# Guide pratique -- Dockerfiles

# Introduction

Docker permet de télécharger une image Docker prédéfinie du Docker Hub.   
Bien qu'il soit utile d'obtenir des images Docker, il est possible de créer des images Docker personnalisées.   
Ceci est important pour exécuter ses applications sur Docker en installant de nouveaux paquets et en personnalisant les paramètres des images Docker prédéfinies.

Cela se fera à l'aide d'un fichier texte appelé Dockerfile.   
Ce fichier se compose de commandes pouvant être exécutées par Docker afin de créer une image Docker.

Les images Docker sont créées à partir d'un Dockerfile à l'aide de la commande **docker image build**.

**Remarque** …  
La syntaxe Docker CLI est structurée sous la forme de …  
**>> docker <Commande> <Sous-commande>**   
Cette structuration permet de simplifier la syntaxe Docker CLI et d’obtenir un regroupement de commandes plus cohérent.

Une image Docker se compose de plusieurs calques, chaque calque représentant les commandes fournies dans un fichier Dockerfile.   
Ces calques en lecture seule sont empilés les uns sur les autres afin de créer l'image Docker finale.

Les images Docker peuvent être stockées dans un registre Docker, tel que Docker Hub, qui est un endroit où l’on peut stocker et distribuer des images.

Un conteneur Docker est une instance en cours d'exécution de l'image Docker.  
Un ou plusieurs conteneurs Docker peuvent être créés à partir d'une seule image Docker à l'aide de la commande **docker container run**.   
Une fois qu'un conteneur Docker est créé à partir de l'image Docker, un nouveau calque inscriptible sera ajouté au‑dessus des calques en lecture seule de l'image Docker.   
Les conteneurs Docker peuvent ensuite être listés avec la commande **docker container list**   
(ou **docker container ls** ou **docker container ps**).

Il peut y avoir un ou plusieurs calques en lecture seule qui composent l'image Docker.   
Ces calques en lecture seulement sont générés pour chaque commande dans le Dockerfile pendant le processus de création de l'image.   
Une fois qu'un conteneur Docker est créé à partir de l'image, un nouveau calque en lecture-écriture (connu sous le nom de calque Container) est ajouté au-dessus des calques d'image et héberge toutes les modifications apportées au conteneur en cours d'exécution.

# Dockerfile

Un Dockerfile est un fichier texte qui contient des instructions sur la façon de créer une image Docker.   
Ces commandes sont appelées directives.   
Un Dockerfile est un mécanisme utilisé pour créer une image Docker personnalisée.

Le format d'un Dockerfile est le suivant …  
**# Ceci est un commentaire  
DIRECTIVE Argument**

Un Dockerfile peut contenir plusieurs lignes de commentaires et de directives.   
Ces lignes seront exécutées dans l'ordre par le moteur Docker lors de la construction de l'image Docker.   
Comme les langages de programmation, un Dockerfile peut également contenir des commentaires.

Toutes les déclarations commençant par le symbole # seront traitées comme des commentaires.  
Actuellement, un Dockerfiles ne prend en charge que les commentaires sur une seule ligne.   
Si on souhaite écrire un commentaire sur plusieurs lignes, on doit ajouter le symbole # au début de chaque ligne.

Les instructions d’un Dockerfile ne sont pas sensibles à la casse.   
Même si une DIRECTIVE est insensible à la casse, il est recommandé d'écrire toutes les directives en majuscules pour les distinguer des arguments.

## Directives communes

Une directive est une commande utilisée pour créer une image Docker.   
Voici cinq directives Dockerfile importantes …

* directive **FROM** ;
* directive **LABEL** ;
* directive **RUN** ;
* directive **CMD** ;
* directive **ENTRYPOINT**.

### Directive FROM

Un Dockerfile commence généralement par la **directive FROM**.   
Elle est utilisée pour spécifier l'image parent de l’image qui sera créée.   
L'image parent est le point de départ de l’image personnalisée.   
Toute la personnalisation qui est appliquée au-dessus de l'image parente.   
L'image parente peut être une image téléchargée du Docker Hub, comme Ubuntu, Debian, nginx et MySQL.   
La **directive FROM** prend un nom d'image valide et une balise comme arguments.   
Si la balise n'est pas spécifiée, la dernière balise (*latest*) est utilisée.

Une **directive FROM** possède le format suivant …  
**FROM <Image>:<Balise>**

Pour utiliser l'**image** **parent** **ubuntu** avec la **balise** **20.04** …  
**FROM ubuntu:20.04**

De plus, on peut utiliser l'image de base si on souhaite créer une image à partir de zéro.  
L'image de base, connue sous le nom de **scratch**, est une image vide principalement utilisée pour créer d'autres images parentes.

Dans la **directive FROM** suivante, on utilise l'image **scratch** pour créer une image personnalisée à partir de zéro …  
**FROM scratch**

### Directive LABEL

Une **directive LABEL** est une paire clé-valeur qui peut être utilisée pour ajouter des métadonnées à une image.   
Ces étiquettes ou libellés peuvent être utilisées pour organiser correctement les images Docker.   
Un exemple serait d'ajouter le nom de l'auteur du Dockerfile ou la version du Dockerfile.

Une **directive LABEL** possède le format suivant …  
**LABEL <Clé>=<Valeur>**

Un Dockerfile peut contenir plusieurs libellés …  
**LABEL** [**Responsable=tux@squiddly.info**](mailto:Responsable=tux@squiddly.info) **LABEL Version=1.0  
LABEL Environnement=dev**

Ces étiquettes peuvent également être incluses sur une seule ligne séparées par des espaces …  
**LABEL** [**Responsable=tux@squiddly.info**](mailto:Responsable=tux@squiddly.info) **Version=1.0 Environnement=dev**

Les étiquettes d’une image existante peuvent être affichées avec la commande **docker image inspect** …  
**…  
"Labels": {  
 "Environnement": "dev",  
 "Responsable": "tux@squiddly.info",  
 "Version": "1.0"  
}  
...**

### Directive RUN

La **directive RUN** est utilisée pour exécuter des commandes lors de la construction de l'image.

La **directive RUN** …

* crée un nouveau calque au-dessus du calque existant ;
* exécute la commande spécifiée   
  et
* valide les résultats dans le calque nouvellement créé.

La **directive RUN** peut être utilisée pour …

* installer des paquets ;
* mettre à jour les paquets déjà installés ;
* créer des utilisateurs et des groupes ;
* ...

La **directive RUN** prend le format suivant …  
**RUN <Commande>**

La valeur **<Commande>** spécifie la commande shell que l’on souhaite exécuter dans le cadre du processus de création d'image.   
Un Dockerfile peut posséder plusieurs **directives RUN** respectant le format précédent.

On exécute trois commandes au-dessus de l'image parent.   
Les commandes apt-get update et at upgrade -y sont utilisées pour mettre à jour les référentiels de paquets et la mise à jour du conteneur. La commande apt-get install nginx -y est utilisé pour installer le paquet nginx …  
**RUN apt-get update  
RUN apt-get upgrade -y  
RUN apt-get install nginx -y**

Il est également possible d’ajouter plusieurs commandes shell dans une seule directive RUN en les séparant par le symbole && .   
Cela permet d’alléger l’image résultante.

On utilise les trois mêmes commandes, dans une seule **directive RUN**, séparées par un   
symbole && …  
**RUN apt-get update && apt-get upgrade -y && apt-get install nginx -y**

### Directive CMD

Un conteneur Docker est normalement censé exécuter un processus.   
La directive CMD est utilisée pour fournir cette commande d'initialisation par défaut qui sera exécutée lors de la création d'un conteneur à partir d’une l'image.   
Un Dockerfile ne peut exécuter qu'une seule directive CMD.   
S'il y a plus d'une directive CMD dans le Dockerfile, Docker n'exécutera que la dernière.

La directive CMD possède le format suivant …  
**CMD ["Binaire","Paramètre1","Paramètre2","Paramètre3", ...]**

On utilise la commande suivante pour renvoyer "Bonjour le monde" comme sortie d'un conteneur Docker …  
**CMD ["echo","Bonjour le monde"]**

La directive CMD précédente produit la sortie suivante lors du démarrage du conteneur avec la commande   
**docker container run <Image>** …  
**>> docker container run <Image>  
Bonjour le monde**

Si on ajoute des arguments à la commande **docker container run <Image>**, ces arguments auront priorité sur la commande CMD définie dans le Dockerfile.   
Si nous exécutons la commande suivante (on remplace <Image> par le nom de l'image), la sortie par défaut "Bonjour le monde" définie avec la directive CMD sera ignorée.   
Au lieu de cela, le conteneur affichera "Bonjour Docker !!! " …  
**>> docker container run <Image> echo "Bonjour Docker !!!"**

Les **directives RUN et CMD** peuvent être utilisées pour exécuter une commande shell.   
La principale différence entre ces deux directives est …

* que la commande fournie avec la **directive RUN** sera exécutée pendant le processus de construction de l'image ;
* que la commande fournie avec la **directive CMD** sera exécutée une fois qu'un conteneur sera lancé à partir de l'image construite.

Une autre différence notable entre les **directives RUN et CMD** est …

* qu'il peut y avoir plusieurs directives RUN dans un Dockerfile ;
* qu’il ne peut y avoir qu'une seule directive CMD   
  (s'il y a plusieurs directives CMD , toutes les autres sauf la dernière seront ignorées).

Il est possible d’utiliser la **directive RUN** pour installer un paquet logiciel pendant le processus de construction de l'image Docker et la **directive CMD** pour démarrer le paquet logiciel une fois qu'un conteneur est lancé à partir de l'image construite.

### Directive ENTRYPOINT

Semblable à la **directive CMD**, la **directive ENTRYPOINT** est également utilisée pour fournir une commande d'initialisation par défaut qui sera exécutée lorsqu'un conteneur est créé à partir d’une image.   
La différence entre la **directive CMD** et la **directive ENTRYPOINT** est que, contrairement à la **directive CMD**, il n’est pas possible de remplacer une commande exécutée par une **directive ENTRYPOINT** à l'aide des paramètres de ligne de commande envoyés avec la commande **docker container** **run**.

**Remarque** …  
Le commutateur **--entrypoint** peut être envoyé avec la commande **docker container run** afin de remplacer la **directive ENTRYPOINT** par défaut de l'image.

La directive ENTRYPOINT possède le format suivant …  
**ENTRYPOINT ["Binaire","Paramètre1","Paramètre2","Paramètre3", ...]**

Semblable à la **directive CMD**, la directive ENTRYPOINT permet également de fournir l'exécutable par défaut et les paramètres.  
On peut utiliser la **directive CMD** avec la **directive ENTRYPOINT** pour fournir des arguments supplémentaires à l'exécutable.

On utilise **echo** comme commande par défaut et **Bonjour** comme paramètre par défaut à l'aide de la **directive** **ENTRYPOINT**.   
On a également fourni **le Monde** comme paramètre supplémentaire à l'aide de la **directive CMD** …  
**ENTRYPOINT ["echo","Bonjour"]  
CMD ["le Monde"]**

La sortie de la commande echo différe en fonction de la façon dont on exécut la commande d'exécution du conteneur.

Si on lance l'image sans aucun paramètre de ligne de commande, elle affichera le message **Bonjour le Monde** …  
**>> docker container run <image>  
Bonjour le Monde**

Toutefois, si on lance l'image avec des paramètres de ligne de commande supplémentaires, le message de sortie est différent …  
**>> docker container run <image> "Docker"  
Bonjour Docker**

# Création d’un Dockerfile

On crée un nouveau répertoire nommé custom-docker-image à l'aide de la commande mkdir ...  
**>> mkdir --verbose custom-docker-image  
mkdir: création du répertoire 'custom-docker-image'**

Ce répertoire sera le contexte de l’image.   
Le contexte est le répertoire qui contient tous les fichiers nécessaires pour créer une image.

On accède au répertoire custom-docker-image car on doit y créer tous les fichiers requis pendant le processus de construction (y compris le Dockerfile) dans ce répertoire …  
**>> cd custom-docker-image**

Dans le répertoire custom-docker-image, on crée un fichier nommé Dockerfile à l'aide de la commande nano ou vim …  
**>> nano Dockerfile**ou  
**>> vim|nano Dockerfile**

On ajoute le contenu suivant au Dockerfile …  
**# Ceci est ma première image Docker  
FROM ubuntu  
LABEL** [**Auteur=tux@squiddly.info**](mailto:Auteur=tux@squiddly.info) **RUN apt-get update  
CMD ["Dockerfile"]  
ENTRYPOINT ["echo", "Mon premier "]**

Quelques explications sur ce Dockerfile …

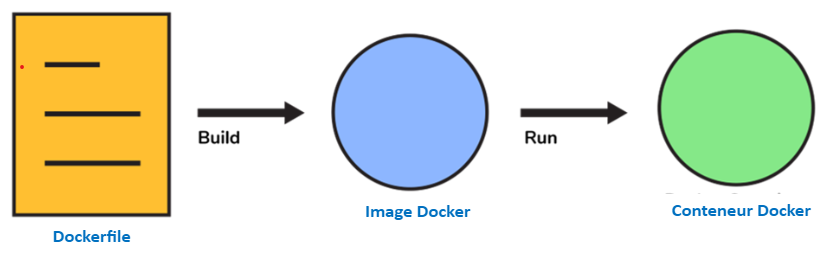
* La **directive FROM** indique que la nouvelle image est basée sur l'image parent Ubuntu ;
* La **directive LABEL** permet de spécifier l'adresse de courriel de l'auteur du Dockerfile ;
* La **directive RUN** exécute la commande apt-get update pour mettre à jour la liste des paquets vers la dernière version disponible ;
* Les **directives ENTRYPOINT et CM**D permettent de définir le binaire par défaut et les paramètres du conteneur.

## Construction d’une image

Une image Docker est un modèle utilisé pour créer des conteneurs Docker.   
Ceci est analogue à la façon dont un plan de maison peut être utilisé pour créer plusieurs maisons à partir du même design.

Une image Docker est un fichier binaire composé de plusieurs calques basés sur les instructions fournies dans le Dockerfile.   
Ces calques sont empilés les uns sur les autres et chaque calque dépend du calque précédent.   
Chacun des calques est le résultat des modifications apportées à la calque situé en dessous.

Toutes les calques de l'image Docker sont en lecture seule.   
Une fois un conteneur créé à partir d'une image Docker, un nouveau calque inscriptible est créé au-dessus des autres calques en lecture seule.   
Ce calque contient toutes les modifications apportées au système de fichiers du conteneur.

  
*Calques d’une image Docker*

La commande **docker image build** crée une image Docker à partir du fichier Dockerfile.   
Les calques de l'image Docker seront mappés aux directives fournies dans le Dockerfile.

Ce processus de création d'image est lancé par la CLI Docker et exécuté par le démon Docker.   
Pour générer une image Docker, le démon Docker doit accéder au Dockerfile, au code source (par exemple, index.html) et à d'autres fichiers qui sont référencés dans le Dockerfile.

Ces fichiers sont généralement stockés dans un répertoire appelé contexte de génération.   
Ce contexte est spécifié lors de l'exécution de la commande docker image build.   
L'intégralité du contexte est envoyée au démon Docker pendant le processus de création d'image.

La commande **docker image build** prend le format suivant …  
**>> docker image build <Contexte>**

On exécute la commande **docker image build** à partir du répertoire qui contient le Dockerfile et ses autres fichiers associés …  
**>> docker image build .  
Sending build context to Docker daemon 33.79kB  
Step 1/5 : FROM ubuntu  
 ---> 27941809078c  
Step 2/5 : LABEL** [**Auteur=tux@squiddly.info**](mailto:Auteur=tux@squiddly.info) **---> Running in 4054cc29e04b  
Removing intermediate container 4054cc29e04b  
 ---> 80e6be2d2076  
Step 3/5 : RUN apt-get update  
 ---> Running in e6efb359f7f6  
  
WARNING: apt-get does not have a stable CLI interface. Use with caution in scripts.  
Get:1 http://archive.ubuntu.com/ubuntu jammy InRelease [270 kB]  
Get:2 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]  
…  
Get:17 http://archive.ubuntu.com/ubuntu jammy-backports/universe amd64 Packages [5812 B]  
Fetched 22.0 MB in 5s (4120 kB/s)  
Reading package lists...  
Building dependency tree...  
Reading state information...  
9 packages can be upgraded. Run 'apt-get list --upgradable' to see them.  
Removing intermediate container e6efb359f7f6  
 ---> 895377849582  
Step 4/5 : CMD ["Dockerfile"]  
 ---> Running in a65c4f2fd163  
Removing intermediate container a65c4f2fd163  
 ---> 583f4610c9b1  
Step 5/5 : ENTRYPOINT ["echo", "Mon premier "]  
 ---> Running in 87b810d3d086  
Removing intermediate container 87b810d3d086  
 ---> 01e2a69f6394  
Successfully built 01e2a69f6394**

**Remarque** …  
Le point (.) à la fin de la commande est utilisé pour indiquer le répertoire courant.

On affiche la liste des images disponibles en utilisant la commande **docker image** list …  
**>> docker image list**

La sortie contient les images créées localement et les images téléchargées de référentiels distants.

On construit à nouveau l'image Docker à l'aide de la commande docker image build …  
**>> docker image build .  
Sending build context to Docker daemon 33.79kB  
Step 1/5 : FROM ubuntu  
 ---> 27941809078c  
Step 2/5 : LABEL Auteur=tux@squiddly.info  
 ---> Using cache  
 ---> 80e6be2d2076  
Step 3/5 : RUN apt-get update  
 ---> Using cache  
 ---> 895377849582  
Step 4/5 : CMD ["Dockerfile"]  
 ---> Using cache  
 ---> 583f4610c9b1  
Step 5/5 : ENTRYPOINT ["echo", "Mon premier "]  
 ---> Using cache  
 ---> 01e2a69f6394  
Successfully built 01e2a69f6394**

Le processus de création d'image a été instantané.   
Étant donné qu’aucun contenu n’a été modifié dans le Dockerfile, Docker a profité du cache et a réutilisé les calques existants du cache d'image local pour accélérer le processus de construction.   
Le cache a été utilisé cette fois car on peut le constater (**---> Using cache**).

Le démon Docker effectue une étape de validation avant de démarrer le processus de construction afin de s'assurer que la syntaxe du Dockerfile fourni est correct.

Dans le cas d'une syntaxe invalide, le processus de construction échoue avec un message d'erreur.

On consulte la sortie de la commande **docker image list** …  
**>> docker image list  
REPOSITORY TAG IMAGE ID CREATED SIZE  
<none> <none> 01e2a69f6394 15 hours ago 113MB  
ubuntu latest** **27941809078c 7 weeks ago 77.8MB  
ubuntu 18.04 ad080923604a 7 weeks ago 63.1MB  
ubuntu 19.04 c88ac1f841b7 2 years ago 70MB**

**Remarque** …  
Il n'y a pas de nom pour l’image personnalisée.   
En effet, on n’a spécifié aucun référentiel ou balise pendant le processus de construction.

On peut baliser une image existante avec la commande **docker image tag** …  
**>> docker image tag 27941809078c mon-image:v1.0**

On consulte la sortie de la commande **docker image list** …  
**>> docker image list  
REPOSITORY TAG IMAGE ID CREATED SIZE  
<none> <none> 01e2a69f6394 15 hours ago 113MB  
ubuntu latest 27941809078c 7 weeks ago 77.8MB  
mon-image v1.0 27941809078c 7 weeks ago 77.8MB  
ubuntu 18.04 ad080923604a 7 weeks ago 63.1MB  
ubuntu 19.04 c88ac1f841b7 2 years ago 70MB**

Il est aussi possible de baliser une image pendant le processus de construction avec le commutateur --tag   
(forme courte -t) …  
**>> docker image build --tag mon-image:v2.0 .  
Sending build context to Docker daemon 33.79kB  
Step 1/5 : FROM ubuntu  
 ---> 27941809078c  
Step 2/5 : LABEL Auteur=tux@squiddly.info  
 ---> Using cache  
 ---> 80e6be2d2076  
Step 3/5 : RUN apt-get update  
 ---> Using cache  
 ---> 895377849582  
Step 4/5 : CMD ["Dockerfile"]  
 ---> Using cache  
 ---> 583f4610c9b1  
Step 5/5 : ENTRYPOINT ["echo", "Mon premier "]  
 ---> Using cache  
 ---> 01e2a69f6394  
Successfully built 01e2a69f6394  
Successfully tagged mon-image:v2.0**

# Autres directives Dockerfile

Les directives suivantes peuvent être utilisées pour créer des images Docker plus avancées …

* directive **ENV** ;
* directive **ARG** ;
* directive **WORKDIR** ;
* directive **COPY** ;
* directive **ADD** ;
* directive **USER** ;
* directive **VOLUME** ;
* directive **EXPOSE**. ;
* directive **HEALTHCHECK** ;
* directive **ONBUILD**.

### Directive ENV

La **directive ENV** est utilisée pour définir des variables d'environnement.   
Les variables d'environnement sont utilisées par les applications et les processus pour obtenir des informations sur l'environnement dans lequel un processus s'exécute.

La variable d'environnement PATH est un exemple de variable qui identifie les répertoires dans lesquels le système recherche des fichiers exécutables.

Les variables d'environnement sont définies comme des paires clé-valeur selon le format suivant …  
**ENV <Clé> <Valeur>**

La variable d'environnement PATH (PATH :/usr/local/monapp/bin/) peut être défini à l'aide de la **directive ENV** …  
**ENV PATH $PTH:/usr/local/monapp/bin/**

Il est possible de définir plusieurs variables d'environnement dans la même ligne séparées par des espaces.   
Cependant, sous cette forme, la clé et la valeur doivent être séparées par le symbole égal à (=) …  
**ENV <Clé>=<Valeur> <Clé>=<Valeur> ...**

On configure deux variables d'environnement …

* la variable d'environnement **PATH** est configurée avec la valeur **$PATH:/usr/local/myapp/bin/** ;
* la variable d'environnement **VERSION** est configurée avec la valeur **1.0.0**.

Pour configurer ces deux variables …  
**ENV PATH=$PATH:/usr/local/myapp/bin/ VERSION=1.0.0**

Une fois qu'une variable d'environnement est définie avec la **directive ENV** dans le Dockerfile, cette variable est disponible dans toutes les calques d'image Docker suivantes.   
Cette variable est même disponible dans les conteneurs Docker lancés à partir de cette image Docker.

### Directive ARG

La **directive ARG** est utilisée pour définir des variables que l'utilisateur peut passer au moment de la construction.   
ARG est la seule directive pouvant précéder la directive FROM dans le Dockerfile.

Les utilisateurs peuvent transmettre des valeurs à l'aide de **--build-arg <Nom de la variable>=<Valeur>** …  
**>> docker image build --tag <Image>:<Balise> --build-arg <Nom de la variable>=<Valeur> .**

La **directive ARG** possède le format suivant …  
**ARG <Nom de la variable>**

Il peut y avoir plusieurs **directives ARG dans** un Dockerfile …  
**ARG UTILISATEUR   
ARG VERSION**

La **directive ARG** peut également avoir une valeur par défaut facultative définie.   
Cette valeur par défaut sera utilisée si aucune valeur n'est transmise au moment de la construction …  
**ARG USER=Tux  
ARG VERSION=1.0.0**

Contrairement aux variables ENV, les variables ARG ne sont pas accessibles depuis le conteneur en cours d'exécution.   
Ils ne sont disponibles que pendant le processus de construction.

On crée un nouveau répertoire nommé env-arg-exercise …  
**>> mkdir --verbose** **env-arg-exercise  
mkdir: création du répertoire 'env-arg-exercise'**

On accède au répertoire custom-docker-image …  
**>> cd env-arg-exercise**

Dans le répertoire env-arg-exercise, on crée un fichier nommé Dockerfile …  
**>> nano Dockerfile**ou  
**>> vim Dockerfile  
# Exemple d’instructions ENV and ARG  
ARG TAG=latest  
FROM ubuntu:$TAG  
LABEL Auteur=tux@squiddly.info  
ENV EDITEUR=EditionsTux APP\_DIR=/usr/local/app/bin  
CMD ["env"]**

On construit l’image à l’aide de la commande **docker image build** …  
**>> docker image build --tag env-arg --build-arg TAG=19.04 .  
Sending build context to Docker daemon 2.048kB  
Step 1/5 : ARG TAG=latest  
Step 2/5 : FROM ubuntu:$TAG  
 ---> c88ac1f841b7  
Step 3/5 : LABEL Auteur=tux@squiddly.info  
 ---> Running in 259c1ae66d7f  
Removing intermediate container 259c1ae66d7f  
 ---> d19c427bc447  
Step 4/5 : ENV EDITEUR=EditionsTux APP\_DIR=/usr/local/app/bin  
 ---> Running in 15a56c32db35  
Removing intermediate container 15a56c32db35  
 ---> c49384126fd7  
Step 5/5 : CMD ["env"]  
 ---> Running in 1cddc6137fba  
Removing intermediate container 1cddc6137fba  
 ---> c9f262c5a350  
Successfully built c9f262c5a350  
Successfully tagged env-arg:latest**

Le commutateur **env-arg --build-arg TAG=19.04** est utilisé pour envoyer l'argument **TAG** au processus de construction.

La balise 19.04 de l'image ubuntu a été utilisée comme image parent conformément au commutateur **--build-arg** avec la valeur **TAG=19.04** pendant le processus de construction.

On exécute la commande **docker container run** pour démarrer un nouveau conteneur à partir de l'image nouvellement créée …  
**>> docker container run env-arg  
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  
HOSTNAME=38aa3dc40001  
EDITEUR=EditionsTux  
APP\_DIR=/usr/local/app/bin  
HOME=/root**

Comme on peut le constater, la variable d'environnement EDITEUR est paramétrée avec la valeur de EditionsTux, et la variable d'environnement APP\_DIR est paramétrée avec la valeur de /usr/local/app/bin.

### Directive WORKDIR

La **directive WORKDIR** est utilisée pour spécifier le répertoire de travail actuel du conteneur Docker.   
Toutes les **directives ADD**, **CMD**, **COPY**, **ENTRYPOINT** et **RUN** qui suivent sont exécutées dans ce répertoire.

La **directive WORKDIR** possède le format suivant …  
**WORKDIR <Répertoire>**

Si le répertoire spécifié n'existe pas, Docker le crée et en fait le répertoire de travail actuel, ce qui signifie que cette directive exécute implicitement les commandes **mkdir** et **cd**.

Il peut y avoir plusieurs **directives WORKDIR** dans le Dockerfile.   
Si un chemin relatif est fourni dans une **directive WORKDIR** ultérieure, il sera relatif au répertoire de travail défini par la **directive WORKDIR** précédente …  
**WORKDIR /un  
WORKDIR two  
WORKDIR three  
RUN pwd**

### Directive COPY

Au cours du processus de création d'image Docker, on doit possiblement copier des fichiers du système de fichiers local vers le système de fichiers d'image du conteneur.

Ces fichiers peuvent être …

* des fichiers de code source (des fichiers JavaScript) ;
* des fichiers de configuration (des fichiers de propriétés)   
  ou
* des artefacts (des fichiers JAR).

La **directive COPY** peut être utilisée pour copier des fichiers et des répertoires du système de fichiers local vers l'image pendant le processus de construction.

Cette directive prend deux arguments ...

* le premier est le chemin source du système de fichiers local …
* le second est le chemin de destination sur le système de fichiers image.

Elle prend la forme suivante …  
**COPY <Source> <Destination>**

On utilise la **directive COPY** pour copier le fichier index.html du système de fichiers local vers le répertoire /var/www/html/ de l'image …  
**COPY index.html /var/www/html/index.html**

Des caractères génériques peuvent également être spécifiés pour copier tous les fichiers qui correspondent au modèle donné ...   
**COPY \*.html /var/www/html/**

En plus de copier des fichiers, le commutateur **--chown** peut être utilisé avec la directive COPY afin de spécifier l'utilisateur et le groupe propriétaire des fichiers …  
**COPY --chown=tux:developpeurs \*.html /var/www/html/**

### Directive ADD

La **directive ADD** est également similaire à la **directive COPY** et possède le format suivant …  
**ADD <Source> <Destination>**

Toutefois, en plus de la fonctionnalité fournie par la **directive COPY**, la **directive ADD** permet également d'utiliser une URL comme paramètre **<Source>** …  
**ADD http://tux.info/test.txt/tmp/test.txt**

Une autre caractéristique de la **directive ADD** est l'extraction automatique des fichiers compressés.   
Si on ajoute un fichier compressé (gzip, bzip2, tar, …) au paramètre **<Source>**, la **directive ADD** extrait l'archive et copie le contenu dans le système de fichiers image …  
**ADD html.tar.gz /var/www/html**

**Remarque** …  
Étant donné que les **directives COPY et ADD** fournissent presque les mêmes fonctionnalités, il est recommandé de toujours utiliser la directive COPY sauf si vous avez besoin des fonctionnalités supplémentaires (ajout à partir d'une URL ou extraction d'un fichier compressé) fournies par la **directive ADD**.   
La **directive ADD** fournit des fonctionnalités supplémentaires qui peuvent se comporter de manière imprévisible si elles sont utilisées de manière incorrecte.

On crée un nouveau répertoire nommé workdir-copy-add …  
**>> mkdir --verbose workdir-copy-add  
mkdir: création du répertoire 'workdir-copy-add'**

On se déplace dans le répertoire …  
**>> cd workdir-copy-add**

On crée un fichier nommé index.html.   
Ce fichier sera copié dans l'image Docker lors de la compilation …  
Dans le répertoire env-arg-exercise, on crée un fichier nommé Dockerfile …  
**>> nano|vim index.html  
<html>  
 <body>  
 <h1>Bonjour le monde !!!</h1>  
 <img src="logo.png" height="350" width="500"/>  
 </body>  
</html>**

**Remarque** …  
Le fichier logo.png sera téléchargé lors du processus de création d'image.

On crée un Dockerfile …  
**>> nano|vim Dockerfile  
# Exemple de directives WORKDIR, COPY and ADD  
FROM ubuntu:latest  
RUN apt-get update && apt-get install nginx -y  
WORKDIR /var/www/html/  
COPY index.html .  
ADD** [**https://www.docker.com/wp-content/uploads/2022/03/horizontal-logo-monochromatic-white.png**](https://www.docker.com/wp-content/uploads/2022/03/horizontal-logo-monochromatic-white.png) **./logo.png  
CMD ["ls"]**

On construit l'image Docker avec la balise workdir-copy-add …  
**>> docker image build --tag workdir-copy-add .**

**Remarque** …  
L'image est construite avec succès et étiquetée comme étant la plus récente.  
**>> docker image list w\*  
REPOSITORY TAG IMAGE ID CREATED SIZE  
workdir-copy-add latest 852bb6c82da7 About a minute ago 169MB**

On exécute la commande **docker container run** pour démarrer un nouveau conteneur à partir de l'image …  
**>> docker container run workdir-copy-add  
index.html  
index.nginx-debian.html  
logo.png**

**>> docker container list --all  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
b4a51d98c0d8 workdir-copy-add "ls" About a minute ago Exited (0) About a minute ago**

### Directive USER

Docker utilise l'utilisateur root comme utilisateur par défaut d'un conteneur.   
La **directive USER** permet de modifier ce comportement par défaut et de spécifier un utilisateur non-root comme utilisateur par défaut d'un conteneur.

L’utilisation de la **directive USER** est un excellent moyen d'améliorer la sécurité en exécutant le conteneur en tant qu'utilisateur non privilégié.

Le nom d'utilisateur spécifié avec la **directive USER** sera utilisé pour exécuter toutes les **directives RUN**, **CMD** et **ENTRYPOINT** suivantes dans le Dockerfile.

La directive USER prend le format suivant ..  
**USER <Utilisateur>**

En plus du nom d'utilisateur, il est également possible de spécifier un nom de groupe facultatif pour exécuter le conteneur …  
**USER <Utilisateur>:<Groupe>**

Il est important de s’assurer que les valeurs **USER** et **GROUP** sont des noms d'utilisateur et de groupe valides.   
Sinon, Docker génère une erreur lors de la tentative d'exécution du conteneur.

**Remarque** …  
L'utilisateur www-data est l'utilisateur par défaut du serveur Web Apache2 sur Ubuntu.

On crée un nouveau répertoire nommé user-exercise …  
**>> mkdir --verbose user  
mkdir: création du répertoire 'user'**

On se déplace dans le répertoire …  
**>> cd user**

On crée un Dockerfile …  
**>> nano|vim Dockerfile  
# Exemple de directive USER  
FROM ubuntu  
RUN apt-get update && apt-get install apache2 -y  
USER www-data  
CMD ["whoami"]**

On construit l’image à l’aide de la commande **docker image build** …  
**>> docker image build --tag user .  
Reading state information...  
9 packages can be upgraded. Run 'apt-get list --upgradable' to see them.  
…  
Successfully built 5f0b024065dd  
Successfully tagged user:latest**

On exécute la commande **docker container run** pour démarrer un nouveau conteneur à partir de l'image …  
**>> docker container run user  
www-data**

### Directive VOLUME

Avec Docker, les données (fichiers, exécutables, …) générées et utilisées par les conteneurs seront stockées dans le système de fichiers du conteneur.  
Lors de la suppression d’un conteneur, toutes les données seront perdues.

Afin de surmonter ce problème, Docker propose le concept de volumes.   
Les volumes sont utilisés pour conserver les données et partager les données entre les conteneurs.

On utilise la **directive VOLUME** dans un Dockerfile pour créer des volumes.   
Une fois qu'un volume est créé dans le conteneur, un répertoire lié (mappage) est créé sur le système hôte sous-jacent. Toutes les modifications de fichier apportées au montage de volume du conteneur Docker sont copiées dans le répertoire lié au système hôte.

La directive VOLUME prend généralement un tableau JSON comme paramètre …  
**VOLUME ["Répertoire"]**

Il est possible de spécifier une chaîne simple avec plusieurs chemins …  
**VOLUME <Répertoire01> <Répertoire02>**

On peut utiliser la commande **docker container inspect <Conteneur>** pour afficher les volumes disponibles dans un conteneur.   
La sortie de la commande affiche les informations de volume …  
**>> docker container inspect <Conteneur>  
"Mounts": [  
 {  
 "Type": "volume",  
 "Name": "77db32d66407…",  
 "Source": "/var/lib/docker/volumes/77db32d66407…/\_data",  
 "Destination": "<Répertoire>",  
 "Driver": "local",  
 "Mode": "",  
 "RW": true,  
 "Propagation": ""  
 }**

**],=**

Selon la sortie précédente, un nom unique est donné au volume par Docker.   
De plus, on retrouve les chemins source et de destination du volume.

On peut exécuter la commande **docker volume inspect <Volume>** pour afficher des informations détaillées relatives à un volume …  
**>> docker volume inspect <Volume>  
[  
    {  
        "CreatedAt": "2022-12-28T12:52:52+05:30",  
        "Driver": "local",  
        "Labels": null,  
        "Mountpoint": "/var/lib/docker/volumes/77db32d66407…/\_data",  
        "Name": "77db32d66407…",  
        "Options": null,  
        "Scope": "local"  
     }  
]**

Cette sortie est similaire à la sortie précédente, avec le même nom unique et le chemin de montage du volume.

On crée un nouveau répertoire nommé volume …  
**>> mkdir --verbose volume  
mkdir: création du répertoire 'volume'**

On se déplace dans le répertoire …  
**>> cd volume**

On crée un Dockerfile …  
**>> vim |nano Dockerfile  
# Exemple de directive VOLUME  
FROM ubuntu  
RUN apt-get update && apt-get install apache2 -y  
VOLUME ["/var/log/apache2"]**

On construit l’image à l’aide de la commande **docker image build** …  
**>> docker image build --tag volume .  
Sending build context to Docker daemon 2.048kB  
Step 1/3 : FROM ubuntu  
 ---> 27941809078c  
Step 2/3 : RUN apt-get update && apt-get install apache2 -y  
 ---> Using cache  
 ---> 39c8b5371e48  
Step 3/3 : VOLUME ["/var/log/apache2"]  
 ---> Running in d2b2ed805c61  
Removing intermediate container d2b2ed805c61  
 ---> 9503476d45bb  
Successfully built 9503476d45bb  
Successfully tagged volume:latest**

On exécute la commande **docker container run** pour démarrer un nouveau conteneur à partir de l'image …  
**>> docker container run --interactive --tty --name volume-conteneur volume /bin/bash  
root@efa55f5ce0d9:/#**

On se déplace dans le répertoire /var/log/apache2/ …  
**root@efa55f5ce0d9:/# cd /var/log/apache2/**

On liste les fichiers disponibles dans le répertoire …  
**root@efa55f5ce0d9:/# ls -l  
total 0  
-rw-r----- 1 root adm 0 Jul 26 13:30 access.log  
-rw-r----- 1 root adm 0 Jul 26 13:30 error.log  
-rw-r----- 1 root adm 0 Jul 26 13:30 other\_vhosts\_access.log**

Ces fichiers e journalisation ont été créés par Apache.   
Les mêmes fichiers devraient être disponibles une fois que l’on a vérifié le point de montage sur le système hôte de ce volume.

On quitte le conteneur pour vérifier le système de fichiers hôte …  
**root@efa55f5ce0d9:/# exit**

On vérifie la présence du volume …  
**>> docker volume list  
DRIVER VOLUME NAME  
local de15d55b0e292f5ffa13e5386303936d688db85165794f2cd80d28bde649016e**

On affiche les informations du point de montage …  
**>> docker volume inspect volume-conteneur  
 {  
 "CreatedAt": "2022-07-26T09:55:42-04:00",  
 "Driver": "local",  
 "Labels": null,  
 "Mountpoint": "/var/lib/docker/volumes/de15d55b0e29…/\_data",  
 "Name": "de15d55b0e292f5ffa13e5386303936d688db85165794f2cd80d28bde649016e",  
 "Options": null,  
 "Scope": "local"  
 }  
]**

On constate que le conteneur est monté sur le chemin de l'hôte …  
 " **Mountpoint": "/var/lib/docker/volumes/de15d55b0e29…/\_data",**

On peut afficher la liste des fichiers disponibles (chemin du répertoire hôte) ...  
**>> sudo ls -l /var/lib/docker/volumes/ …  
 … de15d55b0e292f5ffa13e5386303936d688db85165794f2cd80d28bde649016e/\_data  
total 0  
-rw-r----- 1 root adm 0 26 jui 09:30 access.log  
-rw-r----- 1 root adm 0 26 jui 09:30 error.log  
-rw-r----- 1 root adm 0 26 jui 09:30 other\_vhosts\_access.log**

### Directive EXPOSE

La **directive EXPOSE** est utilisée pour informer Docker que le conteneur écoute sur les ports spécifiés au moment de l'exécution.   
On peut utiliser la **directive EXPOSE** pour publier les ports à l’aide les protocoles TCP ou UDP.

La **directive EXPOSE** possède le format suivant …  
**EXPOSE <Port>**

Toutefois, les ports publiés avec la **directive EXPOSE** ne seront accessibles que depuis les autres conteneurs Docker. Pour publier ces ports en dehors du conteneur, il faut publier les ports avec le commutateur **--publish** de la commande **docker container run** …  
**>> docker container run --publish <Port externe>:<Port interne> <Image>**

**Remarque** …  
Le port externe représente le pour du système hôte et le pour interne représente le port à l’intérieur du conteneur.

Les déclarations suivantes définissent la **directive EXPOSE** …

* Si on spécifie à la fois la **directive EXPOSE** et e commutateur **--publish …**les ports exposés seront accessibles à partir d'autres conteneurs ainsi que de l'extérieur ;
* Si on spécifie la **directive EXPOSE** sans le commutateur **--publish** …  
  les ports exposés ne seront accessibles qu'à partir d'autres conteneurs, mais non de l'extérieur.

### Directive HEALTHCHECK

Les vérifications de l'état sont utilisées dans Docker pour vérifier si les conteneurs fonctionnent correctement.   
On utilise des vérifications de l'état afin de s’assurer de la santé d’une application dans le conteneur.

À moins qu'une vérification de l'état ne soit spécifiée, Docker n'a aucun moyen de dire si un conteneur est sain.   
Ceci peut s’avérer très important si on exécute des conteneurs dans des environnements de production.

La **directive HEALTHCHECK** possède le format suivant …  
**HEALTHCHECK [OPTIONS] CMD <Commande>**

Il ne peut y avoir qu'une seule **directive HEALTHCHECK** dans un Dockerfile.   
S'il y a plus d'une **directive HEALTHCHECK**, seule la dernière prendra effet.

On peut utiliser une **directive** **HEALTHCHECK** pour s’assurer que le conteneur peut recevoir du trafic sur le point de terminaison http://localhost/ …  
**HEALTHCHECK CMD curl -f http://localhost/ || exit 11**

Le code de sortie à la fin de la commande précédente est utilisé pour spécifier l'état de santé du conteneur.

0 et 1 sont dls valeurs valides pour ce champ ...

* **0** est utilisé pour désigner un conteneur sain   
  et
* **1** est utilisé pour désigner un conteneur non sain.

En plus de la commande, il est possible de spécifier d’autres paramètres avec la **directive HEALTHCHECK** …

* **--interval**  
  Cette option spécifie la période entre chaque vérification de l'état (la valeur par défaut est de 30 secondes) ;
* **--timeout**  
  Si aucune réponse de succès n'est reçue dans ce délai, la vérification de l'état est considérée comme un échec (valeur par défaut est de 30 s) ;
* **--start-period**  
  Cette option spécifie la durée d'attente avant d'exécuter la première vérification de l'état.   
  Ceci est utilisé pour donner une heure de démarrage pour le conteneur  
  (valeur par défaut est 0s) ;
* **--retries**  
  Le conteneur sera considéré comme non sain si la vérification de l'état échoue consécutivement pour le nombre de tentatives donné (la valeur par défaut est 3).

On peut remplacer les valeurs par défaut en fournissant des valeurs personnalisées avec la **directive HEALTHCHECK** …  
**HEALTHCHECK --interval=1m --timeout=2s --start-period=2m --retries=3 \ CMD curl -f http://localhost/ || exit 1**

On crée un nouveau répertoire nommé expose-healthcheck …  
**>>** **mkdir --verbose expose-healthcheck  
mkdir: création du répertoire 'expose-healthcheck'**

On se déplace dans le répertoire …  
**>> cd onbuild-parent**

On crée un Dockerfile …  
**>> nano|vim Dockerfile  
# Exemple de directives EXPOSE & HEALTHCHECK  
FROM ubuntu  
RUN apt-get update && apt-get install apache2 curl -y  
HEALTHCHECK CMD curl -f http://localhost/ || exit 1  
EXPOSE 80  
ENTRYPOINT ["apache2ctl", "-D", "FOREGROUND"]**

On construit l’image à l’aide de la commande **docker image build** …  
**>> docker image build --tag expose-healthcheck .  
Sending build context to Docker daemon 2.048kB  
Step 1/3 : FROM ubuntu  
 ---> 27941809078c  
…  
Successfully built 6904aa6cdeb8  
Successfully tagged expose-healthcheck:latest**

On exécute la commande **docker container run** pour démarrer un nouveau conteneur à partir de l'image …   
**>> docker container run --publish 80:80 --name expose-healthcheck-conteneur --detach expose-healthcheck  
b49052019a75049617cf63ce792e5b4e5885c442738cab24e0d711a706ddb623**

On affiche la liste des conteneurs en cours d'exécution avec la commande **docker container list** …  
**>> docker container list  
CONTAINER ID IMAGE COMMAND CREATED   
 STATUS PORTS NAMES  
b49052019a75 expose-healthcheck "apache2ctl -D FOREG…" About a minute ago  
 Up About a minute (healthy) 0.0.0.0:80->80/tcp, :::80->80/tcp expose-healthcheck-conteneur**

On peut constater que l’état (STATUS) du conteneur expose-healthcheck est sain.

Il est maintenant possible d’accéder la page d'accueil d'Apache à l’aide d’un fureteur (*browser*) …  
http://<Adresse IP>.

On arrête et supprime le conteneur …  
**>> docker container stop expose-healthcheck-conteneur  
expose-healthcheck-conteneur  
>> docker container rm expose-healthcheck-conteneur  
expose-healthcheck-conteneur**

**Remarque** …  
Il est possible d’arrêter un conteneur et le supprimer en une seule commande …  
**>> docker container rm --force <Conteneur>**

### Directive ONBUILD

La **directive ONBUILD** est utilisée dans un Dockerfile pour la création d’une image réutilisable qui servira de base à une autre image.   
La **directive ONBUILD** permet créer une image qui contient tous les prérequis, tels que les dépendances et les configurations, afin d'exécuter une application.

Ensuite, il sera possible d’utiliser cette image comme image de base pour exécuter une application.

Lors de la création de cette image de base, on peut utiliser la **directive ONBUILD** qui inclue les instructions qui ne doivent être exécutées que lorsque cette image est utilisée comme image de base dans un autre Dockerfile.

Les **instructions ONBUILD** ne sont pas exécutées lors de la construction du Dockerfile contenant la **directive ONBUILD**, mais uniquement lors de la construction de l'image enfant.

La directive ONBUILD possède le format suivant …  
**ONBUILD <Instruction>**

On inclut une directive ONBUILD dans un Dockerfile …  
**ONBUILD ENTRYPOINT ["echo","Exécution de la directive ONBUILD"]**

La valeur **Exécution de la directive ONBUILD** n’est pas affichée si on crée un conteneur à partir de l’image de base personnalisée.   
Toutefois, la valeur **Exécution de la directive ONBUILD** est affiché si on utilise l’image de base personnalisée comme base pour la nouvelle image enfant.

On utilise la commande **docker image inspect** pour l'image parente afin de répertorier les déclencheurs OnBuild répertoriés pour l'image …  
**>> docker image inspect <Image-parent>**

La commande renvoit une sortie similaire à ce qui suit …  
**...  
"OnBuild": [  
 "CMD [\"echo\",\"Exécution de la directive ONBUILD\"]"  
]  
...**

On crée un nouveau répertoire nommé onbuild-parent …  
**>> mkdir --verbose onbuild-parent  
mkdir: création du répertoire 'onbuild-parent'**

On se déplace dans le répertoire …  
**>> cd onbuild-parent**

On crée un Dockerfile …  
**>> nano|vim Dockerfile  
# Exemple de directive ONBUILD  
FROM ubuntu  
RUN apt-get update && apt-get install apache2 -y  
ONBUILD COPY \*.html /var/www/html  
EXPOSE 80  
ENTRYPOINT ["apache2ctl", "-D", "FOREGROUND"]**

On construit l’image à l’aide de la commande **docker image build** …  
**>> docker image build --tag onbuild-parent .  
Sending build context to Docker daemon 2.048kB  
…  
Successfully built 6904aa6cdeb8  
Successfully tagged onbuild-parent:latest**

On exécute la commande **docker container run** pour démarrer un nouveau conteneur à partir de l'image …  
**>> docker container run --publish 80:80 --name onbuild-parent-conteneur --detach onbuild-parent  
dd96664a00706edc62df1804bc5c0c3d5ae74ac3cd258fcc0de9a2402ef80ad0**

Il est maintenant possible de pouvoir accéder à la page Web du contenur.

On arrête et supprime le conteneur …  
**>> docker container rm --force onbuild-parent-conteneur  
onbuild-parent-conteneur**

On crée une autre image Docker en utilisant onbuild-parent-conteneur comme image parent pour déployer une page d'accueil HTML personnalisée.

On crée un nouveau répertoire nommé onbuild-parent …  
**>> mkdir --verbose ../onbuild-enfant  
mkdir: création du répertoire 'onbuild-enfant'**

On se déplace dans le répertoire …  
**>> cd ../onbuild-enfant**

On crée un fichier nommé index.html …  
**>> nano|vim index.html  
<html>  
 <body>  
 <h1>Exemple de la directive Docker ONBUILD</h1>  
 </body>  
</html>**

On crée un Dockerfile …  
**>> nano|vim Dockerfile  
# Exemple de directive ONBUILD (2)  
FROM onbuild-parent**

Ce Dockerfile n'a qu'une seule directive.

On construit l’image à l’aide de la commande **docker image build** …  
**>> docker image build --tag onbuild-enfant .  
Sending build context to Docker daemon 3.072kB  
Step 1/1 : FROM onbuild-parent  
# Executing 1 build trigger  
 ---> d6b0b53b18d9  
Successfully built d6b0b53b18d9  
Successfully tagged onbuild-enfant:latest**

On exécute la commande **docker container run** pour démarrer un nouveau conteneur à partir de l'image …  
**>> docker container run --publish 80:80 --name onbuild-enfant-conteneur --detach onbuild-enfant  
8819d872b825817b8138c9fea19398f26e4376d1a5194907c93cfc4cc47a8bf2**

On arrête et supprime le conteneur …  
**>> docker container rm --force onbuild-enfant-conteneur  
onbuild-enfant-conteneur**