



people-in-boxes.webp

Explorer et fabriquer des images

Prévoyez du disque et du réseau

Ce TP télécharge une dizaine d'images. Comptez deux à trois gigaoctets. Si vous êtes trop juste, la Partie 9 vous apprend justement à faire le ménage — vous pouvez y jeter un œil en cours de route.

Ce qu'on cherche à obtenir

Le cours affirme beaucoup de choses sur les images : qu'elles sont des piles de couches, qu'un tag n'est qu'une étiquette, que supprimer un fichier n'allège rien, qu'une image forcée hors de son architecture tourne émulée. Tout cela se vérifie en ligne de commande, et c'est ce qu'on va faire.

Dans la seconde moitié du TP, vous fabriquerez votre première image — avec `commit`, une méthode avec ses limites, précisément pour comprendre pourquoi le `Dockerfile` existe — puis vous rendrez cette image "portable" sans passer par aucun registry.

Toujours `--help` d'abord

Les commandes d'images ont beaucoup d'options, et plusieurs existent en deux écritures (`docker images` et `docker image ls` font la même chose). Prenez le réflexe :

```
docker image --help
docker image history --help
```

Notez vos chiffres

Plusieurs parties vous demandent de comparer des tailles, des nombres de couches et des durées. Gardez un bloc-notes ouvert : les comparaisons ne valent que si vous avez les deux chiffres.

Partie 1 — L'image ne change jamais

Au chapitre précédent, vous avez modifié la page d'accueil de Nginx et constaté qu'elle disparaissait avec le conteneur. Regardons la même chose d'un autre angle : deux conteneurs issus d'une seule image.

Lancez-en deux, sur deux ports différents :

```
docker run -d --name web-a -p 8080:80 nginx
docker run -d --name web-b -p 8081:80 nginx
```

Modifiez la page du premier seulement :

```
docker exec web-a sh -c 'echo "<h1>Je suis A</h1>" >
/usr/share/nginx/html/index.html'
```

Ouvrez les deux dans votre navigateur : <http://localhost:8080> et <http://localhost:8081>.

Répondez :

1. Les deux pages sont-elles identiques ?
2. Les deux conteneurs sont issus de la même image. Qu'est-ce qui leur permet de différer ?

Vérifiez combien d'images sont réellement stockées :

```
docker image ls nginx
```

- Combien de lignes ?

Détruisez le premier conteneur et relancez-en un à l'identique :

```
docker rm -f web-a
docker run -d --name web-a -p 8080:80 nginx
```

- Rechargez <http://localhost:8080>. Que voyez-vous ? Où est passé « Je suis A » ?

 Une image est un modèle en lecture seule

Aucun conteneur ne peut la modifier. Chaque conteneur reçoit sa propre couche inscriptible par-dessus, et c'est tout ce qui le distingue de ses frères. Détruire le conteneur détruit cette couche, et seulement elle.

Validation — `docker image ls nginx` affiche une seule ligne, et le nouveau `web-a` sert à nouveau la page d'origine de Nginx.

Partie 2 — Voir la couche inscriptible

La Partie 1 a montré que l'image ne bouge pas. Reste à voir où va ce que le conteneur écrit : dans sa couche inscriptible, qui est observable en direct.

Lancez un conteneur Alpine qui reste en vie, et regardez-le avant toute modification :

```
docker run -dit --name bac alpine sh
docker diff bac
```

- La commande affiche-t-elle quelque chose ? Pourquoi ?

Ajouter un fichier

Connectez-vous à la machine et ajoutez y un fichier avec la commande :

```
dd if=/dev/zero of=/gros bs=1M count=50
```

Relancez ensuite `docker diff`.

Les lettres en début de ligne

`docker diff --help` ne donne que la syntaxe de la commande : la signification des lettres n'y figure pas, et aucune page d'aide locale ne la documente. La voici, une fois pour toutes :

- **A** — *added* : le fichier a été **ajouté** par le conteneur
- **C** — *changed* : le fichier a été **modifié**. Un répertoire passe en **C** dès que son contenu change
- **D** — *deleted* : le fichier a été **supprimé**

- Quelle lettre voyez-vous devant `/gros` ?

Pesez maintenant le conteneur. Il existe une option de `docker ps` qui ajoute une colonne de taille :

```
docker ps --help | grep -i size
```

Lancez `docker ps` avec cette option.

- La colonne affiche deux nombres, dont un entre parenthèses précédé de `virtual`. Lequel correspond à votre fichier de 50 Mo, lequel à l'image ?

Supprimer un fichier venu de l'image

Le fichier `/etc/os-release` appartient à l'image Alpine, donc à une couche en lecture seule. Supprimez-le quand même et refaites un diff :

- Quelles lignes nouvelles apparaissent ? Que signifient leurs lettres ?
- Le fichier était dans une couche **en lecture seule**. Comment Docker a-t-il pu le « supprimer » ?

Le whiteout, vu de l'extérieur

Docker n'a rien supprimé : il a posé dans la couche inscriptible un marqueur qui masque le fichier. L'original est toujours dans la couche de l'image, intact — c'est pour ça qu'un nouveau conteneur le retrouve. C'est aussi pour ça que supprimer un fichier dans une image ne réduit jamais sa taille.

Nettoyez :

```
docker rm -f bac
```

Validation — vous savez produire, pour un conteneur donné, la liste de ce qu'il a ajouté, modifié et supprimé, et expliquer pourquoi une suppression s'y affiche sans rien libérer.

Partie 3 — Compter les couches d'une image

Jusqu'ici, « l'image » tenait en un seul étage, celui du bas. Descendons-y : cet étage est lui-même une pile. On va ouvrir le capot de deux images du même logiciel, mais pas de la même famille.

Récupérez la variante Alpine de Nginx, à côté de celle que vous avez déjà :

```
docker pull nginx:alpine
docker image ls nginx
```

- Quel écart de taille entre les deux ?

Comptez les couches de chacune. L'information est dans la fiche technique, sous `RootFS` :

```
docker image inspect --format '{{len .RootFS.Layers}}' nginx
docker image inspect --format '{{len .RootFS.Layers}}' nginx:alpine
```

- Laquelle a le plus de couches ? Est-ce la plus lourde ?

Regardez maintenant le détail, couche par couche :

```
docker image history nginx:alpine
```

Une ligne par étape de construction, avec sa taille et la commande qui l'a produite. La sortie est large : elle est tronquée à droite, ce n'est pas grave ici.

Répondez :

1. Combien d'étapes ont une taille nulle (0B) ? À votre avis, que contiennent-elles ?
2. Additionnez grossièrement les tailles affichées. Retrouvez-vous l'ordre de grandeur annoncé par `docker image ls` ?

Les étapes de taille nulle

Une étape de construction ne produit pas forcément des fichiers : certaines ne posent que de la **configuration** — une variable d'environnement, un port déclaré, la commande à lancer au démarrage. Elles pèsent zéro octet mais occupent bien une ligne dans l'historique. C'est pour ça que `history` affiche plus de lignes que l'image n'a de couches.

Pourquoi la colonne `IMAGE` affiche `<missing>`

Cette colonne ne contient pas l'identifiant de la couche, mais celui d'une **image**. Et une image, au sens de Docker, c'est un fichier de configuration — la recette : variables d'environnement, commande de démarrage, liste ordonnée des couches. Pas les couches elles-mêmes.

Quand vous tirez `nginx:alpine`, le registry vous envoie les couches, plus **un seul** fichier de configuration : celui de l'image finale. Les étapes intermédiaires n'ont jamais été publiées — elles ne survivent que sous forme de lignes d'historique à l'intérieur de ce fichier unique. Docker peut donc afficher une ligne par étape, mais il n'a d'identifiant à montrer que pour une seule : celle du haut, l'image que vous possédez.

`<missing>` ne veut donc pas dire qu'une couche manque : elles sont toutes là. C'est l'image intermédiaire qui n'a jamais existé chez vous. Vous retrouverez le même affichage en Partie 7, sur l'image que vous fabriquerez vous-même : l'identifiant du haut sera celui de votre `commit`, et tout ce qui vient d'Alpine restera `<missing>`.

Validation — vous savez dire combien de couches contient chacune des deux images, et expliquer pourquoi `history` affiche plus de lignes que `RootFS.Layers` ne compte de couches.

Partie 4 — Tags et digests

Un nom d'image ressemble à un chemin, mais n'en est pas un. On va manipuler les deux façons de désigner une image.

Ce que Docker complète sans le dire

```
docker pull alpine
docker image ls alpine
```

- Quel tag voyez-vous dans la colonne `TAG` ? L'avez-vous demandé ?

Demandez maintenant à Docker l'adresse complète et l'empreinte de ce qu'il a téléchargé :

```
docker image inspect --format '{{.RepoDigests}}' alpine
```

- Combien de parties distinguez-vous dans ce que vous lisez ? Le registry y figure-t-il ?

Un tag n'est qu'une étiquette

Donnez un second nom à cette même image :

```
docker tag alpine mon-alpine:essai
docker image ls | grep -E "alpine|REPOSITORY"
```

- Comparez la colonne `IMAGE ID` des deux lignes. Que constatez-vous ?
- Avez-vous consommé de l'espace disque supplémentaire ?

Supprimez le nom que vous venez de créer :

```
docker rmi mon-alpine:essai
```

- Lisez attentivement ce que répond Docker. Le mot employé est-il « Deleted » ou « Untagged » ?
- `docker image ls alpine` : l'image est-elle toujours là ?

 **rmi supprime un nom, pas forcément une image**

Tant qu'il reste au moins un nom pointant sur une image, `docker rmi` se contente de retirer l'étiquette. L'image n'est réellement effacée que quand plus aucun nom ne la désigne.

Épingler par empreinte

Reprenez l'empreinte affichée plus haut et tirez l'image par son digest plutôt que par son tag. La syntaxe utilise `@` à la place de `:` :

```
docker pull alpine@sha256:LE_DIGEST_QUE_VOUS_AVEZ_LU
```

- Docker a-t-il eu quelque chose à télécharger ? Pourquoi ?
- `docker image ls alpine` : comment la nouvelle ligne est-elle nommée dans la colonne `TAG` ?

Enfin, comparez une version épinglée et `latest` :

```
docker pull alpine:3.19
docker image ls alpine
```

- Les deux lignes ont-elles le même `IMAGE ID` ?
- Dans six mois, lequel de ces deux noms désignera à coup sûr la même image qu'aujourd'hui ?

Validation — vous pouvez expliquer, pour une même image, la différence entre son `IMAGE ID`, son tag et son digest, et laquelle de ces trois désignations est immuable.

Partie 5 — Compter ce qu'il y a dans une image

Le cours annonce un compromis : plus une image est petite, moins elle est confortable à déboguer. Ce confort se compte — en nombre de commandes disponibles.

Récupérez deux images de familles opposées :

```
docker pull debian
docker pull busybox
```

Comptez les commandes présentes dans `/bin` de chaque côté :

```
docker run --rm debian ls /bin | wc -l
docker run --rm busybox ls /bin | wc -l
```

- Combien de chaque côté ?

Regardez maintenant à quoi ressemblent ces commandes dans BusyBox :

```
docker run --rm busybox ls -l /bin | head -5
```

- Vers quoi pointent ces fichiers ?
- Combien de programmes réels se cachent donc derrière toutes ces commandes ?

BusyBox, c'est un seul exécutable

Chaque commande est un lien symbolique vers le même binaire, `/bin/busybox`, qui regarde sous quel nom on l'a appelé pour décider quoi faire. C'est ce qui explique la taille de l'image.

Le prix à payer : ce sont des versions allégées, qui n'acceptent qu'une partie des options de leurs équivalents habituels. Vos réflexes de ligne de commande n'y fonctionneront pas tous.

Une image sans rien dedans

Les images `distroless` vont plus loin : ni shell, ni commandes du tout. Essayez d'y entrer, puis d'y lancer la commande la plus élémentaire qui soit :

```
docker run --rm -it gcr.io/distroless/static-debian12 sh
docker run --rm gcr.io/distroless/static-debian12 ls /
```

- Que répond Docker, les deux fois ? Le problème vient-il du conteneur ou de l'image ?

Si le téléchargement échoue

Cette image vient de `gcr.io`, pas de Docker Hub. Si votre réseau le bloque, passez cette sous-partie : retenez simplement qu'une image `distroless` ne contient **aucune** commande, donc qu'aucun `docker exec` n'y est possible.

Validation — vous savez dire combien de commandes offre chacune des deux images, et expliquer pourquoi BusyBox en propose autant pour si peu de place.

Partie 6 — Une image, plusieurs architectures

Votre machine a une architecture ; les images aussi. Voyons ce que Docker choisit pour vous, et ce qui se passe quand on le contredit.

Commencez par savoir où vous êtes :

```
docker image inspect --format '{{.Os}}/{{.Architecture}}' alpine
docker run --rm alpine uname -m
```

- Quelle architecture ? Correspond-elle à votre processeur ?

Demandez maintenant la liste de **toutes** les variantes publiées sous ce même nom (pour plus de simplicité, pensez à filtrer la sortie) :

```
docker buildx imagetools inspect alpine:latest
```

- Combien de plateformes distinctes sont listées ?
- Celle de votre machine y figure-t-elle ? Qui l'a choisie, et quand ?

Forcer l'autre architecture

Choisissez la plateforme qui n'est **pas** la vôtre : `linux/amd64` si vous êtes en `arm64`, `linux/arm64` sinon. Tirez la variante correspondante, et observez :

```
docker pull --platform linux/???64 alpine
docker image inspect --platform linux/???64 --format '{{.Architecture}}' alpine
docker run --rm alpine uname -m
```

⚠ Ce `pull` écrase votre tag local

`alpine:latest` désigne maintenant la variante étrangère. C'est voulu, et c'est un piège classique en vrai : un `pull --platform` mal placé dans un script et toute la machine travaille en émulation sans que personne ne le remarque. Vous remettrez les choses en ordre à la fin de la partie.

- `uname -m` a-t-il changé ? Le conteneur fonctionne-t-il malgré tout ?

Mesurez ce que l'émulation coûte. Comparez la même boucle, dans l'image étrangère puis dans la variante native :

```
time docker run --rm alpine sh -c 'i=0; while [ $i -lt 200000 ]; do i=$((i+1)); done'
docker pull --platform linux/???64 alpine
time docker run --rm alpine sh -c 'i=0; while [ $i -lt 200000 ]; do i=$((i+1)); done'
```

🔗 Adaptez les deux plateformes à votre machine

Dans ce bloc, la première mesure doit porter sur l'architecture **étrangère** et la seconde sur la **vôtre**. Si vous êtes sur un PC `amd64`, inversez : tirez `--platform linux/arm64` avant la première

mesure, puis `--platform linux/amd64` avant la seconde.

- Quel rapport entre les deux durées ?
- Le cours parlait d'« émulation » et disait « on traduit, et ça se paie ». Combien ça coûte, chez vous ?

⚠ Sur Linux natif, ça peut simplement refuser

Docker Desktop embarque l'émulateur nécessaire. Sur une installation Linux sans émulateur enregistré, le conteneur étranger échoue au démarrage avec `exec format error`. Ce n'est pas un échec de votre part : notez le message et passez à la suite.

Remettez votre architecture native en place avant de continuer :

```
docker pull alpine
docker run --rm alpine uname -m
```

Validation — `docker run --rm alpine uname -m` affiche à nouveau l'architecture de votre machine, et vous avez noté les deux durées de la boucle.

Partie 7 — Fabriquer une image avec `commit`

Vous allez produire votre première image. La méthode employée ici est celle qu'on déconseille, et vous allez comprendre pourquoi en la pratiquant.

L'idée : prendre un conteneur, le bricoler à la main, puis figer sa couche inscriptible en une nouvelle couche d'image.

Bricoler le conteneur

```
docker run -dit --name atelier alpine sh
docker exec atelier apk add --no-cache curl
docker exec atelier sh -c 'echo "outil maison v1" > /etc/mon-outil'
```

Vérifiez que `curl` est bien là, et regardez l'ampleur des dégâts dans la couche inscriptible :

```
docker exec atelier curl --version
docker diff atelier | wc -l
```

- Combien de lignes `docker diff` renvoie-t-il ? Quelle est la taille de la couche conteneur ?

Figier

Cherchez la commande qui transforme un conteneur en image, et ses options utiles :

```
docker commit --help
```

Repérez l'option qui attache un message à la nouvelle couche et celle qui renseigne un auteur. Puis fabriquez votre image, nommée `mon-alpine:1.0`, avec un message qui décrit ce que vous avez fait.

 **Nommez-la exactement** `mon-alpine:1.0`

Les parties 8 et 9 reprennent ce nom tel quel.

Vérifiez le résultat :

```
docker image ls mon-alpine
docker image history mon-alpine:1.0
```

Répondez :

1. Combien de couches votre image a-t-elle de plus qu' `alpine` ? Vérifiez avec `docker image inspect --format '{{len .RootFS.Layers}}'` sur les deux.
2. Quelle taille fait la couche que vous venez d'ajouter ? Et tout le reste de l'image, d'où vient-il ?
3. Votre message apparaît-il dans l'historique ?
4. La colonne `CREATED BY` de votre couche dit-elle comment `curl` a été installé ?

Vérifier que l'image est autonome

Détruisez le conteneur d'origine, puis lancez un conteneur neuf à partir de votre image :

```
docker rm -f atelier
docker run --rm mon-alpine:1.0 curl --version
docker run --rm mon-alpine:1.0 cat /etc/mon-outil
```

- Les deux commandes réussissent-elles ?

Pourquoi `commit` n'est pas la bonne méthode

Votre image fonctionne, et rien dans son historique ne dit comment elle a été produite. Ni la commande `apk add`, ni l'ordre des opérations, ni le contenu exact du fichier. Dans six mois, personne — vous compris — ne saura la reconstruire à l'identique, ni la corriger, ni la relire.

`commit` est utile pour **comprendre** ce qu'est une couche, et pour dépanner. Le chapitre sur le `Dockerfile` apporte la méthode reproductible.

Validation — `docker run --rm mon-alpine:1.0 curl --version` affiche une version de `curl`, et vous savez dire quelle part de votre image vous avez réellement fabriquée.

Partie 8 — Transporter une image sans registry

Votre image n'existe que sur votre machine. Pour la donner à quelqu'un, vous pourriez passer par un registry — mais il existe aussi une manière de la faire "manuellement".

save : l'image dans un fichier

Cherchez la commande qui écrit une image dans une archive, et l'option qui désigne le fichier de sortie :

```
docker image save --help
```

Produisez `mon-alpine.tar` à partir de `mon-alpine:1.0`, puis pesez-le :

```
ls -lh mon-alpine.tar
```

- La taille du fichier correspond-elle à celle annoncée par `docker image ls` ?

Détruire, puis restaurer

Supprimez l'image de votre machine et vérifiez qu'elle a bien disparu :

```
docker rmi mon-alpine:1.0
docker image ls mon-alpine
```

- Que répond Docker cette fois : « Untagged », « Deleted », ou les deux ? Pourquoi, par rapport à la Partie 4 ?

Restaurez-la depuis l'archive. Trouvez l'option d'entrée :

```
docker image load --help
```

Puis, une fois chargée :

```
docker image ls mon-alpine
docker image history mon-alpine:1.0
docker run --rm mon-alpine:1.0 cat /etc/mon-outil
```

Répondez :

1. L'image a-t-elle retrouvé son nom et son tag ?
2. L'historique des couches est-il identique à celui de la Partie 7 ?
3. Le fichier `/etc/mon-outil` est-il présent ?

 **save** / **load** **conserve tout**

Couches, tags, configuration, historique : l'archive contient l'image entière, telle quelle. C'est la façon de transporter / archiver une image manuellement.

Partie 9 — Ranger derrière soi

Ce TP a laissé des traces : des conteneurs qui tournent encore, les images que vous avez téléchargées, celle que vous avez fabriquée, et l'archive produite en Partie 8.

Faites le ménage vous-même. Vous avez déjà toutes les commandes nécessaires : celles du chapitre précédent pour les conteneurs, celles de ce TP pour les images.

- Arrêtez et supprimez les conteneurs lancés au fil du TP.
- Supprimez l'image `mon-alpine:1.0` et l'archive `mon-alpine.tar`.
- **Gardez** les images téléchargées depuis un registry : `nginx`, `alpine`, `debian` et les autres resserviront aux chapitres suivants, et vous éviterez de les retélécharger.

Vérifiez avant de supprimer

Une commande qui liste n'efface rien. Avant chaque suppression, affichez ce que vous vous apprêtez à retirer — c'est le réflexe qui vous évitera, un jour, d'emporter autre chose que prévu.

Validation — `docker ps -a` ne liste plus aucun conteneur de ce TP, `docker image ls` ne contient plus `mon-alpine`, et les images officielles que vous avez tirées sont toujours là.

Mémo des commandes

- **Lister, peser** : `docker image ls`, `docker system df`, `docker ps -s`
- **Inspecter** : `docker image inspect (--format)`, `docker image history`, `docker diff`
- **Récupérer** : `docker pull (--platform, @sha256:...)`, `docker buildx imagetools inspect`
- **Nommer** : `docker tag`, `docker rmi`
- **Fabriquer** : `docker commit (-m, -a)`
- **Transporter** : `docker image save` / `docker image load`

Pour finir

Vous avez vérifié à la main presque tout ce que le cours affirmait : qu'une image est une pile de couches en lecture seule, qu'un tag se déplace mais qu'un digest non, que supprimer n'allège pas, que le confort de débogage se paie en mégaoctets, et qu'une image sortie de son architecture tourne quand même — plus lentement.

Vous avez aussi fabriqué une image, et constaté son défaut : elle marche, mais rien ne dit comment elle a été faite. C'est précisément le problème que le `Dockerfile` vient résoudre, quelques chapitres plus loin.