DOCKER — GESTION DES IMAGES

Images • couches • Dockerfile • Docker Hub • build

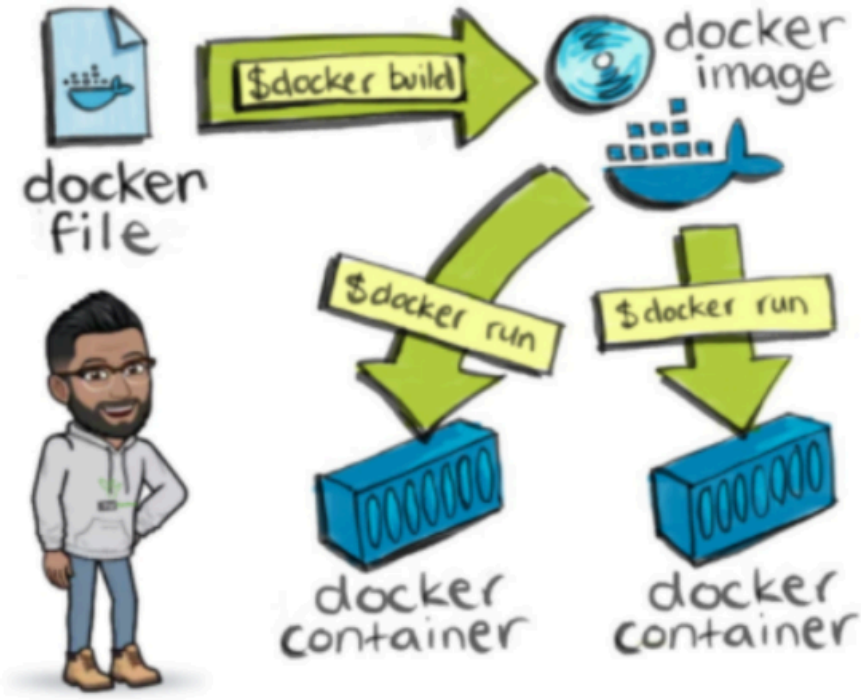
Objectif — Comprendre ce qu'est une image Docker, comment elle est constituée, comment la manipuler et comment construire une image personnalisée avec un Dockerfile.

1. Qu'est-ce qu'une image Docker ?

Une image Docker est le modèle immuable à partir duquel Docker crée des conteneurs. Elle contient le système de fichiers et les métadonnées nécessaires à l'exécution d'une application.

- Elle contient le code de l'application, ses bibliothèques, ses dépendances et ses fichiers de configuration.
- Elle est immuable : pour la modifier, on construit une nouvelle image.
- Docker et les registres utilisent des digests de contenu, généralement en SHA-256, pour identifier et vérifier le contenu.
- Une image peut être référencée par un nom et un tag (ex. httpd:latest) ou par un digest (ex. image@sha256:…).
- Elle est le plus souvent construite à partir d'un fichier nommé Dockerfile.

À retenir — Image = modèle en lecture seule. Conteneur = instance créée à partir de l'image avec une couche d'écriture propre au conteneur.

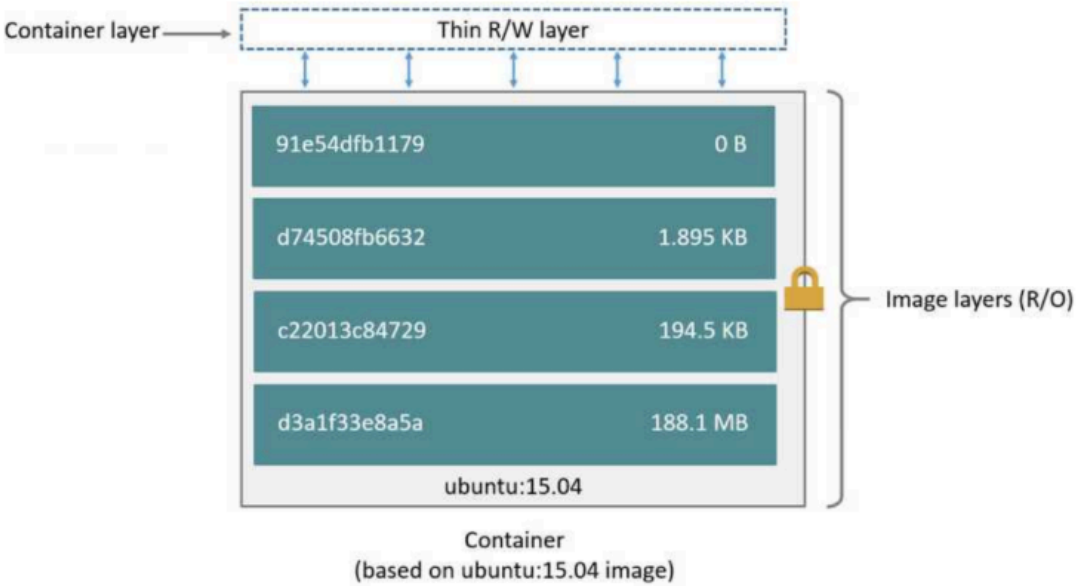


Du Dockerfile à l'image, puis de l'image aux conteneurs : un même modèle peut lancer plusieurs conteneurs.

2. Construction par couches (layers)

Docker construit une image étape par étape. Les instructions qui modifient le système de fichiers produisent des changements enregistrés sous forme de couches ; d'autres instructions modifient surtout les métadonnées de l'image.

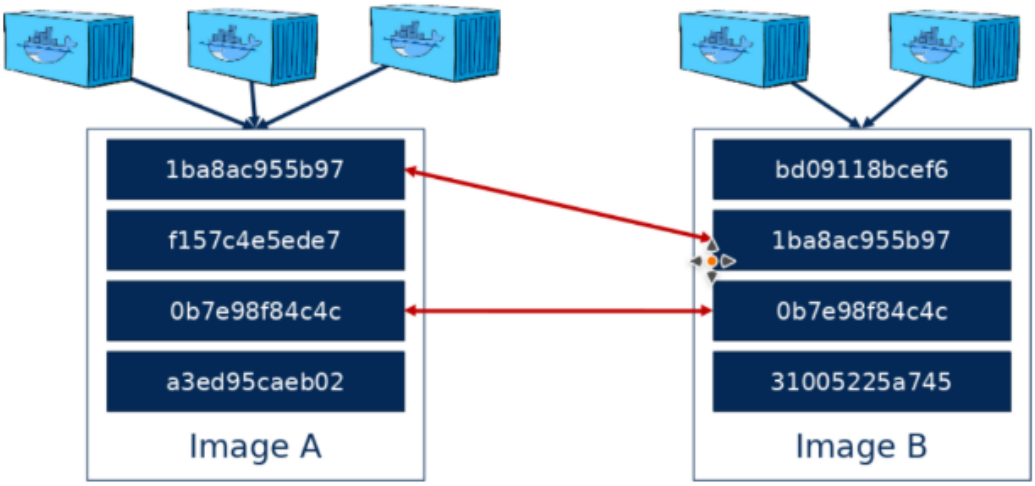
- Les couches de l'image sont en lecture seule (R/O).
- Au démarrage d'un conteneur, Docker ajoute au-dessus une fine couche propre au conteneur, en lecture/écriture (R/W).
- Les modifications faites pendant l'exécution sont écrites dans cette couche R/W, pas dans les couches de l'image.
- Les storage drivers (par exemple overlay2/OverlayFS) présentent une vue unifiée de ces couches : c'est le principe d'un système de fichiers en union.
- La couche R/W disparaît avec le conteneur : les données importantes doivent être placées dans des volumes ou des bind mounts.



Toutes les couches de l'image sont en lecture seule ; chaque conteneur reçoit une couche R/W supplémentaire.

Partage des couches

Les couches sont réutilisables. Plusieurs images peuvent partager des couches identiques, et plusieurs conteneurs issus de la même image partagent les mêmes couches R/O. Cela économise de l'espace disque et accélère les téléchargements/builds lorsque les couches sont déjà présentes localement.



Exemple de couches communes réutilisées entre plusieurs images.

3. Exemple : image Apache

Le support montre un Dockerfile simple basé sur l'image officielle httpd. L'image de base possède déjà la configuration et la commande de démarrage d'Apache ; l'exemple sert surtout à comprendre les instructions.

Exemple basé sur httpd

```
FROM httpd:latest
LABEL maintainer="VotreNom <votre@email.com>"
EXPOSE 80
EXPOSE 443
CMD ["httpd-foreground"]
```

Élément	Rôle
FROM httpd:latest	Utilise l'image httpd comme base du build.
LABEL maintainer=…	Ajoute une métadonnée. Pour l'auteur, le format OCI org.opencontainers.image.authors est aujourd'hui préférable.
EXPOSE 80 / 443	Documente les ports que l'application est censée écouter. Cela ne publie pas les ports sur l'hôte.
CMD ["httpd-foreground"]	Définit la commande par défaut au démarrage du conteneur. Elle peut être remplacée par des arguments fournis à docker run.

Important — EXPOSE n'ouvre pas un port sur l'hôte. Pour publier un port, on utilise par exemple : docker run -p 8080:80 image. De même, EXPOSE 443 ne configure pas HTTPS/TLS : l'application doit réellement être configurée pour écouter en TLS.

Même idée en partant de Debian

Exemple pédagogique corrigé

```
FROM debian:latest
RUN apt-get update \
 && apt-get install -y --no-install-recommends apache2 \
 && rm -rf /var/lib/apt/lists/*
EXPOSE 80
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

Ici, Docker part d'une image Debian, installe Apache pendant le build, puis définit la commande qui garde Apache au premier plan. Le nettoyage de /var/lib/apt/lists réduit la taille de l'image finale.

4. Espace disque, données et logs

Taille et couches partagées

- Une image occupe de l'espace pour ses couches R/O.
- Chaque conteneur ajoute sa propre couche R/W, généralement petite si l'application écrit peu dedans.
- Les couches R/O sont partagées : additionner les « tailles virtuelles » de plusieurs conteneurs peut donc compter plusieurs fois les mêmes données partagées.
- Pour observer l'espace utilisé : docker system df. Pour voir la couche écrivable d'un conteneur : docker ps --size.

Persistance des données

Il est normal de stocker le programme et ses fichiers statiques dans l'image. En revanche, les données qui doivent survivre à la suppression/recréation du conteneur ne doivent pas dépendre de sa couche R/W.

- Volumes Docker : mécanisme recommandé pour les données persistantes gérées par Docker.
- Bind mounts : montage d'un fichier ou répertoire réel de l'hôte dans le conteneur.
- Montages réseau / systèmes de fichiers distribués : possibles selon l'architecture, avec des contraintes de performance, disponibilité et migration.

À retenir — Un conteneur peut être supprimé et recréé ; un volume correctement monté conserve ses données indépendamment de la vie du conteneur.

Logs

Pour les applications conteneurisées, la pratique courante est d'écrire les logs sur stdout/stderr et de laisser Docker ou la plateforme de logs les collecter via un logging driver. Si une application écrit obligatoirement dans des fichiers, on peut monter un volume/bind mount et prévoir une rotation adaptée.

5. Docker Hub et registres

Docker Hub est un registre hébergé permettant de stocker, partager et distribuer des images Docker. Il contient notamment les Docker Official Images et permet aussi d'héberger ses propres dépôts.

- Pull : récupérer une image depuis un registre.
- Push : envoyer une image dans un registre, après authentification et avec un nom/tag correspondant au dépôt cible.
- Des registres privés auto-hébergés existent également, par exemple avec Docker Distribution ou d'autres solutions de registre compatibles OCI.
- Docker Build Cloud est un service de build distant : ce n'est pas un registre local.

Mise à jour 2026 — Les Docker Hub Automated Builds liés à GitHub/Bitbucket existent encore mais sont dépréciés et annoncés pour retrait le 1er avril 2027. Pour de nouveaux workflows, privilégier les CI comme GitHub Actions ou Bitbucket Pipelines.

6. Commandes utiles pour les images

Objectif	Commande fiable	Explication
Lister les images	docker image ls	Affiche les images locales. docker images est un alias historique encore valide.
Télécharger une image	docker pull nom:tag	Télécharge l'image depuis le registre configuré (Docker Hub par défaut si aucun registre n'est précisé).
Supprimer une image	docker rmi nom:tag	Supprime/détache l'image locale. Équivalent : docker image rm.
Envoyer une image	docker push utilisateur/image:tag	Envoie l'image vers un registre. Il faut généralement être connecté et avoir correctement tagué l'image.
Sauvegarder en TAR	docker save -o /chemin/image.tar nom:tag	Exporte une ou plusieurs images, avec leurs couches et tags, dans une archive TAR.
Charger depuis un TAR	docker load -i /chemin/image.tar	Importe une archive créée avec docker save et restaure les images/tags.
Voir l'historique	docker image history nom:tag	Affiche l'historique des étapes/couches de construction de l'image.

Correction du support — docker push ne prend pas un « répertoire » comme second argument. docker load ne prend pas non plus un nom d'image supplémentaire : les tags sont restaurés depuis l'archive.

7. Qu'est-ce qu'un Dockerfile ?

Un Dockerfile est un fichier texte qui décrit les instructions nécessaires pour construire une nouvelle image Docker à partir d'une image de base. Par convention, le fichier s'appelle Dockerfile.

- FROM choisit l'image de départ.
- RUN installe/configure des éléments pendant la construction.
- COPY ou ADD ajoute des fichiers au contexte de l'image.
- ENV, LABEL, EXPOSE, USER, WORKDIR… configurent des métadonnées ou le comportement de l'image.
- CMD et ENTRYPOINT définissent la commande exécutée lorsque le conteneur démarre.

Une image existante n'est pas modifiée « en place » : on relance un build qui produit une nouvelle image avec un nouvel identifiant/digest.

8. Instructions principales du Dockerfile

Instruction	Rôle correct
FROM	Choisit l'image de base et démarre une nouvelle étape de build.
RUN	Exécute une commande pendant le build ; les modifications du système de fichiers sont intégrées à l'image.
CMD	Définit la commande ou les arguments par défaut du conteneur. Les arguments de docker run peuvent la remplacer.
ENTRYPOINT	Définit l'exécutable principal du conteneur. Les arguments de docker run s'ajoutent généralement à l'ENTRYPOINT ; il reste possible de le remplacer avec --entrypoint.
EXPOSE	Documente les ports d'écoute prévus. Ne publie aucun port à lui seul.
ENV	Définit une variable d'environnement qui persiste dans l'image et est disponible au runtime, sauf surcharge.
ARG	Définit une variable disponible pendant le build. Elle peut recevoir une valeur via --build-arg.
COPY	Copie fichiers/répertoires depuis le contexte de build vers l'image. À privilégier pour une copie simple.
ADD	Ajoute des fichiers et possède des fonctions supplémentaires (certaines sources distantes / extraction de certaines archives). Utiliser COPY si ces fonctions ne sont pas nécessaires.

Instruction	Rôle correct
LABEL	Ajoute des métadonnées à l'image (auteur, version, description, etc.).
VOLUME	Déclare un point de montage destiné à recevoir un volume externe au runtime. Le chemin hôte n'est pas défini dans le Dockerfile.
WORKDIR	Définit le répertoire de travail pour les instructions suivantes telles que RUN, CMD, ENTRYPOINT, COPY et ADD.
USER	Définit l'utilisateur (et éventuellement le groupe) utilisé pour les RUN suivants et, par défaut, pour CMD/ENTRYPOINT au runtime.
ONBUILD	Déclare une instruction déclenchée plus tard lorsqu'une autre image utilise cette image comme base.
HEALTHCHECK	Exécute périodiquement une commande afin de déterminer l'état de santé du conteneur (code 0 = healthy, 1 = unhealthy).
STOPSIGNAL	Définit le signal envoyé pour demander l'arrêt du conteneur.

Instruction	Rôle correct
SHELL	Change le shell par défaut utilisé par les instructions en forme shell.
MAINTAINER	Dépréciée. Utiliser LABEL, par exemple org.opencontainers.image.authors.

Exemple ARG

```
ARG VERSION=latest
FROM ubuntu:${VERSION}

# Construction avec une autre valeur :
# docker build --build-arg VERSION=24.04 -t mon_image .
```

Exemple HEALTHCHECK

```
HEALTHCHECK --interval=5m CMD curl -f http://localhost/ || exit 1
```

9. Construire une image personnalisée

Commande de base

```
docker build -t test .
```

Élément	Signification
docker	Client Docker.
build	Demande la construction d'une image à partir d'un contexte et d'un Dockerfile.
-t	Tag : donne un nom (et éventuellement un tag) à l'image produite.
test	Nom choisi pour l'image ; sans tag explicite, Docker utilise généralement test:latest comme référence locale.
.	Contexte de build = dossier courant. Docker y cherche par défaut un fichier nommé Dockerfile.

Si le Dockerfile porte un autre nom :

```
docker build -f Dockerfile.prod -t test .
```

Options de docker build vues avec le professeur

Option	Explication
-t, --tag	Spécifie le nom et éventuellement le tag de l'image. Exemple : docker build -t monapp:1.0 .

Option	Explication
-m, --memory	Définit une limite de mémoire pour la construction avec le builder historique. Exemple : -m 500M ou -m 2G.
--rm	Supprime les conteneurs intermédiaires après un build réussi avec le builder historique. Ne supprime pas l'image finale. Valeur par défaut : true.
-f, --file	Spécifie le Dockerfile à utiliser s'il ne porte pas le nom conventionnel Dockerfile. Exemple : docker build -f Dockerfile.prod -t monapp .

À retenir pour le TP — ce tableau reprend les quatre options montrées par le professeur. Correction de syntaxe : écrire --rm avec deux tirets. Correction de sens : --rm ne supprime pas l'image finale, mais les conteneurs intermédiaires du builder historique après un build réussi. Avec Docker actuel, docker build utilise BuildKit/Buildx par défaut : -t/--tag et -f/--file restent des options courantes ; -m/--memory et --rm appartiennent à la référence du builder historique et ne figurent pas dans la référence actuelle docker buildx build. Pour savoir exactement ce que la machine du TP accepte, utiliser docker build --help. Avec Buildx récent, une limite de ressources peut être définie avec --resource, par exemple : docker buildx build --resource memory=2g .

10. Nettoyage : docker system prune

```
docker system prune
```

Cette commande ne « remet pas Docker à zéro ». Elle supprime les données Docker inutilisées : conteneurs arrêtés, réseaux inutilisés, cache de build et images inutilisées selon les options. Elle doit donc être utilisée volontairement et après lecture du récapitulatif affiché.

- docker system prune -a → supprime aussi les images inutilisées qui ne sont pas seulement dangling.
- docker system prune --volumes → inclut certains volumes anonymes inutilisés ; attention aux données.

Prudence — Ne pas utiliser prune comme commande réflexe en environnement contenant des données importantes.

11. Mémo express

Action	Réflexe
Créer une image	Dockerfile → docker build -t nom:tag .
Voir les images	docker image ls
Télécharger	docker pull nom:tag

Action	Réflexe
Lancer un conteneur	docker run nom:tag
Publier un port	docker run -p PORT_HOTE:PORT_CONTENEUR nom:tag
Exporter une image	docker save -o image.tar nom:tag
Importer une image	docker load -i image.tar
Publier au registre	docker tag … puis docker push utilisateur/image:tag
Persister des données	Volume ou bind mount, pas uniquement la couche R/W du conteneur

Sources officielles vérifiées

Dockerfile reference : https://docs.docker.com/reference/dockerfile/
Understanding image layers :
https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/
Docker storage : https://docs.docker.com/engine/storage/
Docker image CLI : https://docs.docker.com/reference/cli/docker/image/
docker image build (options du builder historique) : https://docs.docker.com/reference/cli/docker/image/build/

docker buildx build : https://docs.docker.com/reference/cli/docker/buildx/build/
Docker Hub : https://docs.docker.com/docker-hub/
Docker Hub Automated Builds — deprecation : https://docs.docker.com/docker-hub/repos/manage/builds/
docker system prune : https://docs.docker.com/reference/cli/docker/system/prune/