FICHE 1 : Apport théorique pour la conteneurisation par DOCKER

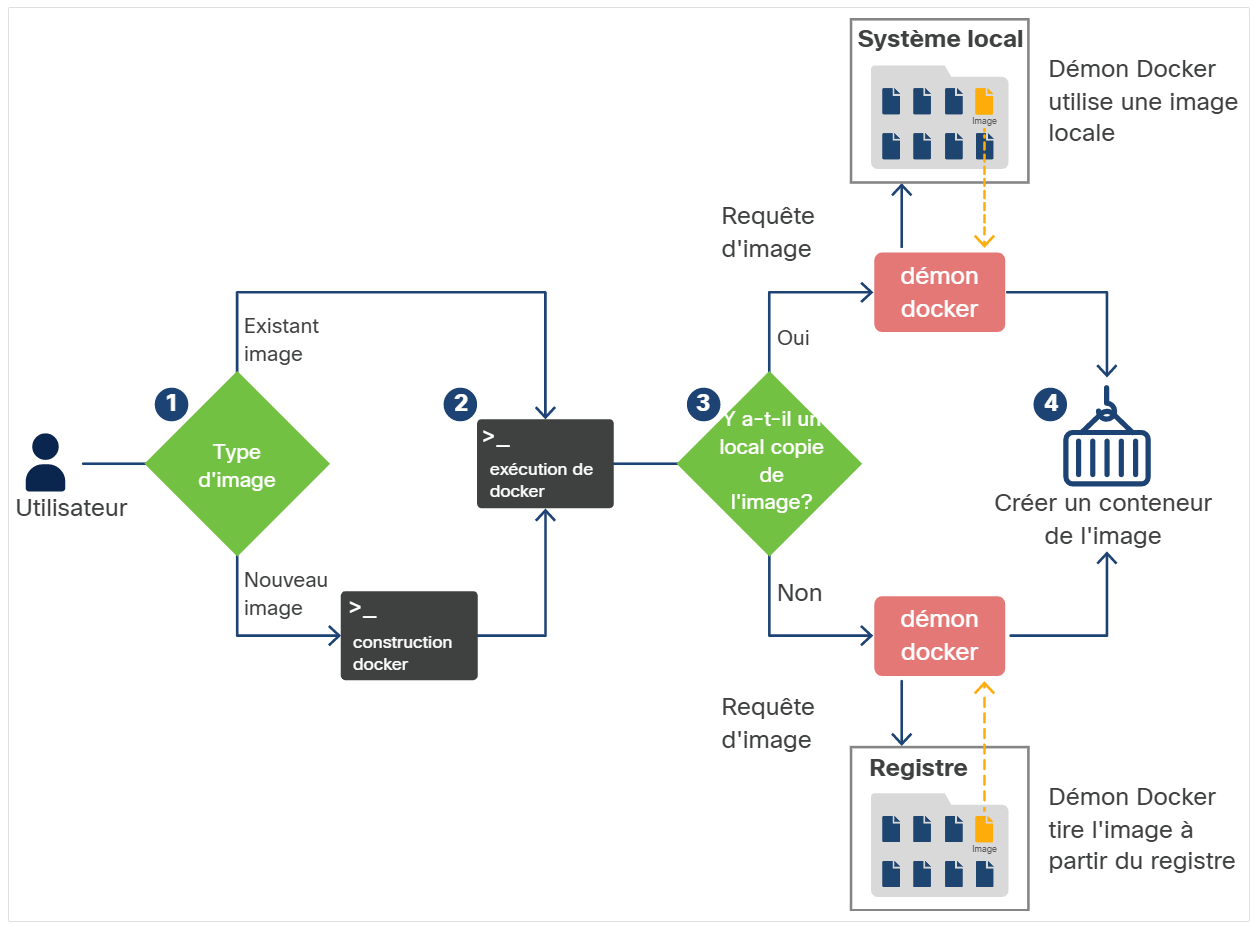
Sources : Cisco Netacad (DevAsc)

Qu' est-ce que Docker ?Le moyen le plus populaire de conteneuriser une application est de la déployer en tant que conteneur Docker. Un conteneur est un moyen d'encapsuler tout ce dont vous avez besoin pour exécuter votre application, afin qu'elle puisse être facilement déployée dans une variété d'environnements. Docker est un moyen de créer et d'exécuter ce conteneur. Plus précisément, Docker est un format qui enveloppe un certain nombre de technologies différentes pour créer ce que nous connaissons aujourd'hui sous le nom de conteneurs. Ces technologies sont les suivantes :

* **Espaces de noms** - Ces espaces isolent différentes parties du conteneur en cours d'exécution. Par exemple, le processus lui-même est isolé dans l'espace de noms pid (ID de processus), le système de fichiers est isolé dans l'espace de noms mnt (mount) et la mise en réseau est isolée dans l'espace de noms réseau.
* **Groupes de contrôle** - Ces cgroups sont un concept Linux standard qui permet au système de limiter les ressources, telles que la RAM ou le stockage, utilisées par une application.
* **Systèmes de fichiers de l'Union** - Ces "UnionFS" sont des systèmes de fichiers qui sont construits couche par couche, en combinant les ressources.

Une image Docker est un ensemble de fichiers en lecture seule qui n'a pas d'état. Une image Docker contient du code source, des bibliothèques et d'autres dépendances nécessaires à l'exécution d'une application. Un conteneur Docker est l'instance d'exécution d'une image Docker. Vous pouvez avoir de nombreux conteneurs en cours d'exécution de la même image Docker. Une image Docker est comme une recette pour un gâteau, et vous pouvez faire autant de gâteaux (conteneurs Docker) que vous le souhaitez.

Les images peuvent à leur tour être stockées dans des registres tels que Docker Hub. Dans l'ensemble, le système ressemble à ceci :



La création d'un conteneur implique d’utiliser une image, ou un modèle, d'un dépôt, puis de l'utiliser pour créer un conteneur.

Une version simplifiée du flux de travail de création d'un conteneur ressemble à ceci :

**Étape 1.** Créez une nouvelle image à l'aide de docker build ou tirez une copie d'une image existante d'un registre à l'aide de docker pull. (Selon les circonstances, cette étape est facultative. Voir l'étape 3.)

**Étape 2.**Exécutez un conteneur basé sur l'image en utilisant docker run ou docker container create.

**Étape 3.** Le démon Docker vérifie s'il possède une copie locale de l'image. Si ce n'est pas le cas, il extrait l'image du registre.

**Étape 4.** Le démon Docker crée un conteneur basé sur l'image et, si docker run a été utilisé, se connecte à celui-ci et exécute la commande demandée.

Comme vous pouvez le voir, si vous voulez créer un déploiement basé sur le conteneur de l'exemple d'application, vous devrez créer une image. Pour ce faire, vous avez besoin d'un Dockerfile.

## Qu' est-ce qu'un Dockerfile ?

Si vous avez déjà utilisé un langage de codage tel que C, vous savez qu'il vous a fallu compiler votre code. Si c'est le cas, vous connaissez peut-être le concept d'un "makefile". C'est le fichier que l'utilitaire make utilise pour compiler et construire toutes les pièces de l'application.

C'est ce qu'un Dockerfile fait pour Docker. Il s'agit d'un fichier texte simple, nommé Dockerfile. Il définit les étapes que la commande docker build doit suivre pour créer une image qui peut ensuite être utilisée pour créer le conteneur cible.

Vous pouvez créer un Dockerfile très simple qui crée un conteneur Ubuntu. Utilisez la commande cat pour créer un Dockerfile, puis ajoutez From ubuntu au fichier. Entrez Ctrl+D pour enregistrer et quitter le fichier avec le texte suivant et l'enregistrer dans votre répertoire actuel :

cned@labdock:~$ **cat > Dockerfile**

**FROM ubuntu:latest**

**<Ctrl+D>**

cned@labdock:~$

C'est tout ce qu'il faut, juste cette ligne. Maintenant, vous pouvez utiliser la commande docker build pour construire l'image comme indiqué dans l'exemple suivant. L'option -t est utilisée pour nommer la construction. Notez le point (.) à la fin de la commande qui spécifie que l'image doit être construite dans le répertoire courant. Vous pouvez utiliser docker build —help pour voir toutes les options disponibles.

cned@labdock:~$ **docker build -t myubuntu:latest .**

Sending build context to Docker daemon 983.3MB

Step 1/1 : FROM ubuntu:latest

latest: Pulling from library/ubuntu

692c352adcf2: Pull complete

97058a342707: Pull complete

2821b8e766f4: Pull complete

4e643cc37772: Pull complete

Digest: sha256:55cd38b70425947db71112eb5dddfa3aa3e3ce307754a3df2269069d2278ce47

Status: Downloaded newer image for ubuntu:latest

---> adafef2e596e

Successfully built adafef2e596e

Successfully tagged myubuntu:latest

cned@labdock:~$

Entrez la commande docker images pour voir la liste des images sur la machine utilisée.

cned@labdock:~$ **docker images**

REPOSITORY TAG IMAGE ID CREATED SIZE

myubuntu latest adafef2e596e 3 days ago 73.9MB

ubuntu latest adafef2e596e 3 days ago 73.9MB

cned@labdock:~$

Maintenant que vous avez l'image, vous pouvez utiliser la commande docker run pour l'exécuter. Vous êtes maintenant dans un shell bash INTÉRIEUR de l'image docker que vous avez créée. Vous pouvez changer de répertoire de base et entrez ls pour voir qu'il est vide et prêt à l'emploi. Entrez exit pour quitter le conteneur Docker et revenir au système d'exploitation principal de votre machine.

cned@labdock:~$ **docker run -it myubuntu:latest /bin/sh**

# **ls**

bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp usr var

# **cd home**

# **ls**

# **exit**

cned@labdock:~$

## Anatomie d'un Dockerfile

Bien sûr, si tout ce que vous pouviez faire avec un Dockerfile était de démarrer un système d'exploitation propre, ce serait utile, mais ce dont vous avez besoin est un moyen de commencer avec un modèle et de construire à partir de là.

Considérez le Dockerfile suivant qui conteneurise une application Python

FROM python

WORKDIR /home/ubuntu

COPY ./sample-app.py /home/ubuntu/.

RUN pip install flask

CMD python /home/ubuntu/sample-app.py

EXPOSE 8080

Voici une explication des commandes utilisées dans le Dockerfile ci-dessus.

* La commande FROM installe Python dans l'image Docker. Il invoque une image par défaut basée sur Debian Linux à partir de Docker Hub, avec la dernière version de Python installée.
* La commande WORKDIR indique à Docker d'utiliser /home/ubuntu comme répertoire de travail
* La commande COPY indique à Docker de copier le fichier le **sample-app.py** du répertoire courant de Dockerfile dans /home/ubuntu.
* La commande RUN  vous permet d'exécuter directement des commandes sur le conteneur. Dans cet exemple, Flask est installé. Flask est une serveur web.
* La commande CMD lancer la commande python pour exécuter **sample-app.py** à l'intérieur du conteneur.
* La commande EXPOSE indique à Docker que vous souhaitez exposer le port 8080. Notez qu'il s'agit du port sur lequel Flask écoute. Si vous avez configuré votre serveur Web pour écouter ailleurs (par exemple, les requêtes https sur le port 443), c'est l'endroit où le noter.

Utilisez la commande docker build pour construire l'image. Dans la sortie suivante, l'image a été précédemment construite. Par conséquent, Docker tire parti de ce qui est stocké dans le cache pour accélérer le processus.

cned@labdock:~$ **docker build -t sample-app-image .**

Sending build context to Docker daemon 3.072kB

Step 1/6 : FROM python

---> 0a3a95c81a2b

Step 2/6 : WORKDIR /home/ubuntu

---> Using cache

---> 17befcf89bab

Step 3/6 : COPY ./sample-app.py /home/ubuntu/.

---> Using cache

---> c0b3a4f9c568

Step 4/6 : RUN pip install flask

---> Using cache

---> 8cf8226c9f31

Step 5/6 : CMD python /home/ubuntu/sample-app.py

---> Running in 267c5d569356

Removing intermediate container 267c5d569356

---> 75cd4bf1d02a

Step 6/6 : EXPOSE 8080

---> Running in cc82eaca2028

Removing intermediate container cc82eaca2028

---> 9616439582f8

Successfully built 9616439582f8

Successfully tagged sample-app-image:latest

cned@labdock:~$

Comme vous pouvez le voir, Docker passe en revue chaque étape du Dockerfile, en commençant par l'image de base, Python. Si cette image n'existe pas sur votre système, Docker la retire du Registre. Le registre par défaut est Docker Hub. Toutefois, dans un environnement sécurisé, vous pouvez configurer votre propre registre d'images de conteneur de confiance. Notez que l'image est en fait un certain nombre d'images différentes superposées les unes sur les autres, tout comme vous superposez vos propres commandes au-dessus de l'image de base.

Notez qu'entre les étapes telles que l'exécution d'une commande, Docker crée réellement un nouveau conteneur et construit une image intermédiaire, une nouvelle couche, en enregistrant ce conteneur. En fait, vous pouvez le faire vous-même en créant un conteneur, en effectuant les modifications souhaitées, puis en enregistrant ce conteneur en tant que nouvelle image.

Dans l'exemple précédent, seul un petit nombre de commandes Dockerfile disponibles ont été utilisées. La liste complète est disponible dans la documentation Docker dans la référence Dockerfile. En voici une liste non exhaustive :

* FROM
* MAINTAINER
* RUN
* CMD
* EXPOSE
* ENV
* COPY
* ENTRYPOINT
* VOLUME
* USER
* WORKDIR
* ARG
* ONBUILD
* STOPSIGNAL
* LABEL

## Démarrer un conteneur Docker localement

Maintenant que l'image est créée, vous pouvez l’utiliser pour créer un nouveau conteneur en entrant la commande docker run, comme indiqué dans la sortie suivante. Dans ce cas, plusieurs paramètres sont spécifiés. Le paramètre -d est abrégé pour —detach et indique que vous voulez l'exécuter en arrière-plan.   
Le paramètre -P indique à Docker de le publier sur les ports que vous avez exposés (dans ce cas, 8080).

$ **docker run -d -P sample-app-image**

1688a2c34c9e7725c38e3d9262117f1124f54685841e97c3c5225af88e30bfc5

$

Vous pouvez maintenant voir le conteneur en listant les processus :

$ **docker ps**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

90edd03a9511 sample-app-image "/bin/sh -c 'python …" 5 seconds ago Up 3 seconds 0.0.0.0:32774->8080/tcp jovial\_sammet

$

Notez que Docker a attribué au conteneur un nom, jovial\_sammet. Vous auriez également pu le nommer vous-même avec l'option —name. Par exemple :

docker run -d -P —name pythontest sample-app-image

Notez également que, même si le conteneur écoute sur le port 8080, ce n'est qu'un port interne. Docker a spécifié un port externe, dans ce cas 32774, qui transmettra à ce port interne. Cela vous permet d'exécuter plusieurs conteneurs qui écoutent sur le même port sans avoir de conflits. Si vous souhaitez extraire votre exemple de site Web d'application, vous pouvez utiliser l'adresse IP publique du serveur hôte et de ce port. Sinon, si vous deviez l'appeler à partir de la machine hôte elle-même, vous utiliseriez toujours ce port externalisé, comme indiqué avec la commande curl suivante.

$ **curl localhost:32774**

You are calling me from 172.17.0.1

$

Docker vous permet également de spécifier un port particulier à transférer,

$ **docker run -d -p 8080:8080 --name pythontest sample-app-image**

$ **docker ps**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

a51da037bf35 sample-app-image "/bin/sh -c 'python …" 28 seconds ago Up 27 seconds 0.0.0.0:8080->8080/tcp pythontest

90edd03a9511 sample-app-image "/bin/sh -c 'python …" 24 minutes ago Up 24 minutes 0.0.0.0:32774->8080/tcp jovial\_sammet

$

Lorsque votre conteneur est en cours d'exécution, vous pouvez vous y connecter comme vous le feriez pour n'importe quel hôte physique ou virtuel à l'aide de la commande exec de l'hôte sur lequel le conteneur s'exécute:

$ **docker exec -it pythontest /bin/sh**

# **whoami**

root

# **pwd**

/var/www/html

# **exit**

$

Pour arrêter et supprimer un conteneur en cours d'exécution, vous pouvez utiliser son nom :

$ **docker stop pythontest**

pythontest

$ **docker rm pythontest**

pythontest

$

Si vous regardiez les processus en cours d'exécution, vous pourriez voir qu'il a disparu.

$ **docker ps**

## Enregistrer une image Docker dans un registre

Maintenant que vous savez comment créer et utiliser votre image, il est temps de la rendre disponible pour d'autres personnes. Une façon de le faire est de la stocker dans un registre d'images.

Par défaut, Docker utilise le registre Docker Hub, mais vous pouvez créer et utiliser votre propre registre.   
Vous devez commencer par vous connecter au registre :

$ **docker login**

Login with your Docker ID to push and pull images from Docker Hub. If you don&apos;t have a Docker ID, head over to https://hub.docker.com to create one.

Username: **devnetstudent** # This would be your username

Password: # This would be your password

WARNING! Your password will be stored unencrypted in /home/ubuntu/.docker/config.json.

Configure a credential helper to remove this warning. See

https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded

$

Ensuite, vous validez une instance de conteneur en cours d'exécution de votre image. Par exemple, le conteneur pythontest est en cours d'exécution dans cet exemple. Valider le conteneur avec la commande docker commit.

$ **docker ps**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

54c44606344c sample-app-image &quot;/bin/sh -c &apos;python …&quot; 4 seconds ago Up 2 seconds 0.0.0.0:8080->8080/tcp pythontest

$ **docker commit pythontest sample-app**

Sha256:bddc326383032598a1c1c2916ce5a944849d90e4db0a34b139eb315af266e68b

$

Ensuite, utilisez la commande docker tag pour donner une balise à l'image que vous avez commitée. La balise prend la forme suivante :

<repository>/<imagename>: <tag>

La première partie, le référentiel, est généralement le nom d'utilisateur du compte stockant l'image. Dans cet exemple : **cnedstudent**. Ensuite est le nom de l'image, puis enfin la balise facultative.

Dans cet exemple, la balise pourrait être **v1**, comme illustré ici:

$ **docker tag sample-app devnetstudent/sample-app:v1**

$

Maintenant, l'image est prête à être poussée dans le référentiel :

$ **docker push devnetstudent/sample-app:v1**

The push refers to repository [docker.io/nickchase/sample-app]

e842dba90a43: Pushed

868914f88a69: Pushed

c7d71f6230b3: Pushed

1ed9b15dd229: Pushed

00947a3aa859: Mounted from library/python

7290ddeeb6e8: Mounted from library/python

d3bfe2faf397: Mounted from library/python

cecea5b3282e: Mounted from library/python

9437609235f0: Mounted from library/python

bee1c15bf7e8: Mounted from library/python

423d63eb4a27: Mounted from library/python

7f9bf938b053: Mounted from library/python

f2b4f0674ba3: Mounted from library/python

v1: digest: sha256:28e119f43e9c8e5e44f167d9baf113cc91d4f8b461714cd6bb578ebb0654f243 size: 3052

$

De là, vous pouvez voir que la nouvelle image est stockée localement :

$ **docker images**

REPOSITORY TAG IMAGE ID CREATED SIZE

sample-app latest bddc32638303 About a minute ago 410MB

devnetstudent/sample-app v1 bddc32638303 About a minute ago 410MB

$