetes

Module 3

Kubernetes

Administrer des applications multiconteneurs complexes

- Cours 1 - Présentation de Kubernetes
- Cours 2 - Mettre en place un cluster Kubernetes
- TP1 - Installation et configuration de Kubernetes
- Cours 3 - Concepts de Kubernetes
- Cours 4 - Objets Kubernetes - Partie 1
- TP 2 - Déployer Wordpress rapidement
- Cours 5 - Le réseau dans Kubernetes
- TP 3 - Déployer des conteneurs de A à Z
- Rappels Docker
- Cours 6 - Objets Kubernetes - Partie 2
- Cours 7 - Helm, le gestionnaire de paquets Kubernetes
- TP 4 - Déployer Wordpress avec Helm
- TP 5 - Cloud Azure
- TD opt. - StatefulSets et bases de données
- TP opt. - Les ingresses
- TP opt. - Le RBAC
- Conclusion
- TP7 - Stratégies de déploiement et monitoring

Cours 1 - Présentation de Kubernetes

Kubernetes est la solution dominante d'orchestration de conteneurs développée en Open Source au sein de la Cloud Native Computing Foundation.

Historique et popularité

Kubernetes est un orchestrateur développé originellement par Google et basé sur une dizaine d'années d'expérience de déploiement d'application énormes.

La première version est sortie en 2014 et K8S est devenu depuis l'un des projets open source les plus populaires du monde.

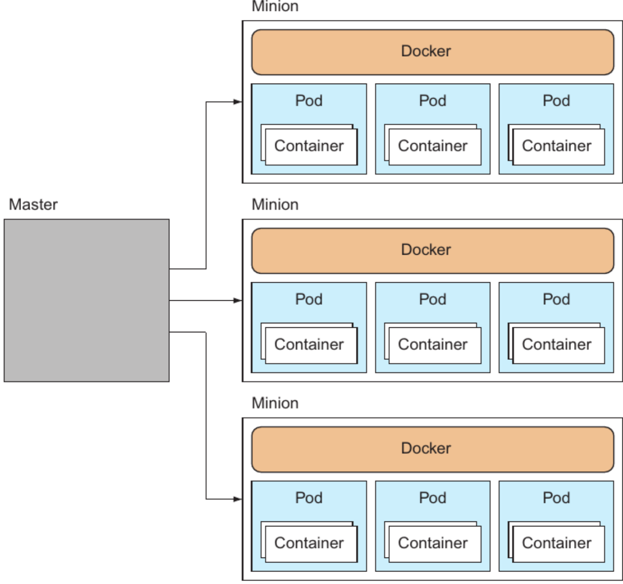
Autour de ce projet s'est développée la **Cloud Native Computing Foundation** qui comprend : Google, CoreOS, Mesosphere, Red Hat, Twitter, Huawei, Intel, Cisco, IBM, Docker, Univa et VMware.

Il s'agit d'une solution **robuste**, **structurante** et **open source** qui se construit autour des objectifs de:

- Rapidité
- Scaling des logiciels et des équipes de développement
- Abstraction et uniformisation des infrastructures

Architecture de Kubernetes

Les ressources k8s en (très) bref



- Kubernetes a une architecture master/worker (cf. cours 2) composés d'un *control plane* et de nœuds **workers**.

- Les **pods** Kubernetes servent à grouper des conteneurs fortement couplés en unités d'application
- Les **deployments** sont une abstraction pour **créer ou mettre à jour** (ex : scaler) des groupes de **pods**.
- Enfin, les **services** sont des groupes de pods (des deployments) exposés à l'intérieur ou à l'extérieur du cluster.

Points forts de Kubernetes

- Open source et très actif.
- Une communauté très visible et présente dans l'évolution de l'informatique.
- Un standard collectif qui permet une certaine interopérabilité dans le cloud.
- Les *pods* tendent à se rapprocher plus d'une VM du point de vue de l'application (ressources de calcul garanties, une ip joignable sur le réseau local, multiprocess).
- Hébergeable de façon quasi-identique dans le cloud, on-premise ou en mixte.
- Kubernetes a un *flat network* ce qui permet de faire des choses puissante facilement comme le multi-datacenter.
- K8s est pensé pour la *scalabilité* et le *calcul distribué*.

Faiblesses de Kubernetes

- Une difficulté à manier tout ce qui est *stateful*, comme des bases de données
    - …même si les Operators et les CRD (Custom Resources Definitions) permettent de combler cette lacune dans la logique *stateless* de k8s
- Beaucoup de points sont laissés à la décision du fournisseur de cloud ou des admins système :
    - Pas de solution de **stockage** par défaut, et parfois difficile de stocker "simplement" sans passer par les fournisseurs de cloud, ou par une solution de stockage décentralisé à part entière (**Ceph, Gluster, Longhorn**…)
        - …même si ces solutions sont souvent bien intégrées à k8s
    - Beaucoup de solutions de **réseau** qui se concurrencent, demandant un comparatif fastidieux
        - …même si plusieurs leaders émergent comme **Calico, Flannel, Weave ou Cilium**
    - Pas de solution de loadbalancing par défaut : soit on se base sur le fournisseur de cloud, soit on configure *MetalLB*

L'écosystème Kubernetes

Kubernetes n'est qu'un ensemble de standards. Il existe beaucoup de variétés (*flavours*) de Kubernetes, implémentant concrètement les solutions techniques derrière tout ce que Kubernetes ne fait que définir : solutions réseau, stockage (distribué ou non),

loadbalancing, service de reverse proxy (Ingress), autoscaling de cluster (ajout de nouvelles VM au cluster automatiquement), monitoring…

Il est très possible de monter un cluster Kubernetes en dehors de ces fournisseurs, mais cela demande de faire des choix (ou bien une solution *opinionated* ouverte comme Rancher) et une relative maîtrise d'un nombre varié de sujets (bases de données, solutions de loadbalancing, redondance du stockage…).

C'est là la relative hypocrisie de Kubernetes : tout est ouvert et standardisé, mais devant la (relative) complexité et connaissance nécessaire pour mettre en place sa propre solution (de stockage distribué par exemple), nous retombons rapidement dans la facilité et les griffes du *vendor lock-in* (enfermement propriétaire).

Google Kubernetes Engine (GKE) (Google Cloud Plateform)

L'écosystème Kubernetes développé par Google. Très populaire car très flexible tout en étant l'implémentation de référence de Kubernetes.

Azure Kubernetes Services (AKS) (Microsoft Azure)

Un écosystème Kubernetes axé sur l'intégration avec les services du cloud Azure (stockage, registry, réseau, monitoring, services de calcul, loadbalancing, bases de données…).

Elastic Kubernetes Services (EKS) (Amazon Web Services)

Un écosystème Kubernetes assez standard à la sauce Amazon axé sur l'intégration avec le cloud Amazon (la gestion de l'accès, des loadbalancers ou du scaling notamment, le stockage avec Amazon EBS, etc.)

Rancher

Un écosystème Kubernetes très complet, assez *opinionated* et entièrement open-source, non lié à un fournisseur de cloud. Inclut l'installation de stack de monitoring (Prometheus), de logging, de réseau mesh (Istio) via une interface web agréable. Rancher maintient aussi de nombreuses solutions open source, comme par exemple Longhorn pour le stockage distribué.

k3s

Un écosystème Kubernetes fait par l'entreprise Rancher et axé sur la légèreté. Il remplace `etcd` par une base de données Postgres, utilise Traefik pour l'ingress et Klipper pour le loadbalancing.

Openshift

Une version de Kubernetes configurée et optimisée par Red Hat pour être utilisée dans son écosystème. Tout est intégré donc plus guidé, avec l'inconvénient d'être un peu captif·ve de l'écosystème et des services vendus par Red Hat.

Concurrents

Apache Mesos

Mesos est une solution plus générale que Kubernetes pour exécuter des applications distribuées. En combinant **Mesos** avec son “application framework” **Marathon** on obtient une solution équivalente sur de nombreux points à Kubernetes.

Elle est cependant moins standard : - Moins de ressources disponibles pour apprendre, intégrer avec d’autre solution etc. - Peu vendu en tant que service par les principaux cloud provider. - Plus chère à mettre en oeuvre.

Comparaison d’architecture : [Mesos vs. Kubernetes]

Parenthèse : Le YAML

Kubernetes décrit ses ressources en YAML. A quoi ça ressemble, YAML ?

```yaml
- marché:
    lieu: Marché de la Place
    jour: jeudi
    horaire:
      unité: "heure"
      min: 12
      max: 20
    fruits:
      - nom: pomme
        couleur: "verte"
        pesticide: avec

      - nom: poires
        couleur: jaune
        pesticide: sans
    légumes:
      - courgettes
      - salade
      - potiron
```

Syntaxe

- Alignement ! (**2 espaces** !!)
- ALIGNEMENT !! (comme en python)
- **ALIGNEMENT !!!** (le défaut du YAML, pas de correcteur syntaxique automatique, c’est bête mais vous y perdrez forcément du temps !)
- des listes (tirets)
- des paires **clé: valeur**
- Un peu comme du JSON, avec cette grosse différence que le JSON se fiche de l’alignement et met des accolades et des points-virgules

- **les extensions Kubernetes et YAML dans VSCode vous aident à repérer des erreurs**

Cours 2 - Mettre en place un cluster Kubernetes

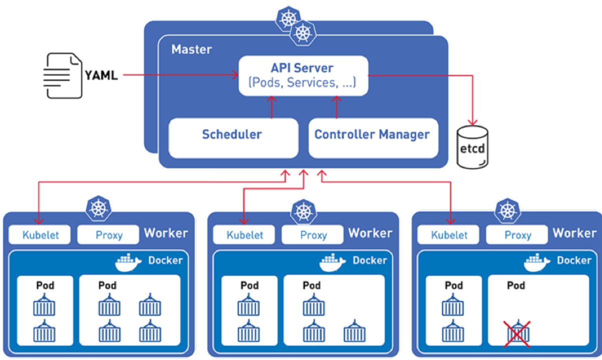
Architecture de Kubernetes - Partie 1

Kubernetes master

- Le Kubernetes master est responsable du maintien de l'état souhaité pour votre cluster. Lorsque vous interagissez avec Kubernetes, par exemple en utilisant l'interface en ligne de commande kubectl, vous communiquez avec le master Kubernetes de votre cluster.
- Le "master" fait référence à un ensemble de processus gérant l'état du cluster. Le master peut également être répliqué pour la disponibilité et la redondance.

Noeuds Kubernetes

Les nœuds d'un cluster sont les machines (serveurs physiques, machines virtuelles, etc.) qui exécutent vos applications et vos workflows. Le master node Kubernetes contrôle chaque noeud; vous interagirez rarement directement avec les nœuds.



- Pour utiliser Kubernetes, vous utilisez les objets de l'API Kubernetes pour décrire l'état souhaité de votre cluster: quelles applications ou autres processus que vous souhaitez exécuter, quelles images de conteneur elles utilisent, le nombre de réplicas, les ressources réseau et disque que vous mettez à disposition, et plus encore.
- Vous définissez l'état souhaité en créant des objets à l'aide de l'API Kubernetes, généralement via l'interface en ligne de commande, `kubectl` . Vous pouvez également utiliser l'API Kubernetes directement pour interagir avec le cluster et définir ou modifier l'état souhaité.
- Une fois que vous avez défini l'état souhaité, le plan de contrôle Kubernetes (control plane) permet de faire en sorte que l'état actuel du cluster corresponde à l'état souhaité. Pour ce faire, Kubernetes effectue automatiquement diverses tâches, telles que le démarrage ou le redémarrage de conteneurs, la mise à jour du nombre de

replicas d'une application donnée, etc.

Le Kubernetes Control Plane

- Le control plane Kubernetes comprend un ensemble de processus en cours d'exécution sur votre cluster:
    - Le master Kubernetes est un ensemble de trois processus qui s'exécutent sur un seul nœud de votre cluster, désigné comme nœud maître (*master node* en anglais). Ces processus sont:
        - `kube-apiserver`
        - `kube-controller-manager`
        - `kube-scheduler`
    - Chaque nœud non maître de votre cluster exécute deux processus : `kubelet`, qui communique avec le Kubernetes master. `kube-proxy`, un proxy réseau reflétant les services réseau Kubernetes sur chaque nœud.

Les différentes parties du control plane Kubernetes, telles que les processus Kubernetes master et kubelet, déterminent la manière dont Kubernetes communique avec votre cluster.

Le control plane conserve un enregistrement de tous les objets Kubernetes du système et exécute des boucles de contrôle continues pour gérer l'état de ces objets. À tout moment, les boucles de contrôle du control plane répondent aux modifications du cluster et permettent de faire en sorte que l'état réel de tous les objets du système corresponde à l'état souhaité que vous avez fourni.

Par exemple, lorsque vous utilisez l'API Kubernetes pour créer un objet `Deployment`, **vous fournissez un nouvel état souhaité pour le systèm**e. Le control plane Kubernetes enregistre la création de cet objet et exécute vos instructions en lançant les applications requises et en les planifiant vers des nœuds de cluster, afin que l'état actuel du cluster corresponde à l'état souhaité.

Le client `kubectl`

Permet depuis sa machine de travail de contrôler le cluster avec une ligne de commande qui ressemble un peu à celle de Docker (cf. TP1 et TP2):

- Lister les ressources
- Créer et supprimer les ressources
- Gérer les droits d'accès
- etc.

Cet utilitaire s'installe avec un gestionnaire de paquet classique mais est souvent fourni directement par une distribution de développement de kubernetes.

Nous l'installerons avec `snap` dans le TP1.

Pour se connecter, `kubectl` a besoin de l'adresse de l'API Kubernetes, d'un nom

d'utilisateur et d'un certificat.

- Ces informations sont fournies sous forme d'un fichier YAML appelé `kubeconfig`
- Comme nous le verrons en TP ces informations sont généralement fournies directement par le fournisseur d'un cluster k8s (provider ou k8s de dev)

Le fichier `kubeconfig` par défaut se trouve sur Linux à l'emplacement `~/.kube/config` .

On peut aussi préciser la configuration au *runtime* comme ceci: `kubectl --kubeconfig=fichier_kubeconfig.yaml <commandes_k8s>`

Le même fichier `kubeconfig` peut stocker plusieurs configurations dans un fichier YAML :

Exemple :

```yaml
apiVersion: v1

clusters:
- cluster:
    certificate-authority: /home/jacky/.minikube/ca.crt
    server: https://172.17.0.2:8443
  name: minikube
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUI
    server: https://5ba26bee-00f1-4088-ae11-22b6dd058c6e.k8s.ondigitalocea
  name: do-lon1-k8s-tp-cluster

contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
- context:
    cluster: do-lon1-k8s-tp-cluster
    user: do-lon1-k8s-tp-cluster-admin
  name: do-lon1-k8s-tp-cluster
current-context: do-lon1-k8s-tp-cluster

kind: Config
preferences: {}

users:
- name: do-lon1-k8s-tp-cluster-admin
  user:
      token: 8b2d33e45b980c8642105ec827f41ad343e8185f6b4526a481e312822d634a
- name: minikube
  user:
    client-certificate: /home/jacky/.minikube/profiles/minikube/client.crt
    client-key: /home/jacky/.minikube/profiles/minikube/client.key
```

Ce fichier déclare 2 clusters (un local, un distant), 2 contextes et 2 users.

Installation de développement

Pour installer un cluster de développement :

- solution officielle : Minikube, tourne dans Docker par défaut (ou dans des VMs)
- alternative qui possède de nombreux addons : microk8s
- avec Docker Desktop depuis peu (dans une VM aussi)
- un cluster léger avec `k3s` , de Rancher

Installer un cluster de production avec `kubeadm`

Installer un cluster de production Kubernetes à la main est nettement plus complexe que mettre en place un cluster Docker Swarm.

- Installer le dæmon `Kubelet` sur tous les noeuds
- Installer l'outil de gestion de cluster `kubeadm` sur un noeud master
- Générer les bons certificats avec `kubeadm`
- Installer un réseau CNI k8s comme `flannel` (d'autres sont possible et le choix vous revient)
- Déployer la base de données `etcd` avec `kubeadm`
- Connecter les nœuds worker au master.

L'installation est décrite dans la documentation officielle

Installer un cluster complètement à la main

On peut également installer Kubernetes de façon encore plus manuelle soit pour déployer une configuration vraiment spécifique ou simplement pour mieux comprendre ses rouages et composants.

Ce type d'installation est décrite par exemple ici : Kubernetes the hard way.

Commander un cluster en tant que service (*managed cluster*) dans le cloud

Tous les principaux provider de cloud fournissent depuis plus ou moins longtemps des solutions de cluster gérées par eux :

- Google Cloud Plateform avec Google Kubernetes Engine (GKE) : très populaire car très flexible et l'implémentation de référence de Kubernetes.
- AWS avec EKS : Kubernetes assez standard mais à la sauce Amazon pour la gestion de l'accès, des loadbalancers ou du scaling.
- DigitalOcean ou Scaleway : un peu moins de fonctions mais plus simple à appréhender

TP1 - Installation et configuration de Kubernetes

Au cours de nos TPs nous allons passer rapidement en revue deux manières de mettre en place Kubernetes :

- Un cluster de développement avec `minikube`
- Un cluster managed loué chez un provider (Scaleway, DigitalOcean, Azure ou Google Cloud)

Nous allons d'abord passer par la première option.

Découverte de Kubernetes

Installer le client CLI `kubectl`

kubectl est le point d'entré universel pour contrôler tous les type de cluster kubernetes. C'est un client en ligne de commande qui communique en REST avec l'API d'un cluster.

Nous allons explorer kubectl au fur et à mesure des TPs. Cependant à noter que :

- `kubectl` peut gérer plusieurs clusters/configurations et switcher entre ces configurations
- `kubectl` est nécessaire pour le client graphique `Lens` que nous utiliserons plus tard.

La méthode d'installation importe peu. Pour installer kubectl sur Ubuntu nous ferons simplement: `sudo snap install kubectl --classic` .

- Faites `kubectl version` pour afficher la version du client kubectl.

Installer Minikube

Minikube est la version de développement de Kubernetes (en local) la plus répendue. Elle est maintenue par la cloud native foundation et très proche de kubernetes upstream. Elle permet de simuler un ou plusieurs noeuds de cluster sous forme de conteneurs docker ou de machines virtuelles.

- Pour installer minikube la méthode recommandée est indiquée ici: https://minikube.sigs.k8s.io/docs/start/

Nous utiliserons classiquement `docker` comme runtime pour minikube (les noeuds k8s seront des conteneurs simulant des serveurs). Ceci est, bien sur, une configuration de développement. Elle se comporte cependant de façon très proche d'un véritable cluster.

- Si Docker n'est pas installé, installer Docker avec la commande en une seule ligne : `curl -fsSL https://get.docker.com | sh` , puis ajoutez-vous au groupe Docker

avec `sudo usermod -a -G docker <votrenom>` , et faites `sudo reboot` pour que cela prenne effet.

- Pour lancer le cluster faites simplement: `minikube start` (il est également possible de préciser le nombre de coeurs de calcul, la mémoire et et d'autre paramètre pour adapter le cluster à nos besoins.)

Minikube configure automatiquement kubectl (dans le fichier `~/.kube/config`) pour qu'on puisse se connecter au cluster de développement.

- Testez la connexion avec `kubectl get nodes` .

Affichez à nouveau la version `kubectl version` . Cette fois-ci la version de kubernetes qui tourne sur le cluster actif est également affichée. Idéalement le client et le cluster devrait être dans la même version mineure par exemple `1.20.x` .

Bash completion

Pour permettre à `kubectl` de compléter le nom des commandes et ressources avec `<Tab>` il est utile d'installer l'autocomplétion pour Bash :

```
sudo apt install bash-completion

source <(kubectl completion bash)

echo "source <(kubectl completion bash)" >> ${HOME}/.bashrc
```

Vous pouvez désormais appuyer sur `<Tab>` pour compléter vos commandes `kubectl` , c'est très utile !

Explorons notre cluster k8s

Notre cluster k8s est plein d'objets divers, organisés entre eux de façon dynamique pour décrire des applications, tâches de calcul, services et droits d'accès. La première étape consiste à explorer un peu le cluster :

- Listez les nodes pour récupérer le nom de l'unique node (`kubectl get nodes`) puis affichez ses caractéristiques avec `kubectl describe node/minikube` .

La commande `get` est générique et peut être utilisée pour récupérer la liste de tous les types de ressources.

De même, la commande `describe` peut s'appliquer à tout objet k8s. On doit cependant préfixer le nom de l'objet par son type (ex : `node/minikube` ou `nodes minikube`) car k8s ne peut pas deviner ce que l'on cherche quand plusieurs ressources ont le même nom.

- Pour afficher tous les types de ressources à la fois que l'on utilise : `kubectl get all`

```
NAME                 TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1   <none>        443/TCP   2m34s
```

Il semble qu'il n'y a qu'une ressource dans notre cluster. Il s'agit du service d'API Kubernetes, pour qu'on puisse communiquer avec le cluster.

En réalité il y en a généralement d'autres cachés dans les autres `namespaces`. En effet les éléments internes de Kubernetes tournent eux-mêmes sous forme de services et de daemons Kubernetes. Les *namespaces* sont des groupes qui servent à isoler les ressources de façon logique et en termes de droits (avec le *Role-Based Access Control* (RBAC) de Kubernetes).

Pour vérifier cela on peut :

- Afficher les namespaces : `kubectl get namespaces`

Un cluster Kubernetes a généralement un namespace appelé `default` dans lequel les commandes sont lancées et les ressources créées si on ne précise rien. Il a également aussi un namespace `kube-system` dans lequel résident les processus et ressources système de k8s. Pour préciser le namespace on peut rajouter l'argument `-n` à la plupart des commandes k8s.

- Pour lister les ressources liées au `kubectl get all -n kube-system`.
- Ou encore : `kubectl get all --all-namespaces` (peut être abrégé en `kubectl get all -A`) qui permet d'afficher le contenu de tous les namespaces en même temps.
- Pour avoir des informations sur un namespace : `kubectl describe namespace/kube-system`

Déployer une application

Nous allons maintenant déployer une première application conteneurisée. Le déploiement est plus complexe qu'avec Docker (et Swarm), en particulier car il est séparé en plusieurs objets et plus configurable.

- Pour créer un déploiement en ligne de commande (par opposition au mode déclaratif que nous verrons plus loin), on peut lancer par exemple: `kubectl create deployment microbot --image=monachus/rancher-demo`.

Cette commande crée un objet de type `deployment`. Nous pourvons étudier ce deployment avec la commande `kubectl describe deployment/microbot`.

- Agrandissons ce déploiement avec `kubectl scale deployment microbot --replicas=5`
- `kubectl describe deployment/microbot` permet de constater que le service est bien passé à 5 replicas.

A ce stade le déploiement n'est pas encore accessible de l'extérieur du cluster pour cela nous devons l'exposer en tant que service :

- `kubectl expose deployment microbot --type=NodePort --port=8080 --name=microbot-service`

- affichons la liste des services pour voir le résultat: `kubectl get services`

Un service permet d'exposer un déploiement soit par port soit grâce à un loadbalancer.

Pour exposer cette application sur le port de notre choix, nous devrions avoir recours à un LoadBalancer.

Nous verrons cela plus en détail dans le TP2.

Nous ne verrons pas ça ici (il faudrait utiliser l'addon MetalLB de Minikube).

Mais nous pouvons quand même lancer une commande dans notre environnement de dev : `kubectl port-forward svc/microbot-service 8080:8080 --address 0.0.0.0`

Vous pouvez désormais accéder à votre app via : `http://localhost:8080`

Minikube intègre aussi une façon d'accéder à notre service : c'est la commande `minikube service microbot-service`

Sauriez-vous expliquer ce que l'app fait ?

Simplifier les lignes de commande k8s

- Pour gagner du temps on dans les commandes Kubernetes on définit généralement un alias: `alias kc='kubectl'` (à mettre dans votre `.bash_profile` en faisant `echo "alias kc='kubectl'" >> ~/.bash_profile`, puis en faisant `source ~/.bash_profile`).
- Vous pouvez ensuite remplacer `kubectl` par `kc` dans les commandes.
- Également pour gagner du temps en ligne de commande, la plupart des mots-clés de type Kubernetes peuvent être abrégés :
  - `services` devient `svc`
  - `deployments` devient `deploy`
  - etc.

La liste complète : https://blog.heptio.com/kubectl-resource-short-names-heptioprotip-c8eff9fb7202

- Essayez d'afficher les serviceaccounts (users) et les namespaces avec une commande courte.

Installer Lens

Lens est une interface graphique sympatique pour Kubernetes.

Elle se connecte en utilisant la configuration `~/.kube/config` par défaut et nous permettra d'accéder à un dashboard bien plus agréable à utiliser.

Vous pouvez l'installer en lançant ces commandes :

```
sudo apt-get update; sudo apt-get install -y libxss-dev
curl -fSL https://github.com/lensapp/lens/releases/download/v4.0.6/Lens-4.(
chmod +x ~/Lens.AppImage
~/Lens.AppImage &
```

Mettre en place un cluster K8s dans le cloud avec un provider type DigitalOcean ou Scaleway

- Créez un compte (ou récupérez un accès) sur DigitalOcean ou Scaleway
- Créez un cluster Kubernetes avec l'interface DigitalOcean ou bien l'interface Scaleway

La création prend environ 5 minutes.

- Sur DigitalOcean, il vous est proposé dans l'étape 3 ou sur la page de votre cluster Kubernetes de télécharger le fichier `kubeconfig`. (*download the cluster configuration file*, ou bien *Download Config File*).
- De même, sur Scaleway, sur la page décrivant votre cluster, un gros bouton en bas de la page vous incite à télécharger ce même fichier `kubeconfig` (*Download Kubeconfig*).

Ce fichier contient la **configuration kubectl** adaptée pour la connexion à notre cluster.

Merger la configuration kubectl

- Ouvrez avec `gedit` les fichiers `kubeconfig` et `~/.kube/config`.
- fusionnez dans `~/.kube/config` les éléments des listes YAML de:
  - `clusters`
  - `contexts`
  - `users`
- mettez la clé `current-context:` à `<nom_cluster>` (compléter avec votre valeur)
- Testons la connection avec `kubectl get nodes`.

Déployer l'application

- Lancez `kubectl cluster-info`, l'API du cluster est accessible depuis un nom de domaine généré par le provider.
- Déployez l'application `microbot` comme dans la partie précédente avec `minikube`
- Pour visitez l'application vous devez trouver l'IP publique d'un des nœuds du cluster en listant les objets de type `Service`, ou sur la page du fournisseur de cloud.

Cours 3 - Concepts de Kubernetes

Principes d'orchestration

Haute disponibilité

- Faire en sorte qu'un service ait un "uptime" élevé.

On veut que le service soit tout le temps accessible même lorsque certaines ressources manquent :

- elles tombent en panne
- elles sont sorties du service pour mise à jour, maintenance ou modification

Pour cela on doit avoir des ressources multiples…

- Plusieurs serveurs
- Plusieurs versions des données
- Plusieurs accès réseau

Il faut que les ressources disponibles prennent automatiquement le relais des ressources indisponibles. Pour cela on utilise généralement:

- des "load balancers" : aiguillages réseau intelligents
- des "healthchecks" : une vérification de la santé des applications

Mais aussi :

- des réseaux de secours
- des IP flottantes qui fonctionnent comme des load balancers
- etc.

Nous allons voir que Kubernetes intègre automatiquement les principes de load balancing et de healthcheck dans l'orchestration de conteneurs

Répartition de charge (load balancing)

- Un load balancer : une sorte d'**"aiguillage" de trafic réseau**, typiquement HTTP(S) ou TCP.
- Un aiguillage **intelligent** qui se renseigne sur plusieurs critères avant de choisir la direction.

Cas d'usage :

- Éviter la surcharge : les requêtes sont réparties sur différents backends pour éviter de les saturer.

L'objectif est de permettre la haute disponibilité : on veut que notre service soit toujours disponible, même en période de panne/maintenance.

- Donc on va dupliquer chaque partie de notre service et mettre les différentes instances derrière un load balancer.
- Le load balancer va vérifier pour chaque backend s'il est disponible (**healthcheck**) avant de rediriger le trafic.
- Répartition géographique : en fonction de la provenance des requêtes on va rediriger vers un datacenter adapté (+ proche).

Solutions de load balancing externe

- **HAProxy** : Le plus répandu en load balancing.
- **Traefik** : Simple à configurer et se fond dans l'écosystème des conteneurs Docker et Kubernetes.
- **NGINX** : Serveur web central qui a depuis quelques années des fonctions puissantes de load balancing et TCP forwarding.

Healthchecks

Fournir à l'application une façon d'indiquer qu'elle est disponible, c'est-à-dire :

- qu'elle est démarrée (*liveness*)
- qu'elle peut répondre aux requêtes (*readiness*).

Découverte de service (service discovery)

Classiquement, les applications ne sont pas informées du contexte dans lequel elles tournent : la configuration doit être opérée de l'extérieur de l'application.

- par exemple avec des fichiers de configuration fournie via des volumes
- ou via des variables d'environnement

Mais dans un environnement hautement dynamique comme Kubernetes, la configuration externe ne suffit pas pour gérer des applications complexes distribuées qui doivent se déployer régulièrement, se parler et parler avec l'extérieur.

La découverte de service désigne généralement les méthodes qui permettent à un programme de chercher autour de lui (généralement sur le réseau ou dans l'environnement) ce dont il a besoin.

- La mise en place d'un système de **découverte de service** permet de rendre les applications plus autonomes dans leur (auto)configuration.
- Elles vont pouvoir récupérer des informations sur leur contexte (dev ou prod, Etats-Unis ou Europe ?)
- Ce type d'automatisation permet de limiter la complexité du déploiement.

Concrètement, au sein d'un orchestrateur, un système de découverte de service est un serveur qui est au courant automatiquement :

- de chaque conteneur lancé.
- du contexte dans lequel chaque conteneur a été lancé.

Ensuite il suffit aux applications de pouvoir interroger ce serveur pour s'autoconfigurer.

Un exemple historique de découverte de service est le DNS : on fait une requête vers un serveur spécial pour retrouver une adresse IP (on découvre le serveur dont on a besoin). Cependant le DNS n'a pas été pensé pour ça :

- certaines application ne rafraichissent pas assez souvent leurs enregistrements DNS en cache

- le DNS devient trop complexe à partir de quelques dizaines d'enregistrements

Solutions de découverte de service

- **Consul** (Hashicorp) : assez simple d'installation et fourni avec une sympathique interface web.
- **etcd** : a prouvé ses performances à plus grande échelle mais un peu plus complexe

Les stratégies de déploiement

Il existe deux types de stratégies de *rollout* native à Kubernetes :

- `Recreate` : arrêter les pods avec l'ancienne version en même temps et créer les nouveaux simultanément
- `RollingUpdate` : mise à jour continue, arrêt des anciens pods les uns après les autres et création des nouveaux au fur et à mesure (paramétrable)

Mais il existe un panel de stratégies plus large pour updater ses apps :

- *blue/green* : publier une nouvelle version à côté de l'ancienne puis changer de trafic
- *canary* : diffuser une nouvelle version à un sous-ensemble d'utilisateurs, puis procéder à un déploiement complet
- *A/B testing*: diffusion d'une nouvelle version à un sous-ensemble d'utilisateurs de manière précise (en-têtes HTTP, cookie, région, etc.).
  - pas possible par défaut avec Kubernetes, implique une infrastructure plus avancée avec reverse proxy (Istio, Traefik, nginx/haproxy personnalisé, etc.).

Source : https://blog.container-solutions.com/kubernetes-deployment-strategies

Cours 4 - Objets Kubernetes - Partie 1

L'API et les Objets Kubernetes

Utiliser Kubernetes consiste à déclarer des objets grâce à l'API Kubernetes pour décrire l'état souhaité d'un cluster : quelles applications ou autres processus exécuter, quelles images elles utilisent, le nombre de replicas, les ressources réseau et disque que vous mettez à disposition, etc.

On définit des objets généralement via l'interface en ligne de commande et `kubectl` de deux façons :

- en lançant une commande `kubectl run <conteneur> ...` , `kubectl expose ...`
- en décrivant un objet dans un fichier YAML ou JSON et en le passant au client `kubectl apply -f monpod.yml`

Vous pouvez également écrire des programmes qui utilisent directement l'API Kubernetes pour interagir avec le cluster et définir ou modifier l'état souhaité. **Kubernetes est**

complètement automatisable !

La commande `apply`

Kubernetes encourage le principe de l'infrastructure-as-code : il est recommandé d'utiliser une description YAML et versionnée des objets et configurations Kubernetes plutôt que la CLI.

Pour cela la commande de base est `kubectl apply -f object.yaml` .

La commande inverse `kubectl delete -f object.yaml` permet de détruire un objet précédement appliqué dans le cluster à partir de sa description.

Lorsqu'on vient d'appliquer une description on peut l'afficher dans le terminal avec `kubectl apply -f myobj.yaml view-last-applied`

Globalement Kubernetes garde un historique de toutes les transformations des objets : on peut explorer, par exemple avec la commande `kubectl rollout history deployment` .

Syntaxe de base d'une description YAML Kubernetes

Les description YAML permettent de décrire de façon lisible et manipulable de nombreuses caractéristiques des ressources Kubernetes (un peu comme un *Compose file* par rapport à la CLI Docker).

Exemples

Création d'un service simple :

```yaml
kind: Service
apiVersion: v1
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
spec:
  ports:
    - port: 443
      targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  type: NodePort
```

Création d'un “compte utiliseur” `ServiceAccount`

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
```

```yaml
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
```

Remarques de syntaxe :

- Toutes les descriptions doivent commencer par spécifier la version d'API (minimale) selon laquelle les objets sont censés être créés
- Il faut également préciser le type d'objet avec `kind`
- Le nom dans `metadata:\n name: value` est également obligatoire.
- On rajoute généralement une description longue démarrant par `spec:`

Description de plusieurs ressources

- On peut mettre plusieurs ressources à la suite dans un fichier k8s : cela permet de décrire une installation complexe en un seul fichier
  - par exemple le dashboard Kubernetes https://github.com/kubernetes/dashboard/blob/master/aio/deploy/recommended.yaml
- L'ordre n'importe pas car les ressources sont décrites déclarativement c'est-à-dire que:
  - Les dépendances entre les ressources sont déclarées
  - Le control plane de Kubernetes se charge de planifier l'ordre correct de création en fonction des dépendances (pods avant le déploiement, rôle avec l'utilisateur lié au rôle)
  - On préfère cependant les mettre dans un ordre logique pour que les humains puissent les lire.
- On peut sauter des lignes dans le YAML et rendre plus lisible les descriptions
- On sépare les différents objets par `---`

Objets de base

Les namespaces

Tous les objets Kubernetes sont rangés dans différents espaces de travail isolés appelés `namespaces` .

Cette isolation permet 3 choses :

- ne voir que ce qui concerne une tâche particulière (ne réfléchir que sur une seule chose lorsqu'on opère sur un cluster)
- créer des limites de ressources (CPU, RAM, etc.) pour le namespace
- définir des rôles et permissions sur le namespace qui s'appliquent à toutes les ressources à l'intérieur.

Lorsqu'on lit ou créé des objets sans préciser le namespace, ces objets sont liés au namespace `default` .

Pour utiliser un namespace autre que `default` avec `kubectl` il faut :

- le préciser avec l'option `-n` : `kubectl get pods -n kube-system`
- créer une nouvelle configuration dans la kubeconfig pour changer le namespace par defaut.

Kubernetes gère lui-même ses composants internes sous forme de pods et services.

- Si vous ne trouvez pas un objet, essayez de lancer la commande kubectl avec l'option `-A` ou `--all-namespaces`

Les Pods

Un Pod est l'unité d'exécution de base d'une application Kubernetes que vous créez ou déployez. Un Pod représente des process en cours d'exécution dans votre Cluster.

Un Pod encapsule un conteneur (ou souvent plusieurs conteneurs), des ressources de stockage, **une IP réseau unique**, et des options qui contrôlent comment le ou les conteneurs doivent s'exécuter (ex: *restart policy*). Cette collection de conteneurs et volumes tournent dans le même environnement d'exécution mais les processus sont isolés.

Un Pod représente une unité de déploiement : un petit nombre de conteneurs qui sont étroitement liés et qui partagent :

- les mêmes ressources de calcul
- des volumes communs
- la même IP donc le même nom de domaine
- peuvent se parler sur localhost
- peuvent se parler en IPC
- ont un nom différent et des logs différents

Chaque Pod est destiné à exécuter une instance unique d'un workload donné. Si vous désirez mettre à l'échelle votre workload, vous devez multiplier le nombre de Pods.

Pour plus de détail sur la philosophie des pods, vous pouvez consulter ce bon article.

Kubernetes fournit un ensemble de commande pour débugger des conteneurs :

- `kubectl logs <pod-name> -c <conteneur_name>` (le nom du conteneur est inutile si un seul)
- `kubectl exec -it <pod-name> -c <conteneur_name> -- bash`
- `kubectl attach -it <pod-name>`

Enfin, pour debugger la sortie réseau d'un programme on peut rapidement forwarder un port depuis un pods vers l'extérieur du cluster :

- `kubectl port-forward <pod-name> <port_interne>:<port_externe>`
- C'est une commande de debug seulement : pour exposer correctement des processus k8s, il faut créer un service, par exemple avec `NodePort` .

Pour copier un fichier dans un pod on peut utiliser: `kubectl cp <pod-name>:</path/to/remote/file> </path/to/local/file>`

Pour monitorer rapidement les ressources consommées par un ensemble de processus il existe les commande `kubectl top nodes` et `kubectl top pods`

Un manifeste de Pod

`kuard-pod.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nom_pod
spec:
  containers:
    - image: tecpi/pod_image:0.1
      name: nom_conteneur
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Les ReplicaSet

Un ReplicaSet ou `rs` est une ressource qui permet de spécifier finement le nombre de réplication d'un pod à un moment donné.

- `kubectl get rs` pour afficher la liste des replicas.

En général on ne les manipule pas directement.

Les Deployments

Plutôt que d'utiliser les replicasets il est recommander d'utiliser un objet de plus haut niveau : les *deployments*.

De la même façon que les ReplicaSets gèrent les pods, les Deployments gèrent les ReplicaSet.

Un déploiement sert surtout à gérer le déploiement d'une nouvelle version d'un pod.

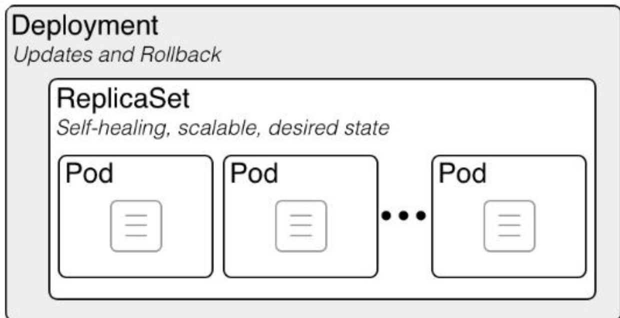
Un *deployment* est un peu l'équivalent d'un *service* docker : il demande la création d'un ensemble de Pods désignés par une étiquette `label` .

Exemple :

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

```yaml
      app: nginx
  spec:
    replicas: 3
    strategy:
      type: Recreate
    selector:
      matchLabels:
        app: nginx
    template:
      metadata:
        labels:
          app: nginx
      spec:
        containers:
          - name: nginx
            image: nginx:1.7.9
            ports:
              - containerPort: 80
```

- Pour les afficher : `kubectl get deployments`
- La commande `kubectl run` sert à créer un *deployment* à partir d'un modèle. Il vaut mieux utilisez `apply -f` .



Les poupées russes Kubernetes : un Deployment contient un ReplicaSet, qui contient des Pods, qui contiennent des conteneurs

Les Services

Dans Kubernetes, un service est un objet qui :

- rassemble un ensemble de pods (grâce à des tags)
- et configure une politique permettant d'y accéder depuis l'intérieur ou l'extérieur du cluster.

L'ensemble des pods ciblés par un service est déterminé par un `selector` .

Par exemple, considérons un backend de traitement d'image (*stateless*, c'est-à-dire ici sans base de données) qui s'exécute avec 3 replicas. Ces replicas sont interchangeables et les frontends ne se soucient pas du backend qu'ils utilisent. Bien que les pods réels qui composent l'ensemble `backend` puissent changer, les clients frontends ne devraient pas avoir besoin de le savoir, pas plus qu'ils ne doivent suivre eux-mêmes l'état de l'ensemble des backends.

L'abstraction du service permet ce découplage : les clients frontend s'addressent à une seule IP avec un seul port dès qu'ils ont besoin d'avoir recours à un backend. Les backends vont recevoir la requête du frontend aléatoirement.

Les Services sont de trois types principaux :

- `ClusterIP` : expose le service **sur une IP interne** au cluster appelée ClusterIP. Les autres pods peuvent alors accéder au service mais pas l'extérieur.
- `NodePort` : expose le service depuis l'IP publique de **chacun des noeuds du cluster** en ouvrant port directement sur le nœud, entre 30000 et 32767. Cela permet d'accéder aux pods internes répliqués. Comme l'IP est stable on peut faire pointer un DNS ou Loadbalancer classique dessus.

- `LoadBalancer` : expose le service en externe à l'aide d'un Loadbalancer de fournisseur de cloud. Les services NodePort et ClusterIP, vers lesquels le Loadbalancer est dirigé sont automatiquement créés.

TP 2 - Déployer Wordpress rapidement

Déployer Wordpress et MySQL avec du stockage et des Secrets

Nous allons suivre ce tutoriel pas à pas : https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/

Il faut :

- copier les 2 fichiers et les appliquer
- vérifier que le stockage a bien fonctionné
- découvrir ce qui manque pour que cela fonctionne
- le créer à la main ou suivre le reste du tutoriel qui passe par l'outil Kustomize (attention, Kustomize ajoute un suffixe aux ressources qu'il créé)

On peut ensuite observer les différents objets créés, et optimiser le process avec un fichier `kustomization.yaml` plus complet.

- Entrez dans un des pods, et de l'intérieur, lisez le secret qui lui a été rendu accessible.

Facultatif : la stack *Wordsmith*

Etudions et lançons ensemble ce YAML :

`wordsmith.yml` :

```
apiVersion: v1
```

```yaml
kind: Service
metadata:
  name: db
  labels:
    app: words-db
spec:
  ports:
    - port: 5432
      targetPort: 5432
      name: db
  selector:
    app: words-db
  clusterIP: None
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db
  labels:
    app: words-db
spec:
  selector:
    matchLabels:
      app: words-db
  template:
    metadata:
      labels:
        app: words-db
    spec:
      containers:
      - name: db
        image: dockersamples/k8s-wordsmith-db
        ports:
        - containerPort: 5432
          name: db
---
apiVersion: v1
kind: Service
metadata:
  name: words
  labels:
    app: words-api
spec:
  ports:
    - port: 8080
      targetPort: 8080
      name: api
  selector:
    app: words-api
  clusterIP: None
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: words
  labels:
```

```yaml
    app: words-api
spec:
  selector:
    matchLabels:
      app: words-api
  replicas: 5
  template:
    metadata:
      labels:
        app: words-api
    spec:
      containers:
      - name: words
        image: dockersamples/k8s-wordsmith-api
        ports:
        - containerPort: 8080
          name: api
---
apiVersion: v1
kind: Service
metadata:
  name: web
  labels:
    app: words-web
spec:
  ports:
    - port: 8081
      targetPort: 80
      name: web
  selector:
    app: words-web
  type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  labels:
    app: words-web
spec:
  selector:
    matchLabels:
      app: words-web
  template:
    metadata:
      labels:
        app: words-web
    spec:
      containers:
      - name: web
        image: dockersamples/k8s-wordsmith-web
        ports:
        - containerPort: 80
          name: words-web
```

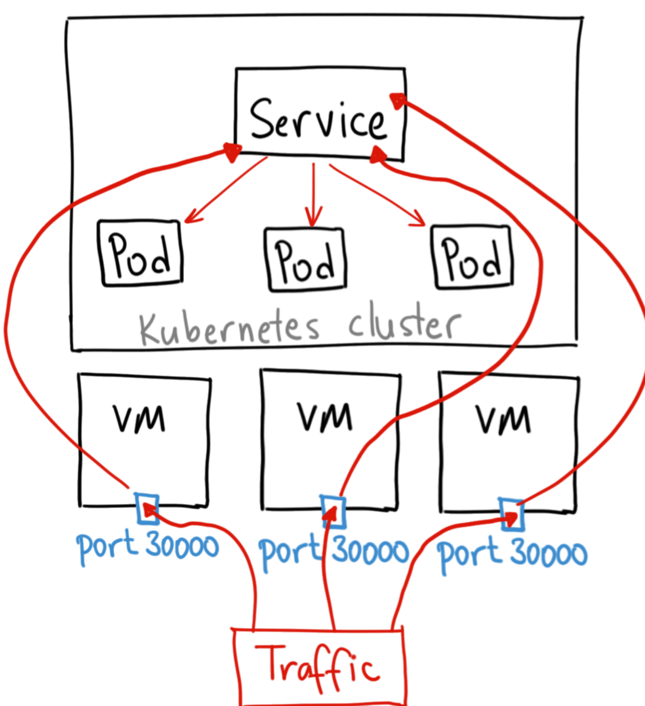
Cours 5 - Le réseau dans Kubernetes

Les solutions réseau dans Kubernetes ne sont pas standard. Il existe plusieurs façons d'implémenter le réseau.

Les objets Services

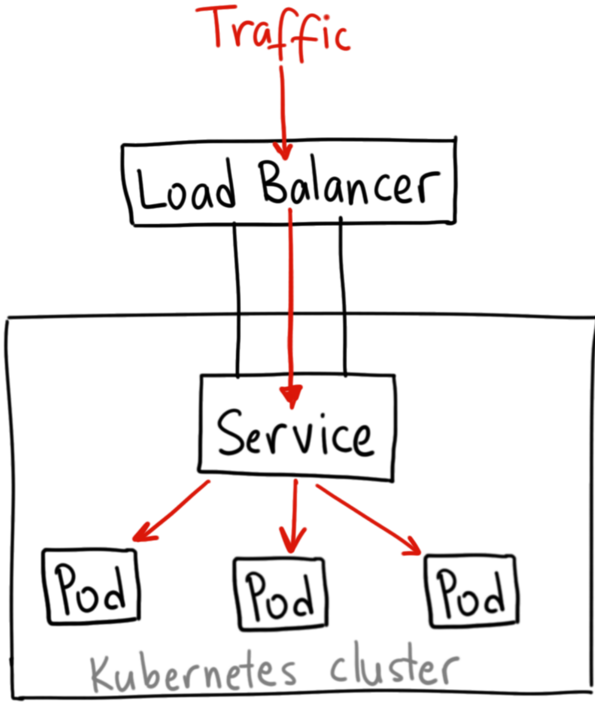
Les Services sont de trois types principaux :

- `ClusterIP` : expose le service **sur une IP interne** au cluster appelée ClusterIP. Les autres pods peuvent alors accéder au service mais pas l'extérieur.
- `NodePort` : expose le service depuis l'IP publique de **chacun des noeuds du cluster** en ouvrant port directement sur le nœud, entre 30000 et 32767. Cela permet d'accéder aux pods internes répliqués. Comme l'IP est stable on peut faire pointer un DNS ou Loadbalancer classique dessus.



Crédits à Ahmet Alp Balkan pour les schémas

- `LoadBalancer` : expose le service en externe à l'aide d'un Loadbalancer de fournisseur de cloud. Les services NodePort et ClusterIP, vers lesquels le Loadbalancer est dirigé sont automatiquement créés.



Crédits Ahmet Alp Balkan

Les implémentations du réseau

Beaucoup de solutions de réseau qui se concurrencent, demandant un comparatif un peu fastidieux.

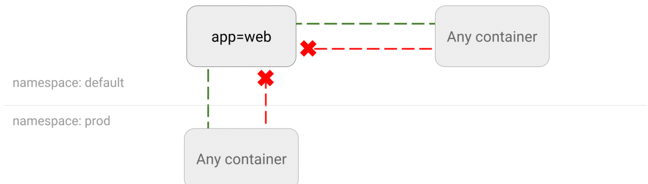
- plusieurs solutions très robustes
- diffèrent sur l'implémentation : BGP, réseau overlay ou non (encapsulation VXLAN, IPinIP, autre)
- toutes ne permettent pas d'appliquer des **NetworkPolicies** : l'isolement et la sécurité réseau
- peuvent parfois s'hybrider entre elles (Canal = Calico + Flannel)

- ces implémentations sont souvent concrètement des *DaemonSets* : des pods qui tournent dans chacun des nodes de Kubernetes
- Calico, Flannel, Weave ou Cilium sont très employées et souvent proposées en option par les fournisseurs de cloud
- Cilium a la particularité d'utiliser la technologie eBPF de Linux qui permet une sécurité et une rapidité accrue

Comparaisons :

- https://www.objectif-libre.com/fr/blog/2018/07/05/comparatif-solutions-reseaux-kubernetes/
- https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/

Les network policies



Crédits [Ahmet Alp Balkan]

Par défaut, les pods ne sont pas isolés au niveau réseau : ils acceptent le trafic de n'importe quelle source.

Les pods deviennent isolés en ayant une NetworkPolicy qui les sélectionne. Une fois qu'une NetworkPolicy (dans un certain namespace) inclut un pod particulier, ce pod rejettera toutes les connexions qui ne sont pas autorisées par cette NetworkPolicy.

- Des exemples de Network Policies : Kubernetes Network Policy Recipes

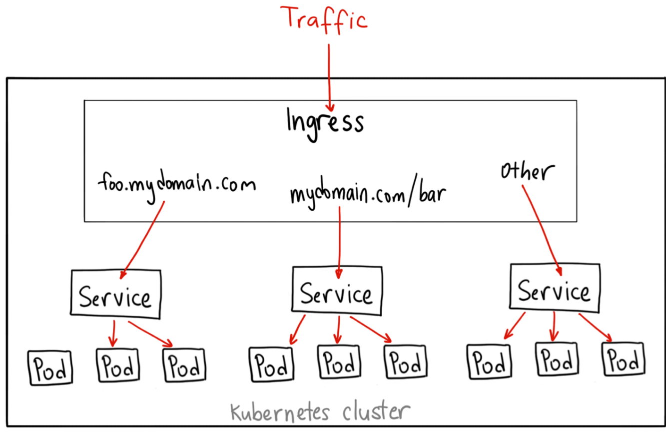
Le loadbalancing

Le loadbalancing permet de balancer le trafic à travers plusieurs nodes Kubernetes.

Pas de solution de loadbalancing par défaut :

- soit on se base sur ce que le fournisseur de cloud propose,
- soit on configure *MetalLB*, seule alternative en dehors des fournisseurs de cloud

Les objets Ingresses



Crédits Ahmet Alp Balkan

Un Ingress est un objet pour gérer le **reverse proxy** dans Kubernetes : il a besoin d'un **ingress controller** installé sur le cluster, qui agit donc au niveau du protocole HTTP et écoute sur un port (`80` ou `443` généralement), pour pouvoir rediriger vers différents services (qui à leur tour redirigent vers différents ports sur les pods) selon l'URL.

- Un ingress basé sur Nginx plus ou moins officiel à Kubernetes et très utilisé
- Traefik est optimisé pour k8s
- il en existe d'autres : celui de l'entreprise Nginx, Istio, Contour, HAProxy....

Comparaison : https://medium.com/flant-com/comparing-ingress-controllers-for-kubernetes-9b397483b46b

Le mesh networking et les *service meshes*

Envoy et Istio sont des *service meshes*.

- Il faut y penser comme des super-ingresses : des proxy qui font beaucoup plus que du reverse proxy
  - en particulier : ajouter des fonctions de monitoring et de sécurité

Ressources sur le réseau

- Documentation officielle : https://kubernetes.io/fr/docs/concepts/services-networking/service/
- An introduction to service meshes - DigitalOcean
- Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?
- Determine best networking option - Project Calico
- Doc officielle sur les solutions de networking

Vidéos

Des vidéos assez complètes sur le réseau, faites par Calico :

- Kubernetes Ingress networking
- Kubernetes Services networking
- Kubernetes networking on Azure

Sur MetalLB, les autres vidéos de la chaîne sont très bien :

- Why you need to use MetalLB - Adrian Goins

TP 3 - Déployer des conteneurs de A à Z

Ce TP va consister à créer des objets Kubernetes pour déployer une stack d'exemple : `monster_stack` . Elle est composée :

- d'un front-end en Flask (Python),
- d'un backend qui génère des images (un avatar de monstre correspondant à une chaîne de caractères),
- et d'une base de données servant de cache pour ces images, Redis.

Vous pouvez utiliser au choix votre environnement Cloud ou Minikube.

Rappel : Installer Lens

Lens est une interface graphique sympathique pour Kubernetes.

Elle se connecte en utilisant la configuration `~/.kube/config` par défaut et nous permettra d'accéder à un dashboard bien plus agréable à utiliser.

Vous pouvez l'installer en lançant ces commandes :

```
sudo apt-get update; sudo apt-get install -y libxss-dev
curl -fSL https://github.com/lensapp/lens/releases/download/v4.0.6/Lens-4.(
chmod +x ~/Lens.AppImage
~/Lens.AppImage &
```

Déploiement de la stack `monsterstack`

Les pods sont des ensembles de conteneurs toujours gardés ensembles.

Nous voudrions déployer notre stack `monster_app`. Nous allons commencer par créer un pod avec seulement notre conteneur `monstericon`.

- Créez un projet vide `monster_app_k8s`.
- Créez le fichier de déploiement suivant:

`monstericon.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: monstericon
  labels:
    <labels>
```

Ce fichier exprime un objet déploiement vide.

- Ajoutez le label `app: monsterstack` à cet objet `Deployment`.
- Pour le moment notre déploiement n'est pas défini car il n'a pas de section `spec:`.
- La première étape consiste à proposer un modèle de `ReplicaSet` pour notre déploiement. Ajoutez à la suite (`spec:` doit être à la même hauteur que `kind:` et `metadata:`) :

```yaml
spec:
  template:
    spec:
```

Remplissons la section `spec` de notre pod `monstericon` à partir d'un modèle de pod lançant un conteneur Nginx :

```yaml
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
```

```
        - containerPort: 80
```

- Remplacez le nom du conteneur par `monstericon` , et l'image de conteneur par `tecpi/monster_icon:0.1` , cela récupérera l'image préalablement uploadée sur le Docker Hub (à la version 0.1)
- Complétez le port en mettant le port de production de notre application, `9090`
- Les objets dans Kubernetes sont hautement dynamiques. Pour les associer et les désigner on leur associe des `labels` c'est-à-dire des étiquettes avec lesquelles on peut les retrouver ou les matcher précisément. C'est grâce à des labels que k8s associe les `pods` aux `ReplicaSets` . Ajoutez à la suite au même niveau que la spec du pod :

```
metadata:
  labels:
    app: monsterstack
    partie: monstericon
```

A ce stade nous avons décrit les pods de notre déploiement avec leurs labels (un label commun à tous les objets de l'app, un label plus spécifique à la sous-partie de l'app).

Maintenant il s'agit de rajouter quelques options pour paramétrer notre déploiement (à la hauteur de `template:`) :

```
selector:
  matchLabels:
    app: monsterstack
    partie: monstericon
strategy:
  type: Recreate
```

Cette section indique les labels à utiliser pour repérer les pods de ce déploiement parmi les autres.

Puis est précisée la stratégie de mise à jour (rollout) des pods pour le déploiement : `Recreate` désigne la stratégie la plus brutale de suppression complète des pods puis de redéploiement.

Enfin, juste avant la ligne `selector:` et à la hauteur du mot-clé `strategy:` , ajouter `replicas: 3` . Kubernetes créera 3 pods identiques lors du déploiement `monstericon` .

Le fichier `monstericon.yaml` jusqu'à présent :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: monstericon
  labels:
    app: monsterstack
```

```yaml
spec:
  template:
    spec:
      containers:
      - name: monstericon
        image: tecpi/monster_icon:0.1
        ports:
        - containerPort: 9090
    metadata:
      labels:
        app: monsterstack
        partie: monstericon
  selector:
    matchLabels:
      app: monsterstack
      partie: monstericon
  strategy:
    type: Recreate
  replicas: 3
```

Appliquer notre déploiement

- Avec la commande `apply -f` appliquez notre fichier de déploiement.
- Affichez les déploiements avec `kubectl get deploy -o wide` .
- Listez également les pods en lançant `kubectl get pods --watch` pour vérifier que les conteneurs tournent.
- Ajoutons un healthcheck de type `readinessProbe` au conteneur dans le pod avec la syntaxe suivante (le mot-clé `readinessProbe` doit être à la hauteur du `i` de `image:`) :

```yaml
readinessProbe:
  failureThreshold: 5 # Reessayer 5 fois
  httpGet:
    path: /
    port: 9090
    scheme: HTTP
  initialDelaySeconds: 30 # Attendre 30s avant de tester
  periodSeconds: 10 # Attendre 10s entre chaque essai
  timeoutSeconds: 5 # Attendre 5s la reponse
```

Ainsi, k8s sera capable de savoir si le conteneur fonctionne bien en appelant la route `/` . C'est une bonne pratique pour que Kubernetes sache quand redémarrer un pod.

- Ajoutons aussi des contraintes sur l'usage du CPU et de la RAM, en ajoutant à la même hauteur que `image:` :

```yaml
resources:
  requests:
    cpu: "100m"
```

```
            memory: "50Mi"
```

Nos pods auront alors **la garantie** de disposer d'un dixième de CPU et de 50 mégaoctets de RAM.

- Lancer `kubectl apply -f monstericon.yaml` pour appliquer.
- Avec `kubectl get pods --watch` , observons en direct la stratégie de déploiement `type: Recreate`
- Avec `kubectl describe deployment monstericon` , lisons les résultats de notre `readinessProbe` , ainsi que comment s'est passée la stratégie de déploiement `type: Recreate`

`monstericon.yaml` final :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: monstericon
  labels:
    app: monsterstack
spec:
  template:
    spec:
      containers:
      - name: monstericon
        image: tecpi/monster_icon:0.1
        ports:
        - containerPort: 9090
        readinessProbe:
          failureThreshold: 5 # Reessayer 5 fois
          httpGet:
            path: /
            port: 9090
            scheme: HTTP
          initialDelaySeconds: 30 # Attendre 30s avant de tester
          periodSeconds: 10 # Attendre 10s entre chaque essai
          timeoutSeconds: 5 # Attendre 5s la reponse
        resources:
          requests:
            cpu: "100m"
            memory: "50Mi"
    metadata:
      labels:
        app: monsterstack
        partie: monstericon
  selector:
    matchLabels:
      app: monsterstack
      partie: monstericon
  strategy:
    type: Recreate
  replicas: 5
```

Déploiement semblable pour dnmonster

Maintenant nous allons également créer un déploiement pour `dnmonster` :

- créez `dnmonster.yaml` et collez-y le code suivant :

`dnmonster.yaml` :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dnmonster
  labels:
    app: monsterstack
spec:
  selector:
    matchLabels:
      app: monsterstack
      partie: dnmonster
  strategy:
    type: Recreate
  replicas: 5
  template:
    metadata:
      labels:
        app: monsterstack
        partie: dnmonster
    spec:
      containers:
      - image: amouat/dnmonster:1.0
        name: dnmonster
        ports:
        - containerPort: 8080
```

Enfin, configurons un troisième deployment `redis` :

`redis.yaml` :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
  labels:
    app: monsterstack
spec:
  selector:
    matchLabels:
      app: monsterstack
      partie: redis
  strategy:
    type: Recreate
  replicas: 1
  template:
    metadata:
```

```
      labels:
        app: monsterstack
        partie: redis
    spec:
      containers:
      - image: redis:latest
        name: redis
        ports:
        - containerPort: 6379
```

Exposer notre stack avec des services

Les services K8s sont des endpoints réseaux qui balancent le trafic automatiquement vers un ensemble de pods désignés par certains labels.

Pour créer un objet `Service` , utilisons le code suivant, à compléter :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: <nom_service>
  labels:
    app: monsterstack
spec:
  ports:
    - port: <port>
  selector:
    app: <app_selector>
    partie: <tier_selector>
  type: <type>
---
```

Ajoutez le code suivant au début de chaque fichier déploiement. Complétez pour chaque partie de notre application : - le nom du service et le nom de la `partie` par le nom de notre programme (`monstericon` , `dnmonster` et `redis`) - le port par le port du service - les selectors `app` et `partie` par ceux du ReplicaSet correspondant.

Le type sera : `ClusterIP` pour `dnmonster` et `redis` , car ce sont des services qui n'ont à être accédés qu'en interne, et `LoadBalancer` pour `monstericon` .

Appliquez vos trois fichiers.

- Listez les services avec `kubectl get services` .
- Visitez votre application dans le navigateur avec `minikube service <nom-du-service-monstericon>` .

Rassemblons les trois objets avec une kustomisation.

Une kustomization permet de résumer un objet contenu dans de multiples fichiers en un seul lieu pour pouvoir le lancer facilement:

- Créez un dossier `monster_stack` pour ranger les trois fichiers:
    - monstericon.yaml
    - dnmonster.yaml
    - redis.yaml
- Créez également un fichier `kustomization.yaml` avec à l'intérieur:

```yaml
resources:
    - monstericon.yaml
    - dnmonster.yaml
    - redis.yaml
```

- Essayez d'exécuter la kustomization avec `kubectl apply -k .` depuis le dossier `monster_stack`.

Ajoutons un loadbalancer ingress pour exposer notre application sur le port standard

Installons le contrôleur Ingress Nginx avec `minikube addons enable ingress`.

Il s'agit d'une implémentation de loadbalancer dynamique basée sur nginx configurée pour s'interfacer avec un cluster k8s.

Ajoutez également l'objet de configuration du loadbalancer suivant dans le fichier `monster-ingress.yaml` :

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: monster-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
        - path: /monstericon
          backend:
            serviceName: monstericon
            servicePort: 9090
```

- Ajoutez ce fichier à notre `kustomization.yaml`
- Relancez la kustomization.

Vous pouvez normalement accéder à l'application en faisant `minikube service monstericon --url` et en ajoutant `/monstericon` pour y accéder.

Solution

Le dépôt Git des solutions est accessible ici : https://github.com/Uptime-Formation/tp2_k8s_monsterstack_correction

Rappels Docker

Les Dockerfiles

- Cours
- TP

Les volumes et les conteneurs

- Cours
- TP

Cours 6 - Objets Kubernetes - Partie 2

Objets k8s, suite

Le stockage dans Kubernetes

StorageClasses

Le stockage dans Kubernetes est fourni à travers des types de stockage appelés *StorageClasses* :

- dans le cloud, ce sont les différentes offres du fournisseur,
- dans un cluster auto-hébergé c'est par exemple :
  - un disque dur local ou distant (NFS)
  - ou bien une solution de stockage distribué
    - les plus connues sont Ceph et GlusterFS

PersistentVolumeClaims et PersistentVolumes

Quand un conteneur a besoin d'un volume, il crée une *PersistentVolumeClaim* : une demande de volume (persistant). Si un des objets *StorageClass* est en capacité de le fournir, alors un *PersistentVolume* est créé et lié à ce conteneur : il devient disponible en tant que volume monté dans le conteneur.

- les *StorageClasses* fournissent du stockage
- les conteneurs demandent du volume avec les *PersistentVolumeClaims*

- les *StorageClasses* répondent aux *PersistentVolumeClaims* en créant des objets *PersistentVolumes* : le conteneur peut accéder à son volume.

StatefulSets

On utilise les `Statefulsets` pour répliquer un ensemble de pods dont l'état est important : par exemple, des pods dont le rôle est d'être une base de données, manipulant des données sur un disque.

Un objet `StatefulSet` représente un ensemble de pods dotés d'identités uniques et de noms d'hôtes stables. Quand on supprime un StatefulSet, par défaut les volumes liés ne sont pas supprimés.

Les StatefulSets utilisent un nom en commun suivi de numéros qui se suivent. Par exemple, un StatefulSet nommé `web` comporte des pods nommés `web-0` , `web-1` et `web-2` . Par défaut, les pods StatefulSet sont déployés dans l'ordre et arrêtés dans l'ordre inverse (`web-2` , `web-1` puis `web-0`).

En général, on utilise des StatefulSets quand on veut :

- des identifiants réseau stables et uniques
- du stockage stable et persistant
- des déploiements et du scaling contrôlés et dans un ordre défini
- des rolling updates dans un ordre défini et automatisées

DaemonSets

Une autre raison de répliquer un ensemble de Pods est de programmer un seul Pod sur chaque nœud du cluster. En général, la motivation pour répliquer un Pod sur chaque nœud est de faire atterrir une sorte d'agent ou de démon sur chaque nœud, et l'objet Kubernetes pour y parvenir est le DaemonSet. Par exemple pour des besoins de monitoring, ou pour configurer le réseau sur chacun des nœuds.

Deployments, DaemonSets, StatefulSets

Étant donné les similitudes entre les DaemonSets, les StatefulSets et les Deployments, il est important de comprendre quand les utiliser.

- Les **Deployments** (liés à des ReplicaSets) doivent être utilisés :
  - lorsque votre application est complètement découplée du nœud
  - que vous pouvez en exécuter plusieurs copies sur un nœud donné sans considération particulière
  - que l'ordre de création des replicas et le nom des pods n'est pas important
  - lorsqu'on fait des opérations *stateless*
- Les **DaemonSets** doivent être utilisés :
  - lorsqu'au moins une copie de votre application doit être exécutée sur tous les nœuds du cluster (ou sur un sous-ensemble de ces nœuds).
- Les **StatefulSets** doivent être utilisés :

- lorsque l'ordre de création des replicas et le nom des pods est important
- lorsqu'on fait des opérations *stateful* (écrire dans une base de données)

Jobs

Les jobs sont utiles pour les choses que vous ne voulez faire qu'une seule fois, comme les migrations de bases de données ou les travaux par lots. Si vous exécutez une migration en tant que Pod normal, votre tâche de migration de base de données se déroulera en boucle, en repeuplant continuellement la base de données.

CronJobs

Comme des jobs, mais se lance à un intervalle régulier, comme avec `cron` .

Les ConfigMaps

D'après les recommandations de développement 12factor, la configuration de nos programmes doit venir de l'environnement. L'environnement est ici Kubernetes.

Les objets ConfigMaps permettent d'injecter dans des pods des fichiers de configuration en tant que volumes.

les Secrets

Les Secrets se manipulent comme des objets ConfigMaps, mais sont faits pour stocker des mots de passe, des clés privées, des certificats, des tokens, ou tout autre élément de config dont la confidentialité doit être préservée. Un secret se créé avec l'API Kubernetes, puis c'est au pod de demander à y avoir accès.

Il y a 3 façons de donner un accès à un secret :

- le secret est un fichier que l'on monte en tant que volume dans un conteneur (pas nécessairement disponible à l'ensemble du pod). Il est possible de ne jamais écrire ce secret sur le disque (volume `tmpfs`).
- le secret est une variable d'environnement du conteneur.
- cas spécifique aux registres : le secret est récupéré par kubelet quand il pull une image.

Pour définir qui et quelle app a accès à quel secret, on utilise les fonctionnalités dites "RBAC" de Kubernetes.

Le Role-Based Access Control, les Roles et les RoleBindings

Kubernetes intègre depuis quelques versions un système de permissions fines sur les ressources et les namespaces.

- Classiquement on crée des `Roles` comme `admin` ou `monitoring` qui désignent un ensemble de permission

- La logique de ce système de permissions est d'associer un **objet** (un type de ressource k8s) à un **verbe** (par exemple : `get` , `list` , `create` , `delete` ...)
- On crée ensuite des utilisateurs appelés `ServiceAccounts` dans k8s.
- On lie les *Roles* et *ServiceAccounts* à l'aide d'objets `RoleBindings` .

A côté des rôles crées pour les utilisateur·ices et processus du cluster, il existe des modèles de rôles prédéfinis qui sont affichables avec :

`kubectl get clusterroles`

La plupart de ces rôles intégrés sont destinés au `kube-system` , c'est-à-dire aux processus internes du cluster.

Cependant quatre rôles génériques existent aussi par défaut :

- Le rôle `cluster-admin` fournit un accès complet à l'ensemble du cluster.
- Le rôle `admin` fournit un accès complet à un espace de noms précis.
- Le rôle `edit` permet à un·e utilisateur·ice de modifier des choses dans un espace de noms.
- Le rôle `view` permet l'accès en lecture seule à un espace de noms.

La commande `kubectl auth can-i` permet de déterminer selon le profil utilisé (défini dans votre `kubeconfig`) les permissions actuelles de l'user sur les objets Kubernetes.

Les CRD et Operators

Les *CustomResourcesDefinition* sont l'objet le plus *méta* de Kubernetes : inventés par Red Hat pour ses *Operators*, ils permettent de définir un nouveau type d'objet dans Kubernetes. Combinés à des Operators (du code d'API en Go), ils permettent d'étendre Kubernetes pour gérer de nouveaux objets qui eux-même interagissent avec des objets Kubernetes.

Exemples :

- la chart officielle de la suite Elastic (ELK) définit des objets de type `elasticsearch`
- KubeVirt permet de rajouter des objets de type VM pour les piloter depuis Kubernetes
- Azure propose des objets correspondant à ses ressources du cloud Azure, pour pouvoir créer et paramétrer des ressources Azure directement via la logique de Kubernetes

Cours 7 - Helm, le gestionnaire de paquets Kubernetes

Nous avons vu que dans Kubernetes la configuration de nos services / applications se fait généralement via de multiples fichiers YAML.

Les fichiers kustomization

Il est courant de décrire un ensemble de resources dans le même fichier, séparées par `---` . Mais on pourrait préférer rassembler plusieurs fichiers dans un même dossier et les appliquer d'un coup.

Pour cela K8s propose le concept de `kustomization` .

Exemple:

```
k8s-mysql/
├── kustomization.yaml
├── mysql-deployment.yaml
└── wordpress-deployment.yaml
```

`kustomization.yaml`

```yaml
secretGenerator:
  - name: mysql-pass
    literals:
      - password=YOUR_PASSWORD
resources:
  - mysql-deployment.yaml
  - wordpress-deployment.yaml
```

On peut ensuite l'appliquer avec `kubectl apply -k ./`

A noter que `kubectl kustomize .` permet de visualiser l'ensemble des modifications avant de les appliquer (`kubectl kustomize . | less` pour mieux lire).

Helm

Quand on a une seule application cela reste gérable avec des kustomizations ou sans, mais dès qu'on a plusieurs environnements, applications et services, on se retrouve vite submergé·es de fichiers de centaines, voire de milliers, de lignes qui sont, de plus, assez semblables. C'est donc "trop" déclaratif, et il faut se concentrer sur les quelques propriétés que l'on souhaite créer ou modifier,

Pour pallier ce problème, il existe l'utilitaire Helm, qui produit les fichiers de déploiement que l'on souhaite.

Helm est le package manager recommandé par Kubernetes, il utilise les fonctionnalités de templating du langage Go.

Helm permet donc de déployer des applications / stacks complètes en utilisant un système de templating et de dépendances, ce qui permet d'éviter la duplication et d'avoir ainsi une arborescence cohérente pour nos fichiers de configuration.

Mais Helm propose également :

- la possibilité de mettre les Charts dans un répertoire distant (Git, disque local ou partagé…), et donc de distribuer ces Charts publiquement.
- un système facilitant les Updates et Rollbacks de vos applications.

Il existe des sortes de *stores* d'applications Kubernetes packagées avec Helm, le plus gros d'entre eux est Kubeapps Hub, maintenu par l'entreprise Bitnami qui fournit de nombreuses Charts assez robustes.

Si vous connaissez Ansible, un chart Helm est un peu l'équivalent d'un rôle Ansible dans l'écosystème Kubernetes.

Concepts

Les quelques concepts centraux de Helm :

- Un package Kubernetes est appelé **Chart** dans Helm.
- Un Chart contient un lot d'informations nécessaires pour créer une application Kubernetes :
  - la **Config** contient les informations dynamiques concernant la configuration d'une **Chart**
  - Une **Release** est une instance existante sur le cluster, combinée avec une **Config** spécifique.

Architecture client-serveur de Helm

Helm désigne une application client en ligne de commande.

Pour fonctionner sur le cluster, Helm a besoin d'installer un gestionnaire appelé Tiller : c'est le serveur qui communique avec le client Helm et l'API de Kubernetes pour gérer vos déploiements.

Lors de l'initialisation de Helm, le client installe Tiller sur un pod du cluster.

Helm utilise automatiquement votre fichier `kubeconfig` pour se connecter.

Quelques commandes Helm:

Voici quelques commandes de bases pour Helm :

- `helm repo add bitnami https://charts.bitnami.com/bitnami` : ajouter un repo contenant des charts
- `helm search repo bitnami` : rechercher un chart en particulier
- `helm install my-chart` : permet d'installer le chart my-chart. Le nom de release est généré aléatoirement dans votre cluster Kubernetes.
- `helm upgrade my-release my-chart` : permet de mettre à jour notre release avec une nouvelle version.
- `helm ls` : Permet de lister les Charts installés sur votre Cluster
- `helm delete my-release` : Permet de désinstaller la release `my-release` de Kubernetes

La configuration d'un Chart: des templates d'objets Kubernetes

Visitons un exemple de Chart : minecraft

On constate que Helm rassemble des fichiers de descriptions d'objets k8s avec des variables (moteur de templates de Go) à l'intérieur, ce qui permet de factoriser le code et de gérer puissamment la différence entre les versions.

TP 4 - Déployer Wordpress avec Helm

Helm est un gestionnaire de paquet k8s qui permet d'installer des paquets sans faire des copier-coller pénibles de YAML :

- pas de duplication de code
- des déploiements avancés avec un processus de mise à jour k8s intégré

Helm ne dispense pas de maîtriser l'administration de son cluster.

Installer Helm

- Pour installer Helm sur Ubuntu, utilisez : `snap install helm --classic`
- Suivez le Quickstart : https://helm.sh/docs/intro/quickstart/

Utiliser une chart Helm pour installer Wordpress

- Cherchez Wordpress sur https://hub.kubeapps.com (vous pouvez prendre une autre chart si le cœur vous en dit).
- Prenez la version de **Bitnami** et ajoutez le dépôt avec la première commande à droite (ajouter le dépôt et déployer une release).
- Installer une release `wordpress-tp` de cette application (ce chart) avec `helm install --template-name wordpress-tp bitnami/wordpress`
- Suivez les instructions affichées

- Notre Wordpress est prêt. Connectez-vous-y avec les identifiants affichés (il faut passer les commandes indiquées pour récupérer le mot de passe stocké dans un secret k8sen).
- Explorez les différents objets k8s créés par Helm avec Lens.

TP 5 - Cloud Azure

Nous allons déployer une application dans Azure à l'aide de *charts* Helm : https://docs.bitnami.com/kubernetes/get-started-aks/

Créer un cluster AKS

Configurer l'environnement Azure

Tout d'abord, il faut se créer un compte Azure. Si c'est la première fois, du crédit gratuit est disponible : https://azure.microsoft.com/fr-fr/free/ Ensuite on peut utiliser le Cloud Shell Azure ou n'importe quel terminal.

```
# Install Azure CLI
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

# Login
az login --allow-no-subscriptions

# Créer le groupe de ressources
az group create --name aks-resource-group --location westeurope
```

Au préalable, installer `kubectl` (pas besoin dans le Cloud Shell) :

```
snap install kubectl --classic
```

Créer le cluster K8S

```
# Créer deux nœuds dans le cluster AKS
az aks create --name aks-cluster --resource-group aks-resource-group --nod

# Récupérer la config AKS
az aks get-credentials --name aks-cluster --resource-group aks-resource-gr
```

Créer le registry pour les images Docker

Pour créer le registry, il faut choisir un nom unique, remplacez `pommedeterrepoirekiwi` par un autre nom.

```
# Créer le registry
az acr create --resource-group aks-resource-group  --name pommedeterrepoire
az acr login --name pommedeterrepoirekiwi

# Créer un compte sur le registry pour K8S
ACR_LOGIN_SERVER=$(az acr show --name pommedeterrepoirekiwi --query loginS
ACR_REGISTRY_ID=$(az acr show --name pommedeterrepoirekiwi --query id --ou
SP_PASSWD=$(az ad sp create-for-rbac --name k8s-read-registry --role Reade
CLIENT_ID=$(az ad sp show --id http://k8s-read-registry --query appId --ou
kubectl create secret docker-registry read-registry-account \
--docker-server $ACR_LOGIN_SERVER \
--docker-username $CLIENT_ID \
--docker-password $SP_PASSWD \
--docker-email cto@example.org
```

Pousser une image sur son registry Azure

Pour installer Docker : `curl -sSL https://get.docker.com | sudo sh`

```
# Récupérer puis pousser une image sur son registry Azure
docker pull docker.io/bitnami/wordpress:latest
docker tag docker.io/bitnami/wordpress:latest pommedeterrepoirekiwi.azurec
docker push pommedeterrepoirekiwi.azurecr.io/bitnami/wordpress:latest
```

Appliquer une chart Helm

Pour installer Helm : `curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash`

```
# Ajout de la chart
helm repo add bitnami https://charts.bitnami.com/bitnami

# Installer la chart
helm install wordpress bitnami/wordpress \
--set serviceType=LoadBalancer \
--set image.registry="pommedeterrepoirekiwi.azurecr.io" \
--set image.pullSecrets={read-registry-account} \
--set image.repository=bitnami/wordpress \
--set image.tag=latest
```

Des messages s'affichent suite à l'application de la chart Helm, suivez les instructions pour accéder au Wordpress.

Documentation

Scaling d'application dans Azure

- https://docs.microsoft.com/fr-fr/azure/aks/tutorial-kubernetes-scale

Stockage dans Azure

- https://docs.microsoft.com/fr-fr/azure/aks/azure-files-dynamic-pv

Registry dans Azure

- https://docs.microsoft.com/fr-fr/azure/container-registry/container-registry-quickstart-task-cli

Terraform avec Azure

Terraform est un outil permettant de décrire des ressources cloud dans un fichier pour utiliser le concept d'infrastructure-as-code avec tous les objets des fournisseurs de Cloud.

- https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/kubernetes_cluster
- https://github.com/terraform-providers/terraform-provider-azurerm/tree/master/examples/kubernetes
- https://registry.terraform.io/modules/Azure/appgw-ingress-k8s-cluster/azurerm/latest
- https://docs.microsoft.com/fr-fr/azure/aks/ingress-basic#create-an-ingress-controller

Le réseau dans Azure

- Vidéo "K8s Networking in Azure" : https://www.youtube.com/watch?v=JyLtg_SJ1lo&list=PLoWxE_5hnZUZMWrEON3wxMBoIZvweGeiq&index=2
- https://docs.microsoft.com/fr-fr/azure/aks/internal-lb
- https://docs.microsoft.com/fr-fr/azure/aks/load-balancer-standard
- https://docs.microsoft.com/fr-fr/azure/aks/http-application-routing
- https://docs.microsoft.com/fr-fr/azure/aks/concepts-network
- https://blog.crossplane.io/azure-secure-connectivity-for-aks-azure-db/
- https://docs.microsoft.com/fr-fr/azure/mysql/concepts-aks

Pour aller plus loin

- https://docs.microsoft.com/fr-fr/azure/aks/
- https://github.com/microsoft/kubernetes-learning-path

Les CRD : utiliser des objets Kubernetes pour définir des ressources Azure

https://github.com/Azure/azure-service-operator

Autres idées d'exercices

- Basique : https://docs.microsoft.com/fr-fr/azure/aks/kubernetes-walkthrough
- Difficile : https://github.com/Microsoft/RockPaperScissorsLizardSpock

TD opt. - StatefulSets et bases de données

Avec la chart PostgreSQL HA

En lançant la chart PostgreSQL HA de Bitnami, et en lisant les logs des conteneurs, observez comment fonctionne les StatefulSets, par exemple avec Lens. Scalez les StatefulSets postgres.

Facultatif : A la main, avec MySQL, des init containers et des ConfigMaps

- Suivre ce tutoriel pas à pas : https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/

TP opt. - Les ingresses

Ressources sur les ingresses

Minikube

https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/

Azure AKS

https://docs.microsoft.com/fr-fr/azure/aks/ingress-basic

Scaleway (avec Traefik)

https://www.scaleway.com/en/docs/using-a-load-balancer-to-expose-your-kubernetes-kapsule-ingress-controller-service/

DNS

Pour les DNS, 3 solutions :

- en local, éditer `/etc/hosts`
- sur Internet, ne pas l'utiliser et faire un Ingress avec l'adresse IP comme hostname
- sur Internet, utiliser https://netlib.re pour configurer un DNS avec un domaine en `netlib.re`

TP opt. - Le RBAC

Les rôles et le RBAC

1. Configurer Minikube pour activer RBAC.

```
minikube start --extra-config=apiserver.Authorization.Mode=RBAC

kubectl create clusterrolebinding add-on-cluster-admin --clusterrole=clust
```

2. Créer trois connexions à minikube dans `~/.kube/config` :

- une en mode `cluster-admin` ,
- une en mode admin sur un namespace
- et une en mode user avec un `rolebinding`

3. En switchant de contexte à chaque fois, lancer la commande `kubectl auth can-i` pour différents cas et observer la différence

Ressources

- https://medium.com/@HoussemDellai/rbac-with-kubernetes-in-minikube-4deed658ea7b
- https://docs.bitnami.com/tutorials/configure-rbac-in-your-kubernetes-cluster/

Conclusion

Points forts de Kubernetes

- Open source et très actif.
- Une communauté très visible et présente dans l'évolution de l'informatique.
- Un standard collectif qui permet une certaine interopérabilité dans le cloud.
- Les *pods* tendent à se rapprocher plus d'une VM du point de vue de l'application.

- Hébergeable de façon quasi-identique dans le cloud, on-premise ou en mixte.
- Kubernetes a un *flat network* ce qui permet de faire des choses puissante facilement comme le multi-datacenter.
- K8s est pensé pour la *scalabilité* et le *calcul distribué*.

Faiblesses de Kubernetes

- Une difficulté à manier tout ce qui est *stateful*, comme des bases de données
  - …même si les Operators et les CRD (Custom Resources Definitions) permettent de combler cette lacune dans la logique *stateless* de k8s
- Beaucoup de points sont laissés à la décision du fournisseur de cloud ou des admins système :
  - Pas de solution de **stockage** par défaut, et parfois difficile de stocker "simplement" sans passer par les fournisseurs de cloud, ou par une solution de stockage décentralisé à part entière (**Ceph, Gluster, Longhorn**…)
    - …même si ces solutions sont souvent bien intégrées à k8s
  - Beaucoup de solutions de **réseau** qui se concurrencent, demandant un comparatif fastidieux
    - …même si plusieurs leaders émergent comme **Calico, Flannel, Weave ou Cilium**
  - Pas de solution de loadbalancing par défaut : soit on se base sur le fournisseur de cloud, soit on configure *MetalLB* –>

Pour approfondir

Monitoring et logging

Avec Prometheus et la suite Elastic.

Déploiement continu

- Exemple de workflow de déploiement continu (CD)
  - par exemple avec Gitlab (possiblement auto-hébergé dans K8s)
  - se connecter à un bastion
  - `git pull`
  - puis `kubectl apply`

Exemple de stack avancée

La Bitnami Kubernetes Production Runtime (BKPR).

- Monitoring avec Prometheus et Grafana
- Logging avec Elasticsearch, Kibana et Fluentd

- HTTPS ingress avec Nginx, ExternalDNS, Cert-Manager et oauth2_proxy

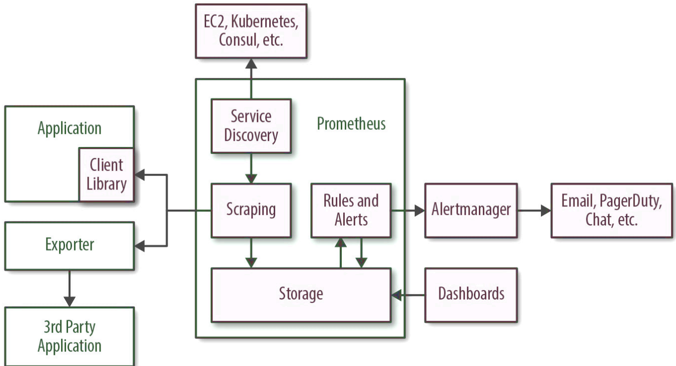
TP7 - Stratégies de déploiement et monitoring

Installer Prometheus pour monitorer le cluster Minikube

Pour comprendre les stratégies de déploiement et mise à jour d'application dans Kubernetes (deployment and rollout strategies) nous allons installer puis mettre à jour une application d'exemple et observer comment sont gérées les requêtes vers notre application en fonction de la stratégie de déploiement choisie.

Pour cette observation on peut utiliser un outil de monitoring. Nous utiliserons ce TP comme prétexte pour installer une des stack les plus populaires et intégrée avec kubernetes : Prometheus et Grafana. Prometheus est un projet de la Cloud Native Computing Foundation.

Prometheus est un serveur de métriques c'est à dire qu'il enregistre des informations précises (de petite taille) sur différents aspects d'un système informatique et ce de façon périodique en effectuant généralement des requêtes vers les composants du système (metrics scraping).



Installer Prometheus avec Helm

Installez Helm si ce n'est pas déjà fait. Sur Ubuntu : `sudo snap install helm --classic`

- Créons un namespace pour prometheus et grafana : `kubectl create namespace monitoring`
- Ajoutez le dépot de chart **Prometheus** et **kube-state-metrics**: `helm repo add prometheus-community https://prometheus-community.github.io/helm-charts` puis `helm repo add kube-state-metrics https://kubernetes.github.io/kube-state-metrics` puis mise à jours des dépots helm `helm repo update` .
- Installez ensuite le chart prometheus :

```
helm install \
  --namespace=monitoring \
  --version=13.2.1 \
  --set=service.type=NodePort \
  prometheus \
  prometheus-community/prometheus
```

kube-state-metrics et le monitoring du cluster

Le chart officiel installe par défaut en plus de Prometheus, kube-state-metrics qui est une intégration automatique de kubernetes et prometheus.

Une fois le chart installé vous pouvez visualisez les informations dans Lens, dans la premiere section du menu de gauche `Cluster` .

Déployer notre application d'exemple (goprom) et la connecter à prometheus

Nous allons installer une petite application d'exemple en go.

- Téléchargez le code de l'application et de son déploiement depuis github: `git clone https://github.com/e-lie/k8s-deployment-strategies`

Nous allons d'abord construire l'image docker de l'application à partir des sources. Cette image doit être stockée dans le registry de minikube pour pouvoir être ensuite déployée dans le cluster. En mode développement Minikube s'interface de façon très fluide avec la ligne de commande Docker grace à quelques variable d'environnement : `minikube docker-env`

- Changez le contexte de docker cli pour pointer vers minikube avec `eval` et la commande précédente.

réponse:

- Allez dans le dossier `goprom_app` et "construisez" l'image docker de l'application avec le tag `uptime-formation/goprom` .

réponse:

- Allez dans le dossier de la première stratégie `recreate` et ouvrez le fichier `app-v1.yml` . Notez que `image:` est à `uptime-formation/goprom` et qu'un paramètre `imagePullPolicy` est défini à `Never` . Ainsi l'image sera récupéré dans le registry local du docker de minikube ou sont stockées les images buildées localement plutôt que récupéré depuis un registry distant.
- Appliquez ce déploiement kubernetes:

réponse:

Observons notre application et son déploiement kubernetes

- Explorez le fichier de code go de l’application `main.go` ainsi que le fichier de déploiement `app-v1.yml`. Quelles sont les routes http exposées par l’application ?

réponse:

- Faites un forwarding de port `Minikube` pour accéder au service `goprom` dans votre navigateur.

réponse:

- Faites un forwarding de port pour accéder au service `goprom-metrics` dans votre navigateur. Quelles informations récupère-t-on sur cette route ?

réponse:

- Pour tester le service `prometheus-server` nous avons besoin de le mettre en mode NodePort (et non ClusterIP par défaut). Modifiez le service dans Lens pour changer son type.
- Exposez le service avec Minikube (n’oubliez pas de préciser le namespace monitoring).
- Vérifiez que prometheus récupère bien les métriques de l’application avec la requête PromQL : `sum(rate(http_requests_total{app="goprom"}[5m])) by (version)`.
- Quelle est la section des fichiers de déploiement qui indique à prometheus ou récupérer les métriques ?

réponse:

Installer et configurer Grafana pour visualiser les requêtes

Grafana est une interface de dashboard de monitoring facilement intégrable avec Prometheus. Elle va nous permettre d’afficher un histogramme en temps réel du nombre de requêtes vers l’application.

Créez un secret Kubernetes pour stocker le loging admin de grafana.

```
cat <<EOF | kubectl apply -n monitoring -f -
apiVersion: v1
kind: Secret
metadata:
  namespace: monitoring
  name: grafana-auth
type: Opaque
data:
  admin-user: $(echo -n "admin" | base64 -w0)
  admin-password: $(echo -n "admin" | base64 -w0)
EOF
```

Ensuite, installez le chart Grafana en précisant quelques paramètres:

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
```

```
helm install \
  --namespace=monitoring \
  --version=6.1.17 \
  --set=admin.existingSecret=grafana-auth \
  --set=service.type=NodePort \
  --set=service.nodePort=32001 \
  grafana \
  grafana/grafana
```

Maintenant Grafana est installé vous pouvez y acccéder en forwardant le port du service grace à Minikube:

```
$ minikube service grafana
```

Pour vous connectez utilisez, username: `admin` , password: `admin` .

Il faut ensuite connecter Grafana à Prometheus, pour ce faire ajoutez une `DataSource` :

```
Name: prometheus
Type: Prometheus
Url: http://prometheus-server
Access: Server
```

Créer une dashboard avec un Graphe. Utilisez la requête prometheus (champ query suivante):

```
sum(rate(http_requests_total{app="goprom"}[5m])) by (version)
```

Pour avoir un meilleur aperçu de la version de l'application accédée au fur et à mesure du déploiement, ajoutez `{{version}}` dans le champ `legend` .

Observer un basculement de version

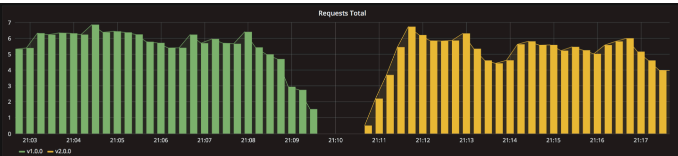
Ce TP est basé sur l'article suivant: https://blog.container-solutions.com/kubernetes-deployment-strategies

Maintenant que l'environnement a été configuré :

- Lisez l'article.
- Vous pouvez testez les différentes stratégies de déploiement en lisant leur `README.md` .
- En résumé, pour les plus simple, on peut:
  - appliquer le fichier `app-v1.yml` pour une stratégie.
  - lançer la commande suivante pour effectuer des requêtes régulières sur l'application: `service=$(minikube service goprom --url) ; while sleep 0.1; do curl "$service"; done`
  - Dans un second terminal (pendant que les requêtes tournent) appliquer le fichier `app-v2.yml` correspondant.

  - Observez la réponse aux requêtes dans le terminal ou avec un graphique adapté dans `graphana` (Il faut configurer correctement le graphique pour observer de façon lisible la transition entre v1 et v2). Un aperçu en image des histogrammes du nombre de requêtes en fonction des versions 1 et 2 est disponible dans chaque dossier de stratégie.
  - supprimez le déploiement+service avec `delete -f` ou dans Lens.

Par exemple pour la stratégie **recreate** le graphique donne:



Exporter les supports en pdf

Pour exporter correctement les TPs et autres pages de ce site au format pdf, utilisez la fonction `imprimer` de Google Chrome ou Firefox (vous pouvez aussi activer le Mode Lecture de Firefox en cliquant Affichage > Passer en Mode Lecture) en ouvrant la page suivante :

Contenu intégral

Contenu intégral

Tutoriels utiles

Vous trouverez ici quelques tutoriels qui peuvent être utiles dans le cadre des formations.

- Redimensionner le disque d'une machine virtualbox
- Traduire des documents

Redimensionner le disque d'une machine

virtualbox

- Bien éteindre la machine.
- Sur le système hôte (Windows ou Linux):
    - Téléchargez `gparted` : lien sourceforge
- Pour redimensionner le disque sur Windows:
    - Ouvrir une invite de commande.
    - Visitez `C:\Users\<votre_user>\Virtualbox VMs\<votre_machine>\`

```
# la première ligne est utile seulement si le disque est au format vdmk
"C:\Program Files\Oracle\VirtualBox\VBoxManage" clonemedium "<votre_disque
"C:\Program Files\Oracle\VirtualBox\VBoxManage" modifymedium "cloned.vdi" -
```

- Pour redimensionner le disque sur linux:
    - Dans un terminal, visitez `"~/Virtualbox VMs"` , entrez dans le dossier de la machine en question.

```
# la première ligne est utile seulement si le disque est au format vdmk
VBoxManage clonemedium "<votre_disque>.vmdk" "<votre_disque>.vdi" --format
VBoxManage modifymedium "cloned.vdi" --resize 20480 # 20Gio par exemple.
```

- Allez dans la configuration de la machine, déconnectez le disque VMDK et connectez le nouveau disque VDI.
- Ajoutez le CD gparted à la machine.
- Lancez la machine.
- Gparted demarre. choisissez le type de clavier (fr) puis le lancement avec serveur X (option 0).
- Une fenêtre avec votre disque s'affiche. Cliquez sur le disque dans la liste puis "redimensionner/déplacer"
- Dans la fenêtre suivante, agrandissez à la souris la partition pour occuper tout l'espace disponible.
- Faites `ok` puis `appliquer` .

Traduire des documents

Pour l'anglais, si un texte ne vous paraît pas clair, quelques liens :

- Pour les textes : https://www.deepl.com/translator

- Pour les pages web : https://translate.google.com/
- Pour les mots : https://linguee.fr/

Bibliographie

Linux

- Shotts 2012 - The Linux Command Line - A complete introduction
- Yao 2014 - Linux command line - A beginner's guide

Ressources

- OpenClassrooms - Reprenez le contrôle à l'aide de Linux !
  https://openclassrooms.com/fr/courses/43538-reprenez-le-controle-a-l-aide-de-linux

Ansible

- Jeff Geerling - Ansible for DevOps - Leanpub

Pour aller plus loin

- Keating2017 - Mastering Ansible - Second Edition - Packt

Cheatsheet

- https://www.digitalocean.com/community/cheatsheets/how-to-use-ansible-cheat-sheet-guide

Docker

- McKendrick, Gallagher 2017 Mastering Docker - Second Edition

Pour aller plus loin

- Miell,Sayers2019 - Docker in Practice

Cheatsheet

- https://devhints.io/docker

Ressources

- Doc officielle : https://docs.docker.com/

- Référence officielle : https://docs.docker.com/reference/
- Awesome Docker, liste de ressources sur Docker : https://github.com/veggiemonk/awesome-docker
- Portainer, GUI pour Docker : https://www.portainer.io/installation/
- Lazy Docker, terminal pour Docker : https://github.com/jesseduffield/lazydocker
- Convoy, driver pour volumes Docker : https://github.com/rancher/convoy
- Accéder à ses containers dans Minecraft : https://github.com/docker/dockercraft
- Doc officielle sur la sécurité dans Docker : https://docs.docker.com/engine/security/
- Documentation sur le système de filesystem overlay de Docker : https://docs.docker.com/storage/storagedriver/overlayfs-driver/
- Tutoriels officiels sur la sécurité dans Docker : https://github.com/docker/labs/tree/master/security
- Vidéo sur les bonnes pratiques dans Docker : https://noti.st/aurelievache/PrttUh
- Vidéo "Containers, VMs... Comment ces technologies fonctionnent et comment les différencier ?" (Quentin Adam) https://www.youtube.com/watch?v=wG4_JQXvZIc
- Diapositives sur Docker, Swarm, Kubernetes : https://container.training/
    - en particulier les prolèmes du stateful : https://container.training/swarm-selfpaced.yml.html#450

Kubernetes

- Kubernetes Up and Running, O'Reilly 2019

Ressources

- Awesome Kubernetes

Réseau

- Documentation officielle : https://kubernetes.io/fr/docs/concepts/services-networking/service/
- An introduction to service meshes - DigitalOcean
- Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?
- Determine best networking option - Project Calico
- Doc officielle sur les solutions de networking
- Comparatif de solutions réseaux (fr) : https://www.objectif-libre.com/fr/blog/2018/07/05/comparatif-solutions-reseaux-kubernetes/#Flannel
- Comparatif de solutions réseaux (en) :https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/

Vidéos sur le réseau

Des vidéos assez complètes sur le réseau, faites par Calico :

- Kubernetes Ingress networking
- Kubernetes Services networking
- Kubernetes networking on Azure

Sur MetalLB, les autres vidéos de la chaîne sont très bien :

- Why you need to use MetalLB - Adrian Goins

Stockage

- Rook et Ceph (fr)
- Longhorn

Sécurité de Kubernetes

- [xmco-actusecu-51-dossier_kubernetes (fr)](http://repository.root-me.org/Exploitation - Système/FR - xmco-actusecu-51-dossier_kubernetes.pdf)
- [hacking_and_hardening_kubernetes_by_example_v2 (en)](http://repository.root-me.org/Exploitation - Système/EN - hacking_and_hardening_kubernetes_by_example_v2.pdf)
- [ht-w02_hacking_and_hardening_kubernetes (en)](http://repository.root-me.org/Exploitation - Système/EN - ht-w02_hacking_and_hardening_kubernetes.pdf)

Autres

- Bitnami Helm : https://github.com/bitnami/charts/tree/master/bitnami
- BKPR : https://github.com/bitnami/kube-prod-runtime
- Vitess : A database clustering system for horizontal scaling of MySQL : https://vitess.io
- Rancher
- Charts Helm : https://hub.kubeapps.com
- Stratégies de déploiement : https://blog.container-solutions.com/kubernetes-deployment-strategies

Azure AKS

Documentation

- https://docs.microsoft.com/fr-fr/azure/aks/
- https://github.com/microsoft/kubernetes-learning-path

Scaling d'application dans Azure

- https://docs.microsoft.com/fr-fr/azure/aks/tutorial-kubernetes-scale

Stockage dans Azure

- https://docs.microsoft.com/fr-fr/azure/aks/azure-files-dynamic-pv

Registry dans Azure

- https://docs.microsoft.com/fr-fr/azure/container-registry/container-registry-quickstart-task-cli

Le réseau dans Azure

- Vidéo "K8s Networking in Azure" : https://www.youtube.com/watch?v=JyLtg_SJ1lo&list=PLoWxE_5hnZUZMWrEON3wxMBoIZvweGeiq&index=2

- https://docs.microsoft.com/fr-fr/azure/aks/internal-lb
- https://docs.microsoft.com/fr-fr/azure/aks/load-balancer-standard
- https://docs.microsoft.com/fr-fr/azure/aks/http-application-routing
- https://docs.microsoft.com/fr-fr/azure/aks/concepts-network
- https://blog.crossplane.io/azure-secure-connectivity-for-aks-azure-db/
- https://docs.microsoft.com/fr-fr/azure/mysql/concepts-aks

Terraform avec Azure

Terraform est un outil permettant de décrire des ressources cloud dans un fichier pour utiliser le concept d'infrastructure-as-code avec tous les objets des fournisseurs de Cloud.

- https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/kubernetes_cluster
- https://github.com/terraform-providers/terraform-provider-azurerm/tree/master/examples/kubernetes
- https://registry.terraform.io/modules/Azure/appgw-ingress-k8s-cluster/azurerm/latest
- https://docs.microsoft.com/fr-fr/azure/aks/ingress-basic#create-an-ingress-controller

Autres

- Les CRD : utiliser des objets Kubernetes pour définir des ressources Azure : https://github.com/Azure/azure-service-operator
- Demo : https://github.com/Microsoft/RockPaperScissorsLizardSpock

Pour aller plus loin

- Luksa, Kubernetes in Action, 2018

Cheatsheets

- https://kubernetes.io/fr/docs/reference/kubectl/cheatsheet/
- Short names in k8s : https://blog.heptio.com/kubectl-resource-short-names-heptioprotip-c8eff9fb7202

DevOps

- Krief - Learning DevOps - The complete guide (Azure Devops, Jenkins, Kubernetes, Terraform, Ansible, sécurité) - 2019
- The DevOps Handbook