**Docker**

1. Isolation et virtualisation

1. Principe de l'isolation

L'isolation est un concept fondamental dans les architectures micro-services et plus généralement en programmation. Il s'agit d'un ensemble de techniques consistant à isoler des processus entre eux et/ou de la machine hôte.

**Prenons un exemple:**  
Une entreprise possède un serveur de production, hébergé dans le cloud ou non, sur lequel sont déployés les différents processus créés dans l'entreprise. Ces processus peuvent être des applications de Data Science ou non. Les développeurs, Data Scientists et autres acteurs de l'entreprise vont créer des applications, des routines, des processus dans des environnements de développement (en général, ordinateur personnel).

Il peut déjà se présenter un premier problème: comment s'assurer que le fonctionnement de l'application soit le même sur le serveur de production que dans l'environnement de développement? En effet, l'application requiert certaines librairies, certains outils et surtout certaines versions de ces outils pour fonctionner. Il est donc important d'embarquer l'environnement dans lequel l'application fonctionne localement.

On peut ensuite rencontrer un second problème lorsque plusieurs applications sont en production sur le même serveur: ces applications peuvent avoir des besoins différents en outil mais surtout des besoins contradictoires. Par exemple, une application utilisant Python pourrait ne fonctionner qu'avec la version 1.16.1 de la librairie numpy alors qu'une autre nécessiterait une version plus récente. Ces conflits peuvent se révéler problématiques si on ne veut pas uniformiser les différentes versions de nos applications (ce qui demanderait un effort de développement et de mise à jour considérable).

Il faut donc trouver une manière d'isoler les différentes applications. On peut alors se représenter cette notion d'isolation en imaginant des boîtes dans lesquelles les applications sont enfermées, isolées du reste de la machine et d'autres applications et dans lesquelles elles peuvent trouver toutes les ressources nécessaires pour fonctionner correctement.

Attention, l'isolation n'a pas à être totale: dans certains cas, on a besoin de communiquer avec ou entre les applications: l'isolation se fait donc de manière contrôlée et non hermétique.

2 . Isolation par virtualisation

Il existe différentes façons d'isoler des processus. Une des techniques les plus utilisées est la virtualisation: il s'agit de créer un environnement qui cache les véritables fonctionnalités, outils et ressources de la machine hôte. L'application ne voit plus alors les ressources offertes par la machine hôte de manière générique mais des ressources supplémentaires.

Plaçons-nous du point de vue d'une application dans sa "boîte". L'application ne voit pas qu'elle est dans un environnement qui n'est pas un environnement réel: elle a accès à toutes les ressources dont elle a besoin (mémoire, CPU, disque, ...) même si ces ressources sont en réalité traitées par des intermédiaires. De plus, elle ne voit que les outils qui sont compris dans la boîte et opère comme si elle était seule sur la machine.

Il existe différents dispositifs de virtualisation:

* Machines virtuelles (locales ou hébergées dans le cloud)
* Conteneurs
* Environnements virtuels

Une machine virtuelle est en fait une émulation d'une machine complète: un ordinateur dans l'ordinateur. Une machine virtuelle possède son propre système d'exploitation, des ressources (mémoire et disque) allouées. Par exemple, sur un ordinateur personnel dont le système d'exploitation est Windows, on peut créer une machine virtuelle avec le système d'exploitation Ubuntu. Ainsi, le fonctionnement des applications dans cette machine virtuelle n'est pas concerné par le système d'exploitation Windows. Elles fonctionnent comme si elles étaient sur une machine hôte fournie avec Ubuntu.

Développer dans une machine virtuelle permet donc d'isoler les processus de la machine hôte. Si on souhaite alors déployer l'application sur un serveur de production, il suffit de déployer l'application ainsi que la machine virtuelle.

Aujourd'hui, on utilise facilement des machines virtuelles hébergées dans le Cloud: les fournisseurs de Cloud permettent en effet d'utiliser des machines hébergées sur des serveurs. Les ressources de ces serveurs sont partagées par différents utilisateurs, souvent d'organismes différents. L'utilisation de machines virtuelles permet aux utilisateurs de ne pas voir les autres utilisateurs tout en partageant le même matériel physique.

Notons que ces fournisseurs de Cloud fournissent des patrons de machines virtuelles bien définies: certaines versions de Ubuntu ou de Windows par exemple. Ces patrons permettent de facilement créer des environnements de production similaires aux environnements de développement.

Mais ces machines comportent un aspect problématique: comme elles utilisent un système d'exploitation, elles sont gourmandes en ressources. Il faut en effet des ressources pour faire fonctionner le système d'exploitation.

Le container est une autre technologie de virtualisation. Si le principe est similaire à celui de la machine virtuelle, elle n'embarque pas de système d'exploitation. Les containers sont donc plus légers, à la fois du point de vue des ressources d'exécution ainsi que du point de vue de la taille des images. On étudiera dans ce cours l'utilisation des containers Docker.

Enfin, on pourrait parler d'une technologie propre à Python: les environnements virtuels: pour exécuter un script Python, nous avons besoin de deux choses, à savoir un interpréteur, c'est-à-dire une version de Python, et un répertoire dans lequel sont stockées les fichiers sources des librairies utilisées. Utiliser un environnement virtuel consiste à utiliser un répertoire spécifique où sont stockées ces librairies. Ainsi L'environnement virtuel permet d'avoir plusieurs versions d'une même librairie installées sur une même machine et utilisées par différentes applications.

Nous pouvons aussi noter une chose assez intéressante: toutes les solutions proposant d'utiliser ces technologies de virtualisation proposent des systèmes d'images à reproduire (celles-ci seront détaillées dans la prochaine partie). On a, un peu comme avec la programmation orientée objet, un système de modèle que l'on peut choisir d'instancier à volonté. Pour les fournisseurs de Cloud, c'est le catalogue de machines pré-paramétrées (AMIs pour AWS). Pour les containers, ce sont les images Docker pré-existantes ainsi que les DockerFiles. Dans le cas des environnements virtuels, on trouve les fichiers requirements.txt généralement utilisés.

Ainsi la virtualisation permet à la fois d'isoler des processus entre eux, de les isoler de la machine hôte mais aussi d'instancier facilement, rapidement des dispositifs virtuels identiques entre eux: on peut facilement reproduire un environnement virtuel, une machine virtuelle ou un container.

## 2. Docker en théorie

Docker est un des outils les plus populaires et les plus utilisés de containerisation.

1. Technologie de containerization

Les conteneurs contiennent un ensemble d'applications, de librairies et d'exécutables. A la différence d'une machine virtuelle qui utilise un hyperviseur pour simuler le matériel physique sur le système d'exploitation de la machine que l'on utilise, la machine hôte, et qui fait tourner un système d'exploitation, le container reprend le système d'exploitation de la machine hôte.

 Le terme de container ou conteneur est très bien choisi pour décrire cette technologie: dans le transport de marchandise, un container est une boîte qui peut tout contenir. De l'extérieur du container, on ne sait pas ce qu'il y a dedans et tout ce dont nous avons besoin est de savoir gérer une grosse boîte en métal. Depuis l'intérieur, on ne sait pas quel est le moyen de transport utilisé (cargo, camion, train, ...) et on ne sait pas non plus ce que contiennent les autres containers.

En fait, un container n'est qu'en ensemble de restrictions sur les ressources (cgroups) ou sur les groupes d'utilisateur, le réseau et les processus (namespaces) qui s'appliquent autour d'un processus. Lorsqu'un container est lancé, un processus est lancé en même temps. Lorsque le processus termine, le container est terminé lui aussi.

Prenons l'exemple d'un système Linux: un container pourra faire tourner un processus, par exemple l'entraînement d'un modèle de Machine Learning, en attribuant certaines ressources à ce container mais surtout en isolant le processus du reste de la machine, c'est-à-dire en ne lui laissant pas l'accès au système de fichiers de la machine hôte, aux différents utilisateurs de la machine hôte et en lui donnant son propre système d'adressage IP.

Un des premiers intérêts de la containerization est dans la légèreté des containers eux-mêmes: ils ne contiennent pas de système d'exploitation et donc sont plus faciles à faire tourner si on a des ressources limitées. De plus, là où les machines virtuelles nécessitent le fonctionnement d'un hyperviseur, un processus émulant les ressources physiques d'une machine, les containers utilisent directement les propriétés des systèmes Linux pour tourner.

 Les containers fonctionnent particulièrement bien avec les systèmes Linux. Ils peuvent aussi fonctionner avec des systèmes Windows mais on doit alors utiliser une couche de virtualisation supplémentaire pour émuler un système Linux: on a alors besoin d'une VM.

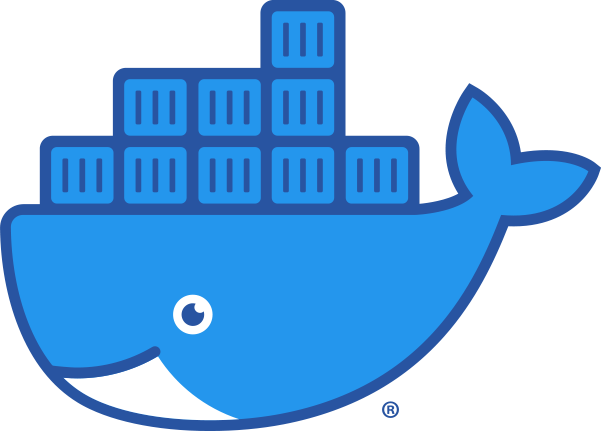
Un autre intérêt des containers est le système d'images de containers. Les images sont des modèles, des patrons de containers qu'on peut choisir d'instancier à l'envie. On peut rapprocher les images des classes en programmation orientée objet: les containers sont les instances de classe alors que les images sont les classes.

Les images sont construites selon un système d'arbres. On part d'une image de base et petit à petit on ajoute des composants. Sur l'image de base on vient ainsi ajouter une distribution Linux, une distribution de distribution, des librairies, des logiciels, ... Ce système d'arborescence est intéressant car il permet de mutualiser les ressources nécessaires à créer les images: si deux images sont basées sur un système Linux, Ubuntu par exemple, la machine hôte n'a pas besoin de stocker deux fois cette partie de l'image.

Il existe différentes technologies qui permettent d'utiliser des containers: runC, containerd, cri-o... Une initiative de normalisation des technologies container est menée par la Linux Foundation via l'Open Container Initiative.

2. Docker

Docker est un outil reposant sur containerd qui permet de créer facilement des containers. C'est à l'origine un projet interne d'une entreprise française, nommée dotCloud et le projet sera ensuite publié en projet Open Source en 2013. Docker est un acteur majeur de l'écosystème des containers car il a permis de faciliter et de démocratiser l'utilisation des containers.



Docker est en effet un outil simple à prendre en main, qui permet un déploiement rapide et sécurisé. De plus, une des grandes forces de Docker réside dans son répertoire d'images: [DockerHub](https://hub.docker.com/). Ce répertoire contient plus de 5 millions d'images avec notamment de nombreuses images officielles, c'est-à-dire créée par des entreprises développant des outils: quasiment toutes les entreprises qui développent des bases de données Open Source sont amenées à proposer une image "Dockerisée" de leurs bases de données sur le DockerHub.

 DockerHub est un outil très intéressant et, sans aucun doute, un des gros points forts de Docker. Nous vous conseillons donc de vous y créer un compte et d'explorer les différentes images qui s'y trouvent: MySQL, MongoDB, Spark, Hadoop, Jupyter, ...

Une autre force de Docker est la facilité avec laquelle on peut créer une image et la partager, notamment sur le DockerHub. Via un système de fichiers, les Dockerfiles, on arrive à créer une image de container en quelques minutes et à la partager tout aussi vite. Ainsi un bon nombre d'images que l'on peut trouver sur DockerHub sont en fait créées par des "amateurs" et offertes à tous.

Enfin, il est intéressant de noter que la plupart des fournisseurs de Cloud proposent aujourd'hui des solutions PaaS (Platform as a Service) dans lesquels vous pouvez déployer facilement des images Docker: le fournisseur de cloud nous permet alors de lancer des containers sans avoir à s'occuper de l'architecture et de la paramétrisation de la machine qui héberge le container.

Docker présente aujourd'hui une très grande communauté (7 millions d'utilisateurs) avec deux grandes éditions de son outil, une version communautaire, gratuite, que nous utiliserons dans ce cours et une version pour les entreprises qui permet une plus grande automatisation ainsi qu'une sécurité accrue.

3. Fonctionnement de Docker

Docker fonctionne selon une architecture client-serveur. Un client, dans ce cours docker-cli, c'est-à-dire l'interface en ligne de commande de Docker vient interroger un démon, docker daemon en utilisant une API REST. C'est ce démon qui s'occupe des différentes tâches de construction et de gestion des images, de lancement des containers.

Enfin, une dernière composante importante de Docker sont les Docker registries, qui sont des répertoires d'images que le démon Docker va interroger pour télécharger une image à lancer. Nous avons vu dans la partie précédente que DockerHub était le répertoire public utilisé en général par les utilisateurs de Docker mais dans certains cas, en particuliers avec les éditions enterprise de Docker, on utilise plutôt des trusted registories, c'est-à-dire des répertoires privés.

Notons enfin que le fonctionnement exact de Docker et des containers en général est géré par le système Linux de la machine hôte ou par son émulateur si on travaille sur une machine Windows.

## 3. Manipuler containers et images

Dans cette partie, nous allons voir les commandes de base de Docker en utilisant l'interface en ligne de commande de Docker: docker-cli.

 Si vous utilisez la machine virtuelle proposée, Docker est déjà installé. Si vous souhaitez installer Docker sur votre machine pour suivre le cours, vous pouvez suivre les instructions disponibles [ici](https://docs.docker.com/engine/install/).

Une fois installé, nous pouvons lancer le démon Docker en utilisant la commande suivante:

Exécutez cette commande pour lancer le démon Docker

sudo service docker start

Lorsque Docker est lancé, nous allons pouvoir lancer notre premier container. Pour cela, nous utiliserons une image classique: hello-world.

Exécutez cette commande pour instancier une image hello-world

docker container run hello-world

Le message suivante devrait apparaître:

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

0e03bdcc26d7: Pull complete

Digest: sha256:31b9c7d48790f0d8c50ab433d9c3b7e17666d6993084c002c2ff1ca09b96391d

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

(amd64)

3. The Docker daemon created a new container from that image which runs the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it

to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:

https://hub.docker.com/

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

Nous avons réussi à lancer notre premier container! Cette image ne présente pas un grand intérêt car elle ne permet pas de faire autre chose que d'afficher ce message.

Essayons de comprendre ce qu'il s'est passé. En utilisant cette commande, on utilise docker-cli, c'est-à-dire l'interface en ligne de commande de Docker. On aurait pu utiliser un autre client (Python, JS, Docker Desktop, ... ) mais celui-ci est très simple à utiliser. Ce client va directement contacter le démon Docker en lui demandant de lancer un container basé sur l'image dont le nom est hello-world (1). Comme cette image n'est pas disponible localement, le démon Docker télécharge cette image depuis DockerHub (2). Le démon Docker se charge ensuite de lancer le container en utilisant cette image comme guide (3). Lors de l'exécution du container, une sortie est produite qui est ensuite re-transmise au client, ici l'interface en ligne de commande (4).

Notons qu'une fois ces étapes terminées le container est ensuite arrêté.

Si nous souhaitons relancer une image de ce même container, nous pouvons exécuter la même commande:

Ré-exécutez cette commande

docker container run hello-world

Si vous faites attention, nous avons perdu une partie du message. En effet, l'image ayant déjà été téléchargée, le démon Docker n'a pas besoin de refaire ce téléchargement: elle est sauvegardée localement.

On peut renvoyer la liste des images téléchargées localement en utilisant la commande suivante.

Exécutez cette commande pour voir la liste des images disponibles

docker image ls

Pour l'instant, nous n'avons qu'une image disponible. On peut voir plusieurs informations dans ce tableau:

* REPOSITORY: le nom de l'image
* TAG: une étiquette qui permet de différencier des images qui ont le même nom
* IMAGE ID: un identifiant unique qui permet d'identifier une image
* CREATED: la date de création de l'image
* SIZE: la taille de l'image

Lorsqu'on crée une image Docker, on lui donne un nom qui permet de l'identifier facilement. On peut aussi lui donner un tag ou une étiquette qui permet de différencier différentes versions d'une même image. Si on ne précise pas de tag, Docker considère que le tag utilisé est latest. Pour un même nom, ou même repository, on peut avoir plusieurs tags différents qui coexistent. De plus, un id unique lui est attribué lorsque l'image est téléchargée ou créée localement: l'id correspond à l'image hashée.

 On peut utiliser plusieurs syntaxes pour manipuler des objets avec l'interface en ligne de commande de Docker. Par exemple, on peut lancer un container en utilisant les commandes docker run hello-world ou docker container run hello-world. Toutefois, par soucis de clarté, nous choisirons de ne présenter que la syntaxe qui introduit le type d'objet docker container run ..., docker image ls, ...

Pour télécharger une image sans lancer de container, on peut utiliser la commande docker image pull. Nous allons télécharger une image de la distribution Ubuntu.

Exécutez cette commande pour télécharger l'image de ubuntu avec le tag latest

docker image pull ubuntu:latest

Exécutez cette commande pour lister les images présentes sur la machine

docker image ls

Nous avons bien deux images à présent. Tentons de lancer une instance de l'image de ubuntu:

Exécutez cette commande pour lancer une instance de l'image ubuntu:latest.

docker container run ubuntu:latest

Cette commande termine sans produire de sortie. En effet, un container Docker n'a pas vocation à tourner indéfiniment. Lorsqu'on lance un container, une commande est exécutée et lorsqu'elle se termine, le container s'arrête. On peut choisir d'exécuter une commande qui nous permet d'interagir avec le container. Dans notre cas, nous allons utiliser une console bash pour interagir avec l'intérieur du container. Pour cela, nous allons devoir utiliser l'argument -it ou --interactive et préciser l'outil à utiliser.

Exécutez cette commande pour lancer une instance de ubuntu et interagir avec elle

docker container run -it ubuntu:latest

On peut voir que nous sommes dans le container qui se comporte comme une machine ubuntu. Nous y sommes ici connectés en tant que root. Vous pouvez explorer l'intérieur de ce container pour constater qu'on a bien affaire à un système ubuntu.

Dans une autre console, nous allons afficher la liste des containers en fonctionnement.

Exécutez cette commande pour lister l'ensemble de containers en fonctionnement

docker container ls

Nous pouvons voir notre container représenté par ses caractéristiques:

* CONTAINER ID: identifiant unique du container
* IMAGE: le nom de l'image utilisée
* COMMAND: la commande utilisée lors du lancement du container
* CREATED: la date de lancement du container
* STATUS: le statut du container
* PORTS: les tunnels créés entre la machine et le container (nous y reviendrons)
* NAMES: nom unique utilisé pour identifier facilement le container

Exécutez la commande exit depuis l'intérieur du container

Une fois cette commande exécutée, la commande bash est terminée et le container s'arrête.

Ré-exécutez cette commande pour lister les containers en fonctionnement

docker container ls

On ne voit plus auncu container. On peut toutefois lister les containers arrêtés en ajoutant l'argument -a ou --all.

Exécutez cette commande pour lister tous les containers

docker container ls –all

On voit les 4 containers lancés plus tôt. Remarquons que tous ces containers possèdent une commande à exécuter. Les images de hello-world utilisent la commande /hello. Par défaut, le container ubuntu est lancé avec la commande /bin/bash.

Nous pouvons relancer un container arrêté en utilisant la commande docker start suivie du nom du container ou de son id. Par exemple, si le container s'appelle practical\_murdock (les noms sont générés de manière aléatoire), on peut exécuter la commande suivante:

Par défaut, relancer un container arrêté se fait en arrière-plan et n'affiche ni la sortie standard, ni l'erreur standard. Pour rediriger ces sorties vers la sortie standard de la machine, on peut utiliser l'argument -a ou --attach.

Relancez le premier container de l'image hello-world avec la commande start

# attention de changer le nom du container avec le vôtre !

docker container start -a practical\_murdock

Le message s'affiche bien, puis la commande termine et le container s'arrête. Si on affiche à nouveau les containers arrêtés, la date de création du container ne change pas mais son statut a changé. On ne peut a priori pas interagir avec un container déjà arrêté: en réalité, Docker n'est pas vraiment fait pour ce genre de situation. Un container peut être utilisé plusieurs fois si il réalise une tâche donnée mais si on souhaite utiliser un container pour interagir avec lui, on préférera utiliser un nouveau container.

On peut utiliser la commande docker container stop suivi de l'identifiant d'un container en fonctionnement ou son nom pour l'arrêter. Essayons de reproduire cette commande:

Dans une console, lancez la commande suivante pour lancer un container

docker container run -it ubuntu:latest bash

Dans une autre console, exécutez la commande suivante pour afficher les containers en fonctionnement

docker container ls

Dans cette même console, exécutez la commande suivante pour arrêter le container en le désignant par son nom

# attention de changer le nom du container avec le vôtre !

docker container stop mystifying\_matsumoto

Vérifiez que le container a bien été arrêté en utilisant la commande suivante:

docker container ls -a

On commence à avoir quelques containers arrêtés. On peut choisir de supprimer ces containers en utilisant la commande docker container rm suivi du nom ou de l'identifiant du container arrêté.

Supprimez les containers arrêtés

Les noms des containers sont choisis de manière aléatoire à partir d'un adjectif et d'un nom de scientifique célèbre. Pour choisir soi-même un nom de container, on peut utiliser l'argument -n ou --name suivi du nom du container. Attention toutefois à ne pas nommer deux containers de la même manière: si on souhaite renommer un container de la même manière qu'un autre container, il faut arrêter puis supprimer le premier.

Un container peut être lancé de manière détachée, c'est-à-dire en arrière-plan, en utilisant l'argument -d ou --detach.

**Dans une commande**, exécutez la commande suivante pour lancer un container en arrière-plan

docker container run -it --detach --name my\_ubuntu ubuntu:latest bash

**Dans une autre console**, exécutez la commande suivante pour afficher les containers en fonctionnement

docker container ls

Dans cette même console, exécutez la commande suivante pour arrêter le container en le désignant par son nom

# attention de changer le nom du container avec le vôtre !

docker container stop mystifying\_matsumoto

Vérifiez que le container a bien été arrêté en utilisant la commande suivante:

docker container ls -a

On commence à avoir quelques containers arrêtés. On peut choisir de supprimer ces containers en utilisant la commande docker container rm suivi du nom ou de l'identifiant du container arrêté.

Supprimez les containers arrêtés

 Les noms des containers sont choisis de manière aléatoire à partir d'un adjectif et d'un nom de scientifique célèbre. Pour choisir soi-même un nom de container, on peut utiliser l'argument -n ou --name suivi du nom du container. Attention toutefois à ne pas nommer deux containers de la même manière: si on souhaite renommer un container de la même manière qu'un autre container, il faut arrêter puis supprimer le premier.

Un container peut être lancé de manière détachée, c'est-à-dire en arrière-plan, en utilisant l'argument -d ou --detach.

**Dans une commande**, exécutez la commande suivante pour lancer un container en arrière-plan

docker container run -it --detach --name my\_ubuntu ubuntu:latest bash

N'est affiché dans la console que l'id du container mais si on lance la commande docker container ls, on peut voir le container en fonctionnement.

Arrêtez puis supprimez le container

docker container stop my\_ubuntu

docker container rm my\_ubuntu

En ajoutant l'argument -rm ou --remove, on peut forcer la suppression à l'arrêt d'un container.

Relancez ce container avec ce nouvel argument

docker container run -it --rm --detach --name my\_ubuntu ubuntu:latest bash

Arrêtez le container puis vérifiez que le container a bien été supprimé

docker container stop my\_ubuntu

docker container ls –all

Un des derniers arguments simples que l'on peut utiliser pour lancer un container est -e. Cet argument permet de créer une variable d'environnement. Cela revient à définir une variable avec un export par exemple. Cette commande peut être très utile pour faire passer des arguments spécifiques à un container.

Exécutez la commande suivante pour lancer un container avec une variable d'environnement ma\_variable

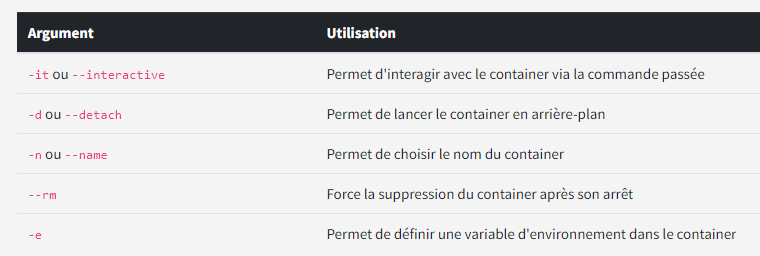
docker container run -it --rm --name my\_ubuntu -e "ma\_variable='bonjour le monde'" ubuntu:latest bash

Une fois dans le container, exécutez la commande suivante pour vérifier le contenu de la variable ma\_variable

echo $ma\_variable

Sortez du container avec la commande exit (remarquez qu'une fois sorti du container, le container est arrêté et donc supprimé)

Dans cette partie, nous avons vu les commandes de base pour télécharger une image et pour lancer un container. Dans le tableau suivant, vous trouverez un résumé des arguments utilisés dans cette partie.



La dernière commande de cette partie permet de voir les propriétés d'un container.

Lancez un container ubuntu nommé my\_ubuntu en arrière-plan. On ne forcera pas la suppression du container lors de son arrêt

docker container run -d -it --name my\_ubuntu ubuntu:latest

On peut utiliser la commande docker container inspect suivi de l'identifiant du container ou son nom pour pouvoir voir les propriétés du container:

Exécutez cette commande pour inspecter le container

docker container inspect my\_ubuntu

Cette commande renvoie une liste au format JSON. Cette liste ne contient que le container my\_ubuntu mais on aurait pu inspecter plusieurs containers d'un coup. Nous verrons plus en détails certains des attributs du container mais pour l'instant, nous pouvons voir les informations sur le statut, sur le nom, l'id, ... On peut aussi voir les détails sur l'image utilisée.

Arrêtez le container

Lancez la commande pour inspecter le container

docker container inspect my\_ubuntu

On peut voir que le contenu de ce JSON a changé, en particulier l'attribut State.

 La structure de la commande docker ... inspect ... est une structure classique pour inspecter les attributs d'un objet. Par exemple, pour voir les attributs d'une image, on peut utiliser docker image inspect hello-world. On peut utiliser la commande docker inspect ... suivi d'un nom d'objet mais la première syntaxe est plus propre. Nous verrons par la suite que cette commande peut s'appliquer à tous les objets de Docker.

## 4. Communication avec les containers

Dans les parties précédentes, nous avons vu que la virtualisation permet d'isoler des processus ou une machine hôte d'autres processus. Toutefois, cette isolation ne doit pas être hermétique mais contrôlée. On peut souhaiter communiquer avec le container. Nous verrons dans cette partie comment communiquer avec des containers via le protocole HTTP.

Pour cette partie, nous allons utiliser une image du moteur de recherche ElasticSearch.

Téléchargez l'image elasticsearch avec le tag 7.2.0

docker image pull elasticsearch:7.2.0

Lancez une instance d'ElasticSearch avec les propriétés suivantes:

* en mode détaché
* en forçant la suppression lors de l'arrêt du container
* le nom du container doit être my\_es\_container
* une variable d'environnement discovery.type doit avoir la valeur single-node

docker container run -d --rm -e "discovery.type=single-node" --name my\_es\_container elasticsearch:7.2.0

Le moteur de recherche d'ElasticSearch expose une API sur le port 9200 de la machine hôte sur laquelle il est lancé. On peut donc accéder à cette API en effectuant une requête HTTP de type GET à l'adresse http://localhost:9200.

Exécutez la commande suivante pour effectuer cette requête

curl -X GET -i http://localhost:9200

Cette commande génère une erreur: l'adresse n'est pas disponible. En effet, le container possède sa propre adresse IP qui n'est pas celle de la machine.

Dans les réseaux informatiques, localhost désigne le réseau local de la machine. Cela désigne généralement une plage d'adresses IP. Lorsqu'on l'utilise dans une requête HTTP, dans la plupart des cas, ce mot désigne la première adresse de cette plage, à savoir 127.0.0.1.

En utilisant la commande docker container inspect, nous pouvons voir quelle est l'adresse IP attribuée à ce container.

Exécutez la commande suivante pour trouver l'adresse IP du container

docker container inspect my\_es\_container | grep IPAddress

Si le container est le seul à fonctionner, l'adresse IP devrait être 172.17.0.2: les adresses IP sont incrémentées de 1 au fur et à mesure à partir de cette adresse IP.

Ré-exécutez la commande précédente avec l'adresse du container

curl -X GET -i http://172.17.0.2:9200

On devrait obtenir le résultat suivant:

HTTP/1.1 200 OK

content-type: application/json; charset=UTF-8

content-length: 508

{

"name" : "eda03f03ebe4",

"cluster\_name" : "docker-cluster",

"cluster\_uuid" : "dtuY0R3nSOy9SRcP7j3FCw",

"version" : {

"number" : "7.2.0",

"build\_flavor" : "default",

"build\_type" : "docker",

"build\_hash" : "508c38a",

"build\_date" : "2019-06-20T15:54:18.811730Z",

"build\_snapshot" : false,

"lucene\_version" : "8.0.0",

"minimum\_wire\_compatibility\_version" : "6.8.0",

"minimum\_index\_compatibility\_version" : "6.0.0-beta1"

},

"tagline" : "You Know, for Search"

}

La suite de cette partie rentre beaucoup dans les détails de la communication entre containers et n'est pas nécessaire pour comprendre ces notions. Si vous le souhaitez, vous pouvez vous reportez directement à la [conclusion](https://train.datascientest.com/data_engineering/course/243/718#conclusion-communication) de cette partie

Pour rappel, le container se comporte comme une machine virtuelle: elle possède ses propres ports, son propre réseau localhost et ses propres IP locales. Ainsi, si on utilise ElasticSearch sans Docker, c'est-à-dire si on installe directement ElasticSearch sur une machine, on peut interroger les ports de la machine, en l'occurrence le port 9200 sur l'adresse 127.0.0.1. Dans le cas d'un container Docker, il se passe la même chose **à l'intérieur** du container: ElasticSearch est disponible à l'adresse 127.0.0.1:9200 à l'intérieur du container.

Du point de vue de la machine hôte, on donne une adresse IP au container, ce qui permet de communiquer avec l'extérieur du container. Toutefois, sur le même schéma des tunnels ssh, on peut faire suivre certains ports du container sur les ports de la machine hôte en utilisant l'argument -p ou --port.

Arrêtez le container my\_es\_container

docker container stop my\_es\_container

On peut à présent relancer notre container en utilisant cette redirection.

docker container run -d --rm\

-e "discovery.type=single-node"\

-p 9201:9200 -p 9301:9300\

--name my\_es\_container elasticsearch:7.2.0

Dans notre cas, ElasticSearch tourne donc sur le port 9200 du container qui est redirigé vers le port 9201 de la machine hôte.

Exécutez la commande suivante pour adresser une requête sur le port 9201 de la machine hôte

curl -X GET -i http://127.0.0.1:9201

On retrouve bien notre message. Affichons à présent la liste des containers en fonctionnement:

Exécutez la commande suivante pour afficher la liste des containers en fonctionnement

docker container ls

On voit que les informations concernant notre container ont changé: on a à présent des informations sur les redirections. Si on tente de relancer un container sur ces ports, on va rencontrer une erreur:

Exécutez cette commande pour tenter de relancer un container sur les mêmes ports

docker container run -d --rm\

-e "discovery.type=single-node"\

-p 9201:9200 -p 9301:9300\

--name my\_es\_container2 elasticsearch:7.2.0

Error response from daemon: driver failed programming external connectivity on endpoint my\_es\_container2 (2a3a6427339cbee0388acb7f87e22cf8e1295fa5053c271558715bce37e475dc): Bind for 0.0.0.0:9301 failed: port is already allocated.

Les ports de la machine hôte sont déjà occupés donc on ne peut pas surcharger la machine. Intéressons-nous à présent aux communications entre containers. Nous allons utiliser une image "maison" qui permet de faire une requête HTTP depuis l'intérieur d'un container.

Téléchargez l'image datascientest/curl:latest

docker image pull datascientest/curl:latest

Pour lancer ce container, il suffit de lancer la commande suivante:

docker container run --rm datascientest/curl:latest http://example.org

On fait ici une requête HTTP de type GET à l'adresse http://example.org. Tentons de faire une requête sur le port 9200 à l'adresse 127.0.0.1, c'est-à-dire comme si nous étions à l'intérieur du container ElasticSearch.

Exécutez la commande suivante pour effectuer cette requête

docker container run --rm datascientest/curl:latest http://127.0.0.1:9200

La connexion est refusée. Il s'agit en effet de l'adresse de l'API ElasticSearch à l'intérieur du container ElasticSearch et n'est donc pas accessible depuis un autre container. Tentons à présent une requête sur le port 9201 de l'adresse 127.0.0.1.

Exécutez la commande suivante pour effectuer cette requête

docker container run --rm datascientest/curl:latest http://127.0.0.1:9201

La connexion est encore refusée. Il s'agit en effet de l'adresse de redirection du container ElasticSearch sur la machine hôte et n'est pas accessible depuis l'intérieur d'un container. Tentons enfin de faire cette requête sur l'adresse réelle du container ElasticSearch 172.17.0.2:

Exécutez la commande suivante pour effectuer cette requête

# si l'adresse du container n'est pas la même sur votre machine, changez là

docker container run --rm datascientest/curl:latest http://172.17.0.2:9200

On obtient à présent le résultat classique. Pour comprendre comment ces communications fonctionnent, il faut parler de la notion de network. En effet, lorsqu'on lance un container, il est attribué à un network (réseau). Les containers sur un même network peuvent communiquer "facilement" entre eux.

Nous allons faire la liste des networks:

Exécutez cette commande pour afficher la liste des networks

docker network ls

Par défaut, il n'existe que trois networks: bridge, none et host. Par défaut, le network utilisé est le network bridge. On peut l'inspecter en utilisant la commande docker network inspect suivie du nom du network à inspecter.

Inspectez le network bridge

Inspect network bridge

[

{

"Name": "bridge",

"Id": "8112c4b4784e4ffc83cfefd958cb88152781eef25799d4595be13aa43fef2b5f",

"Created": "2021-01-27T11:04:45.243012448+01:00",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": null,

"Config": [

{

"Subnet": "172.17.0.0/16",

"Gateway": "172.17.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Ingress": false,

"ConfigFrom": {

"Network": ""

},

"ConfigOnly": false,

"Containers": {

"ce38a8de52d0c7b9d2863480d4440e78cbabe985371de3454a836cbb5c281f46": {

"Name": "my\_es\_container",

"EndpointID": "91cd3bb193de02f8c09c93caa5cce8f700a32b8196df258501b12880eedc8c7e",

"MacAddress": "02:42:ac:11:00:02",

"IPv4Address": "172.17.0.2/16",

"IPv6Address": ""

}

},

"Options": {

"com.docker.network.bridge.default\_bridge": "true",

"com.docker.network.bridge.enable\_icc": "true",

"com.docker.network.bridge.enable\_ip\_masquerade": "true",

"com.docker.network.bridge.host\_binding\_ipv4": "0.0.0.0",

"com.docker.network.bridge.name": "docker0",

"com.docker.network.driver.mtu": "1500"

},

"Labels": {}

}

]

On retrouve plusieurs informations intéressantes:

* des informations générales sur le nom et la date de création du network
* des informations sur les redirections qui ont lieu
* des informations sur les containers qui fonctionnent sur ce network

Dans la clef IPAM, pour IP Address Management, on retrouve le Subnet qui vaut 172.17.0.0/16, c'est à dire que bridge peut donner aux containers qu'il contient, les adresses allant de 172.17.0.2 à 172.17.255.255. En effet l'adresse 172.17.0.1 est le gateway (point d'entrée) utilisé par le démon docker pour gérer ce sous-réseau.

Docker distribuera les IP disponibles dans un réseau en incrémentant les adresses de 1. On peut créer nos propres networks en utilisant la commande docker network create.

Exécutez cette commande pour créer le réseau my\_network

docker network create --subnet 172.50.0.0/16 --gateway 172.50.0.1 my\_network

Ici, on a choisi d'attribuer la plage d'adresses 172.50.0.0 à 172.50.255.255 à ce réseau.

Listez les réseaux

docker network ls

Inspectez le réseau my\_network

docker network inspect my\_network

De la même façon qu'avec les containers ou les images, les ID et les noms des networks doivent être uniques. On peut les manipuler de la même façon en utilisant les commandes docker network create, docker network rm docker network ls , ...\*

Pour préciser sur quel réseau un container doit être lancé, on utiliser l'argument -n ou --network suivi du nom du réseau.

Exécuter cette cellule pour lancer un autre container ElasticSearch sur ce réseau

docker container run -d --rm\

-e "discovery.type=single-node"\

--network my\_network\

--name my\_es\_container3 elasticsearch:7.2.0

Inspectez le réseau my\_network

docker network inspect my\_network

On peut voir que le container fonctionne bien sur ce réseau. L'adresse donnée à ce container est 172.50.0.2.

Exécutez la commande suivante pour faire une requête à cette adresse depuis l'intérieur du réseau bridge

docker container run datascientest/curl:latest http://172.50.0.2:9200

La commande expire. Par contre, si on fait la requête dans le réseau my\_network, la requête termine:

Exécutez la commande suivante pour faire une requête à cette adresse depuis l'intérieur du réseau my\_network

docker container run --network my\_network datascientest/curl:latest <http://172.50.0.2:9200>

L'adresse est bien atteinte. Résumons la situation: un network est un sous-réseau de la machine hôte. Lorsqu'on container est déployé sur un network, il se voit attribuer une adresse IP locale de la machine hôte. Le container est alors atteignable depuis la machine hôte. De l'intérieur du network, le container est atteignable à cette même adresse. Mais depuis l'intérieur d'un autre network, cette adresse IP n'est pas atteignable.

Notons que les adresses IP attribuées ne sont pas fixes: si on arrête un container, il perd son adresse et pourra se voir attribuer une autre adresse IP. Pour pallier ce problème, on peut utiliser le nom des containers qui agiront alors comme des noms de domaine:

Exécutez cette commande pour utiliser le nom du container

docker container run --network my\_network datascientest/curl:latest <http://my_es_container3:9200>

Jusqu'à présent, nous avons vu que les network construits sur le modèle de bridge sont utiles pour créer des espaces dans lesquels les containers sont relativement isolés. Utiliser judicieusement ces réseaux permet de contrôler quelles sont les communications qui ont lieu entre les différents containers. Et nous avons vu que les noms des containers peuvent remplacer les adresses pour les containers au sein d'un même réseau. Intéressons-nous au réseau host.

Arrêtez tous les containers en fonctionnement

Le réseau host est un réseau particulier qui permet de ne pas avoir de frontière entre le container et les ports de la machine.

Lancez un container ElasticSearch sur ce réseau, sans préciser de redirection et vérifiez que le container fonctionne bien

docker container run -d --rm\

-e "discovery.type=single-node"\

--network host\

--name my\_es\_container elasticsearch:7.2.0

Si on lance la commande docker network inspect host, on peut voir qu'il ne possède pas les mêmes caractéristiques que les autres réseaux: pas de Subnet, ni de GateWay. Si on inspecte le container avec la commande docker container inspect my\_es\_container, on s'aperçoit qu'il ne possède pas l'IP: son IP est en fait 127.0.0.1.

Exécutez une requête de type GET avec curl sur le port 9200 de la machine hôte

curl -X GET -i http://127.0.0.1:9200

Arrêtez les containers en fonctionnement

Le réseau host permet de faciliter les communications entre le container et la machine hôte. Toutefois on n'isole plus nécessairement les communications, ce qui peut présenter des problèmes de sécurité. Il ne fonctionne d'ailleurs pas sur des machines utilisant des systèmes d'exploitation Mac ou Windows.

### Conclusion

Dans cette partie, nous avons vu comment communique les containers. Par défaut, ils sont déployés sur un réseau bridge. Ils peuvent donc se voir entre eux et communiquer, soit via leur IP, soit via leur nom, qu'on préférera utiliser. On peut créer différents réseaux pour les isoler les uns des autres. On peut aussi accéder à ces containers depuis la machine hôte en utilisant leur adresse IP donnée par Docker. Enfin, on a vu qu'on pouvait exposer directement les ports de la machine en utilisant des redirections ou le réseau host.

## 5. Persistence des données

Par défaut, Docker ne permet pas la persistence des données: les données créées à l'intérieur d'un container finissent par être supprimées lorsque le container est arrêté.

Lancez un container, instance de ubuntu:latest, nommée my\_ubuntu de manière interactive. On pourra forcer la suppression lors de l'arrêt.

docker container run --rm --name my\_ubuntu -it ubuntu:latest

ous allons écrire des données dans un fichier **à l'intérieur du container**.

Exécutez cette commande pour créer le fichier

echo "hello world from Docker" > /home/test.txt

Vous pouvez vérifiez le contenu du fichier en utilisant cat /home/test.txt puis sortir du container avec exit. Si on relance un container, on ne peut pas retrouver le fichier.

Relancez le container précédent et essayez d'afficher le contenu du fichier /home/test.txt.

Cette commande génère une erreur: il s'agit d'une autre image. L'image datascientest/time-keeper:latest permet d'écrire l'heure dans un fichier à l'intérieur d'un container si ce fichier n'existe pas déjà. Si il existe déjà, le fichier n'est pas créé. Dans les deux cas, on imprime le résultat.

Exécutez la commande suivante

docker run --name my\_time\_keeper datascientest/time-keeper:latest

Après quelques secondes, relançons ce container

docker container start -a my\_time\_keeper

On remarque dans ce cas que l'heure n'a pas changé. Le fichier a été conservé entre les deux démarrages du container. Toutefois, conserver un container Docker arrêté est peu économe en termes de place occupée sur le disque. Pour permettre la persistance des données entre deux lancements, on préférera utiliser des volumes. Les volumes sont des dossiers qui permettent de stocker des données. Ils sont montés (un peu comme une clef USB) sur des containers et continuent d'exister après que le container ait été arrêté et même supprimé.

Pour créer un volume, on peut utiliser la commande docker volume create.

Exécutez cette commande pour créer un volume

docker volume create --name my\_volume

Exécutez cette commande pour lister les volumes existants

docker volume ls

Inspectez le volume my\_volume

docker volume inspect my\_volume

Dans cette inspection du volume, on peut voir la date de création, son nom, certains attributs ainsi que sa localisation: /var/lib/docker/volumes/my\_volume/\_data.

On peut monter un volume sur un container en utilisant l'argument --mount. Pour utiliser cet argument, il faut préciser:

* type: volume
* src: le nom du volume utilisé
* dst: le dossier de destination à l'intérieur du container

Exécutez cette commande pour lancer un container ubuntu

docker container run -it --name my\_ubuntu\

--mount type=volume,src=my\_volume,dst=/home/my\_folder\

--rm\

ubuntu:latest bash

Une fois dans le container, exécutez la commande suivante pour lister les dossier dans /home

ls /home

On trouve un dossier my\_folder: il correspond au point de monture du volume.

Exécutez cette commande pour créer un fichier dans le dossier my\_folder

echo "hello world from Docker" > /home/my\_folder/test.txt

Dans une autre console, affichez le contenu du dossier correspondant au volume

sudo ls /var/lib/docker/volumes/my\_volume/\_data

sudo cat /var/lib/docker/volumes/my\_volume/\_data/test.txt

Le fichier correspond bien à celui que nous venons de créer dans le container. On peut créer un fichier depuis la machine et le mettre dans ce dossier: il apparaîtra dans le container:

Depuis la machine

echo "hello world from the host machine" > ~/test2.txt

sudo mv ~/test2.txt /var/lib/docker/volumes/my\_volume/\_data/test2.txt

Depuis le container

ls /home/my\_folder

cat /home/my\_folder/test2.txt

On peut arrêter le container (exit). On peut vérifier le contenu du fichier /var/lib/docker/volumes/my\_volume/\_data mais rien n'a changé.

Dans une console, relancez un container ubuntu avec la commande suivante

docker container run -it\

--name my\_ubuntu1\

--mount type=volume,src=my\_volume,dst=/home/my\_folder1\

--rm\

ubuntu:latest bash

Dans cette même console, lister les fichiers présents dans /home/my\_folder1

Le volume a bien permis de conserver les fichiers créés entre deux lancements du container.

Dans une autre console, relancez un autre container ubuntu avec la commande suivante

docker container run -it\

--name my\_ubuntu2\

--mount type=volume,src=my\_volume,dst=/home/my\_folder2\

--rm\

ubuntu:latest bash

Dans cette même console, lister les fichiers présents dans /home/my\_folder2

Le volume est bien partagé entre les deux containers: les volumes sont un outil particulièrement intéressant pour partager des ressources de stockage entre containers. De plus, ils sont difficilement atteignables depuis la machine hôte, notamment car manipuler ces dossiers depuis la machine hôte requiert des droits d'administrateur.

Arrêtez les containers en fonctionnement

Supprimez le volume my\_volume en exécutant la commande suivante

docker volume rm my\_volume

Pour facilement échanger des fichiers avec la machine hôte et permettre la persistence des données, on peut utiliser l'argument --volume avec un dossier qui existe déjà.

Par exemple, on peut lancer la commande suivante:

docker container run -it\

--name my\_ubuntu\

--volume $HOME:/home/my\_folder\

--rm\

ubuntu:latest bash

Avec cette commande, on relie directement le dossier $HOME de la machine hôte au dossier /home/my\_folder dans le container. On peut le vérifier en listant le contenu de ce dossier ls /home/my\_folder. Cette technique est très pratique car on peut facilement faire passer ou retirer des données du container depuis la machine hôte sans avoir les droits d'administrateur. Toutefois, c'est évidemment moins sécurisé.

## 6. Créer une image

Un des intérêts de Docker est qu'on peut créer facilement des images de containers et les partager. Pour créer une image, il suffit de créer un fichier Dockerfile dans un dossier et d'exécuter la commande docker build dans ce dossier. Dans cette partie, nous allons voir comment créer un container Docker contenant un serveur Flask.

Créez un dossier my\_docker\_image sur la machine hôte et créez un fichier Dockerfile

mkdir my\_docker\_image

cd my\_docker\_image

nano Dockerfile

Pour créer une image, on doit utiliser une image de base. L'image la plus simple est scratch mais elle est vide. Une des images les plus simples est alpine mais on choisira ici l'image debian:latest dans notre cas. Pour ce faire, il faut utiliser le mot-clef FROM.

Ajoutez la ligne suivante dans le fichier Dockerfile

FROM debian:latest

Pour que notre serveur fonctionne, il faut que Python soit installé dans le container. Pour exécuter une commande dans l'image, on peut utiliser le mot-clef RUN.

Ajoutez la ligne suivante dans le fichier Dockerfile

RUN apt-get update && apt-get install python3-pip -y && pip3 install flask==2.1.2

Cette commande installe Python et pip puis installe la librairie flask. Ici, on choisit d'enchaîner les commandes grâce à des &&. En effet, le fait de n'utiliser qu'une commande RUN permet d'alléger l'image.

Pour la suite, nous avons besoin d'utiliser un fichier Python contenant le code du serveur.

Dans une autre console, créez un fichier server.py dans le dossier my\_docker\_image et y placer les lignes suivantes:

from flask import Flask

server = Flask('my\_server')

@server.route('/')

def index():

return 'Hello World from a containerized server'

if \_\_name\_\_ == '\_\_main\_\_':

server.run(host='0.0.0.0', port=5000)

Sauvegardez ce fichier en en sortant

Ce fichier permet de créer un serveur qui tournera sur le port 5000 de la machine qui le lance, dans notre cas, ce sera le container. Pour ajouter ce fichier à l'image Docker, nous pouvons utiliser la commande ADD:

Ajoutez la ligne suivante dans le fichier Dockerfile

ADD server.py /my\_server/server.py

Cette commande peut être vue comme un cp classique: on copie le fichier server.py de la machine hôte dans l'image dans le dossier my\_server. Nous allons faire changer le dossier courant à l'intérieur de l'image avec le mot-clef WORKDIR. Ce mot-clef fonctionne comme la commande cd des systèmes Linux.

Ajoutez la ligne suivante dans le fichier Dockerfile

WORKDIR /my\_server/

Comme le serveur est exposé sur le port 5000, il faut préciser que ce port peut être atteint depuis l'extérieur du container si on en fait la demande:

Ajoutez la ligne suivante dans le fichier Dockerfile

EXPOSE 5000

Enfin, l'image Docker nécessite un point d'entrée, une commande qui doit s'exécuter lors du lancement du container.

Ajoutez la ligne suivante dans le fichier Dockerfile

CMD python3 server.py

FROM debian:latest

RUN apt-get update && apt-get install -y python-pip

ADD server.py /my\_server/server.py

WORKDIR /my\_server/

EXPOSE 5000

CMD python3 server.py

Une fois ces lignes ajoutées, sauvegardez et quittez le fichier

Créez l'image en utilisant la commande suivante dans le dossier contenant le Dockerfile

docker buildx build . -t my\_image:latest

docker image build . -t my\_image:latest

Le . désigne le dossier courant, contenant le dockerfile et l'argument -t permet de préciser le nom de l'image ainsi que le tag. Si on ne précise pas le tag, le tag latest lui est donné par défaut. Cette étape peut prendre un peu de temps puisqu'il faut télécharger l'image debian:latest puis différents outils: ce téléchargement se fait bien lors de la création de l'image et non lors de son instanciation.

Vérifiez que l'image a bien été créée en listant les images locales

docker image ls

Lancez le container en créant une redirection entre le port 5000 de la machine et celui du container et en mode détaché

docker container run -p 5000:5000 -d my\_image:latest

Exécutez une requête GET sur le port 5000 de la machine hôte

curl -X GET -i http://localhost:5000

Le message s'affiche bien: le serveur est bien opérationnel depuis l'intérieur du container. Dans le tableau suivant, vous trouverez des informations sur les principales commandes qui peuvent être utilisées dans un dockerfile.



Pour terminer cette partie, nous allons parler des différentes façons de partager une image. La manière la plus simple est de créer un fichier .tar, une archive, pour partager facilement cette image. Pour créer une archive, on peut utiliser la commande docker image save et pour extraire une image d'une archive, on peut utiliser la commande docker image load.

Créez une archive à partir de l'image créée en utilisant la commande suivante

docker image save --output my\_docker\_image.tar my\_image

Supprimez l'image my\_image: on pourra utiliser l'argument -f pour forcer la suppression d'un container en marche

docker image rm -f my\_image

Recréez l'image à partir de l'archive en utilisant la commande suivante

docker image load --input my\_docker\_image.tar

Une fois cette commande effectuée, on peut vérifier que l'image existe bien.

Vérifiez que cette image a bien été créée

Une autre méthode consiste à utiliser le DockerHub. Nous l'avons dit dans les parties précédentes mais le DockerHub contient aussi bien des images créées par des particuliers que des images officielles créées par des entreprises. Nous avons utilisé certaines images officielles (debian, ubuntu, elasticsearch, ...) qui sont développées par des organisations officielles ainsi que des images propres à DataScientest: datascientest/curl, ...

Si vous souhaitez héberger une image dans DockerHub et donc la rendre accessible à tout le monde, vous devez vous créer un compte DockerHub et vous connecter à ce compte sur votre machine en utilisant docker login.

 Nous ne conseillons pas de mettre des identifiants personnels sur une machine hébergée (en l'occurence ici, hébergée par nous). Si vous souhaitez réaliser cette étape, vous pouvez tout à fait installer Docker localement et réaliser cette étape sur votre ordinateur personnel.

Une fois connecté, on doit créer des images qui comportent un nom en deux parties username/imagename:tag avec username qui correspond à votre nom d'utilisateur de DockerHub, imagename au nom de l'image et tag qui correspond au tag que vous souhaitez donner à cette image. Lorsque l'image est créée, on peut la mettre en ligne en utilisant la commande:

docker image push username/imagename:tag

L'image est alors disponible depuis le DockerHub et n'importe quel utilisateur de Docker peut utiliser cette image. Pour vous en assurer, vous pouvez supprimer l'image localement puis la télécharger avec un docker image pull username/imagename:tag.

7. Docker Compose

Docker Compose est un outil qui permet de lancer un ensemble de containers Docker en même temps. Si une application doit utiliser plusieurs containers, lancer tous ces containers d'un coup est très pratique. Une des utilisations communes de Docker Compose est de créer des environnements de test: par exemple, si on doit tester un script qui doit interagir avec une interaction, on peut lancer un ensemble de containers qui constitueront l'ensemble de test. En particulier, Docker Compose n'est pas utilisé pour créer un ensemble de containers sur différentes machines mais sur une seule machine.

 Si vous utilisez la machine virtuelle proposée, Docker-Compose est déjà installé. Si vous souhaitez installer Docker-Compose sur votre machine pour suivre le cours, vous pouvez suivre les instructions disponibles [ici](https://docs.docker.com/compose/install/).

Docker Compose a été installé sur la machine virtuelle. Pour créer une application Docker Compose, on doit éditer un fichier yaml.

Dans un fichier docker-compose.yml, collez les lignes suivantes:

version: "3.9"

services:

jupyter:

image: jupyter/minimal-notebook:ubuntu-18.04

elasticsearch:

image: elasticsearch:7.2.0

docker-compose up

Dans une autre console, listez les containers en fonctionnement

On peut voir que nous avons pu créer un container: une instance de jupyter/minimal-notebook. Dans le fichier docker-compose.yml, on peut voir deux services jupyter et elasticsearch: ce fichier est en réalité censé lancer deux containers mais comme nous l'avons vu précédemment, les images d'ElasticSearch nécessitent que l'on définissent une variable d'environnement. On peut préciser des variables d'environnement en utilisant le mot-clef environment.

On peut arrêter les containers lancés par Docker Compose en utilisant la ctrl + C dans la console dans laquelle docker-compose up a été effectuée ou en tapant docker-compose down dans une autre console.

Arrêtez les containers lancés par Docker Compose

Remplacez le contenu du fichier docker-compose.yml par les lignes suivantes

version: "3.9"

services:

jupyter:

image: jupyter/minimal-notebook:ubuntu-18.04

elasticsearch:

image: elasticsearch:7.2.0

environment:

discovery.type: single-node

Quittez ce fichier, exécutez la commande docker-compose up puis listez les containers en fonctionnement

Cette fois-ci on peut voir que les containers sont bien tous les deux en fonctionnement. Pour pouvoir utiliser jupyter, on doit faire suivre le port 8888 du container vers un des ports de la machine hôte.

Remplacez le contenu du fichier docker-compose.yml par ces lignes, puis relancez les services Docker Compose

version: "3.9"

services:

jupyter:

image: jupyter/minimal-notebook:ubuntu-18.04

ports:

- target: 8888

published: 4444

protocol: tcp

mode: host

elasticsearch:

image: elasticsearch:7.2.0

environment:

discovery.type: single-node

Ici on fait suivre le port 8888 vers le port 4444 de la machine hôte. Vous pouvez ouvrir ce port pour arriver sur l'interface graphique de jupyter. On peut noter qu'il existe une autre manière de faire suivre des ports. Par exemple, si on veut faire suivre les ports 9200 et 9300 du container elasticsearch vers ceux de la machine hôte, on peut faire:

version: "3.9"

services:

jupyter:

image: jupyter/minimal-notebook:ubuntu-18.04

ports:

- target: 8888

published: 4444

protocol: tcp

mode: host

elasticsearch:

image: elasticsearch:7.2.0

environment:

discovery.type: single-node

ports:

- "9200:9200"

- "9300:9300"

On peut aussi spécifier les networks ou donner un nom aux containers. Dans le code suivant, on place les containers sur le réseau my\_network\_from\_compose, on les renomme my\_jupyter\_from\_compose et my\_es\_from\_compose. Enfin, on précise que le mot de passe à utiliser pour utiliser jupyter est bonjour.

version: "3.9"

services:

jupyter:

image: jupyter/minimal-notebook:ubuntu-18.04

container\_name: my\_jupyter\_from\_compose

networks:

- my\_network\_from\_compose

ports:

- target: 8888

published: 4444

protocol: tcp

mode: host

environment:

JUPYTER\_TOKEN: "bonjour"

elasticsearch:

image: elasticsearch:7.2.0

container\_name: my\_es\_from\_compose

networks:

- my\_network\_from\_compose

ports:

- "9200:9200"

- "9300:9300"

environment:

discovery.type: single-node

networks:

my\_network\_from\_compose:

Mettez à jour le contenu du fichier docker-compose.yml avec ces lignes et relancez ces services

Une fois que les services sont lancés, allez sur le navigateur à l'adresse IP de votre machine virtuelle et le port 4444, vous devrez avoir l'interface jupyter.

Dans un nouveau notebook, exécutez le code suivant:

import requests

import pprint

import json

content = json.loads(requests.get('http://my\_es\_from\_compose:9200').content)

pprint.pprint(content)

Enfin, on peut préciser créer des points de monture en précisant volumes. Par exemple, dans le code suivant, on peut lier le dossier work du container vers le dossier local:

version: "3.9"

services:

jupyter:

image: jupyter/minimal-notebook:ubuntu-18.04

container\_name: my\_jupyter\_from\_compose

networks:

- my\_network\_from\_compose

volumes:

- .:/home/jovyan/work

ports:

- target: 8888

published: 4444

protocol: tcp

mode: host

environment:

JUPYTER\_TOKEN: "bonjour"

elasticsearch:

image: elasticsearch:7.2.0

container\_name: my\_es\_from\_compose

networks:

- my\_network\_from\_compose

ports:

- "9200:9200"

- "9300:9300"

environment:

discovery.type: single-node

networks:

my\_network\_from\_compose:

Mettez à jour le fichier, relancez les services et vérifier le contenu du dossier work

Docker Compose permet donc de lancer facilement un ensemble de containers. Nous avons vu ici les principaux arguments de Docker Compose. Nous n'irons pas plus loin car ce n'est pas l'objet de ce cours. Notez que si nous souhaitons lancer des containers sur un système distribué, on utilisera des outils comme Docker Swarm ou Kubernetes.

## 8. Clients Docker

Dans ce cours, nous avons utilisé docker-cli, l'interface en ligne de commande de Docker. Dans cette partie, nous allons lister d'autres clients Docker qui peuvent vous être utiles.

### 1. Docker Desktop

Docker Desktop est une application pour MacOS ou Windows qui propose une interface graphique pour manipuler les différents objets créés par Docker. Si vous souhaitez en apprendre plus, vous pouvez vous rendre sur ce [lien](https://docs.docker.com/desktop/).

### 2. Portainer

[Portainer](https://www.portainer.io/) est une interface graphique de Docker Open Source, qui tourne dans un container. Sa prise en main est assez intuitive et son lancement est plutôt simple:

docker volume create portainer\_data

docker run -d -p 8000:8000 -p 9000:9000 --name=portainer --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v portainer\_data:/data portainer/portainer-ce

On peut alors se rendre sur le port 9000 de la machine et explorer les différents menus.

### 3. Client Python

Python est le langage le plus utilisé dans le monde de la Data Science. On peut utiliser la librairie docker (pip3 install docker) pour créer, lister, lancer ou supprimer différents objets.

Par exemple, on peut lancer un container en utilisant le code suivant:

from docker import DockerClient

client = DockerClient()

client.containers.run(image='datascientest/neo4j:latest',

name='my\_neo4j',

detach=True,

auto\_remove=True,

ports={

'7474/tcp': 7474,

'7687/tcp':7687},

network='bridge'

)

# printing names of active containers

for c in client.containers.list():

print(c.name, c.image)

La documentation est disponible [ici](https://docker-py.readthedocs.io/en/stable/index.html)