



docker-playful.webp

Prise en main des conteneurs

Prérequis

Votre environnement doit être prêt : voir [Activity - Préparer son environnement Linux](#). **Docker Desktop doit être lancé** — ce n'est pas un service qui démarre tout seul. Si `docker version` ne répond pas, commencez par là.

Dans quel terminal travailler

Un terminal où la commande `docker` répond, et qui soit un shell Linux ou macOS :

- **Windows** : votre Ubuntu WSL, pas PowerShell. Le TP utilise `time` et la séquence `Ctrl-P Ctrl-Q`, que PowerShell intercepte ou ne connaît pas.
- **macOS** : le Terminal suffit, pas besoin de la VM Lima du chapitre précédent.
- **Linux** : votre shell habituel.

Ce qu'on cherche à obtenir

Au chapitre précédent, vous avez fabriqué un conteneur à la main. Ici, c'est Docker qui le fabrique, et vous apprenez à lui parler.

À la fin de ce TP, vous saurez lancer un conteneur, le mettre en pause et le relancer, y ouvrir un shell, lire ses journaux, en sortir sans l'arrêter, y copier un fichier, l'arrêter proprement ou brutalement, et faire le ménage derrière vous.

On teste beaucoup, on explique peu

Ce chapitre est un tour d'horizon : l'objectif est de **voir l'étendue** de ce que Docker sait faire. Plusieurs commandes vont produire des résultats que vous ne saurez pas encore expliquer — c'est normal et c'est voulu. Les chapitres suivants reprennent chaque sujet (images, données, réseau) en profondeur. Notez vos questions au fil du TP, elles trouveront leur réponse.

La documentation est dans la commande

Chaque sous-commande Docker se documente elle-même :

```
docker --help
docker run --help
```

`docker help run` est équivalent. En revanche, n'utilisez pas `man docker` : selon votre installation, les pages de manuel ne sont pas installées, alors que `--help` est toujours disponible.

La sortie de `docker run --help` est longue. Prenez le réflexe de la filtrer :

```
docker run --help | grep -i detach
```

Partie 1 — À qui parlez-vous ?

La commande `docker` ne crée rien elle-même. C'est un **client** : elle transmet vos demandes, via une API, à un **démon** (`dockerd`) qui fait le travail. Vérifions-le.

Demandez les versions des deux côtés :

```
docker version
```

Puis les informations détaillées du démon :

```
docker info
```

La sortie de `docker info` est longue ; cherchez les lignes `Server Version`, `Operating System` et `Kernel Version`.

Répondez :

1. La sortie de `docker version` comporte deux blocs. Comment s'appellent-ils ?
2. Quel noyau `docker info` annonce-t-il ?
3. Si vous êtes sur macOS ou Windows : ce noyau est-il celui de votre machine ? Où tourne-t-il, alors ?
4. Repérez quelques briques essentiels à Docker que l'on a évoqué au chapitre précédent.

Vous ne lancez jamais un conteneur vous-même

Vous demandez au démon de le faire. C'est pour ça que Docker Desktop doit tourner avant toute commande, et c'est aussi pour ça que, sur macOS et Windows, le noyau annoncé est celui d'une VM Linux et non celui de votre machine — exactement le mécanisme vu au chapitre précédent.

Validation — vous savez dire quelle version a votre client, quelle version a votre serveur, et sur quel noyau le serveur s'appuie.

Partie 2 — Premier conteneur

On lance le conteneur le plus simple qui existe, et surtout on lit ce qu'il raconte.

Vous avez déjà exécuté `hello-world` en préparant votre environnement : son image est donc déjà sur votre machine, et vous rateriez la moitié de la démonstration. Supprimez-la d'abord :

```
docker rmi -f hello-world
```

Maintenant, lancez :

```
docker run hello-world
```

Lisez la sortie de haut en bas, sans la survoler. Les premières lignes décrivent ce que Docker a fait avant même de démarrer quoi que ce soit ; le paragraphe en anglais qui suit récapitule les quatre étapes.

Répondez :

1. Quelle est la toute première ligne affichée ? Que dit-elle sur l'endroit où Docker a cherché l'image en premier ?
2. D'où l'image a-t-elle été téléchargée ?
3. Combien d'étapes Docker décrit-il entre votre commande et le message affiché ?

Vérifiez que l'image est maintenant stockée localement :

```
docker images
```

Relancez exactement la même commande qu'avant :

```
docker run hello-world
```

- La sortie est-elle identique à la première fois ? Qu'est-ce qui a disparu, et pourquoi ?

Enfin, regardez ce qui reste derrière :

```
docker ps
docker ps -a
```

- Pourquoi la première commande n'affiche-t-elle rien, alors que la seconde affiche deux lignes ? (ou plus si vous utilisez déjà Docker pour d'autres applications, pour plus de simplicité, vous pouvez filtrer

votre sortie avec `| grep hello`)

Un conteneur arrêté existe toujours

`hello-world` affiche son message puis se termine. Le conteneur n'a pas disparu pour autant : il reste sur le disque, à l'état `Exited`, jusqu'à ce que vous le supprimiez. On y revient en Partie 3 et on fait le ménage en Partie 8.

Validation — `docker images` liste `hello-world`, et `docker ps -a` montre au moins deux conteneurs `Exited`.

Partie 3 — Le cycle de vie d'un conteneur

Un conteneur passe par des états, et des commandes permettent de le faire passer d'un état à l'autre. On va les parcourir sur un conteneur qui, contrairement à `hello-world`, ne se termine pas immédiatement.

Vous aurez besoin de deux options de `docker run` :

```
docker run --help | grep -iE "detach|name"
```

Repérez l'option qui lance le conteneur en arrière-plan, et celle qui lui donne un nom lisible plutôt qu'un identifiant aléatoire.

Lancez un conteneur Alpine qui ne fait qu'attendre, en arrière-plan, nommé `veilleuse` :

```
docker run -d --name veilleuse alpine sleep 3600
```

La commande affiche normalement un long identifiant et vous rend la main. Vérifiez son état :

```
docker ps
```

Maintenant, parcourez le cycle. Après **chacune** de ces commandes, relancez `docker ps -a` et notez la colonne `STATUS` :

```
docker stop veilleuse
docker start veilleuse
docker restart veilleuse
```

- Quel `STATUS` après `stop` ? Après `start` ?
- Utiliser `start` sur un conteneur déjà lancé a-t-il un effet ? Et `restart` ?

Un état que vous n'avez pas encore vu

Créez un conteneur sans le démarrer :

```
docker create --name jamais-lance alpine sleep 60
docker ps -a
```

- Quel est son `STATUS` ? En quoi diffère-t-il de celui d'un conteneur arrêté ?

Démarrez-le, puis vérifiez :

```
docker start jamais-lance
docker ps
```

🔗 Trois états à retenir

`created` (préparé, jamais exécuté), `running` (en cours), `exited` (le processus principal s'est terminé). `docker run` n'est rien d'autre qu'un `docker create` suivi d'un `docker start`.

Supprimer un conteneur

Essayez de supprimer `veilleuse`, qui tourne encore :

```
docker rm veilleuse
```

- Que répond Docker ? Que vous propose-t-il de faire ?

Suivez le conseil du message d'erreur, puis supprimez les conteneurs `veilleuse` et `jamais-lance`.

Ne rien laisser derrière soi

Il existe une option de `docker run` qui supprime le conteneur dès qu'il se termine. Trouvez-la :

```
docker run --help | grep -i remove
```

Lancez un conteneur jetable qui affiche un message et s'en va, avec cette option :

```
docker run --?? alpine echo "je ne laisse pas de trace"
docker ps -a
```

- Le conteneur apparaît-il dans la liste ? Comparez avec le comportement de la Partie 2.

Validation — `docker ps -a` ne contient plus ni `veilleuse`, ni `jamais-lance`, ni le conteneur jetable. Seuls les deux `hello-world` de la Partie 2 subsistent.

Partie 4 — Les deux façons de lancer un conteneur

Jusqu'ici, vous avez lancé des conteneurs qui parlent et s'en vont. On passe aux deux modes réellement utiles : le mode interactif, pour travailler dedans, et le mode détaché, pour faire tourner un service.

Mode interactif : un shell dans le conteneur

Deux options sont nécessaires, et aucune des deux ne suffit seule :

```
docker run --help | grep -iE "interactive|tty"
```

Lancez un shell dans Alpine :

```
docker run -?? --rm alpine sh
```

L'invite change : vous êtes **dans** le conteneur. Regardez où vous êtes atterri :

```
cat /etc/os-release
ls /
whoami
```

- Sur quelle distribution êtes-vous ? Est-ce la même que votre machine ?
- Sous quelle identité travaillez-vous ?

Sortez avec `exit` .

⚠ Ce que fait `exit`

Dans ce conteneur, le shell **est** le processus principal. En sortant, vous le terminez, donc vous arrêtez le conteneur. Ce n'est pas un détail : c'est la source de confusion numéro un quand on débute. La section suivante montre comment sortir *sans* arrêter le conteneur.

Se détacher, se rattacher

Relancez un shell interactif, mais cette fois sans `--rm` et avec un nom :

```
docker run -it --name bavard alpine sh
```

Vous êtes dedans. Au lieu de taper `exit` , pressez **Ctrl-P puis Ctrl-Q** . Vous revenez à votre terminal.

```
docker ps
```

- Le conteneur `bavard` tourne-t-il encore ? En quoi est-ce différent de ce qui se passe avec `exit` ?

Revenez dedans :

```
docker attach bavard
```

Vous retrouvez votre shell, là où vous l'aviez laissé. Cette fois, sortez avec `exit` , et vérifiez :

```
docker ps -a
docker rm bavard
```

⚠ La séquence de détachement a deux conditions

`Ctrl-P Ctrl-Q` ne fonctionne que si le conteneur a été lancé avec un terminal attaché (`-t`) et en mode interactif (`-i`). Sur un conteneur lancé autrement, `docker attach` vous branche sur sa sortie sans vous donner d'invite, et seul `Ctrl-C` vous en sortira — au risque d'arrêter le conteneur.

En pratique, pour entrer dans un conteneur déjà lancé, on utilise `docker exec` (Partie 6) et pas `docker attach`.

Mode détaché : un service qui tourne

Lancez un serveur web Nginx en arrière-plan, et rendez-le joignable depuis votre navigateur :

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

Vérifiez :

```
docker ps
```

- Que contient la colonne `PORTS` ?

Ouvrez maintenant <http://localhost:8080> dans votre navigateur. Vous devez voir la page d'accueil de Nginx.

⚠ Si le port 8080 est déjà occupé

Docker refuse de démarrer avec un message du type « *port is already allocated* ». Choisissez un autre port côté gauche — `-p 8081:80`, `-p 9000:80` — et adaptez l'URL du navigateur. Seul le nombre de gauche change ; celui de droite doit rester `80`.

`-p 8080:80` : admis pour l'instant

Cette option relie un port de votre machine à un port du conteneur. Pourquoi c'est nécessaire, ce que ça implique côté hôte et ce qui se passe quand on ne la met pas : tout ça est le sujet du chapitre sur le réseau. Ici, on l'utilise sans chercher à l'expliquer.

Validation — `docker ps` montre `web` en `Up`, et votre navigateur affiche « *Welcome to nginx!* ».

Partie 5 — Observer un conteneur qui tourne

`web` tourne, mais vous n'en voyez rien. Docker fournit quatre commandes pour regarder à l'intérieur sans y entrer.

Les journaux

Docker capture tout ce que le processus principal écrit sur sa sortie standard.

```
docker logs web
```

Deux options vous seront utiles au quotidien ; trouvez-les :

```
docker logs --help | grep -iE "follow|tail"
```

Mettez-vous en écoute continue avec l'option de suivi, puis **rechargez la page** dans votre navigateur et regardez votre terminal :

```
docker logs -f web
```

- Que voyez-vous apparaître à chaque rechargement ?

Quittez l'écoute avec `Ctrl-C`.

 **Ctrl-C ici n'arrête pas le conteneur**

Vous n'interrompez que la commande `docker logs`, qui tourne sur votre machine. Vérifiez-le avec `docker ps : web` est toujours `Up`.

La fiche technique

```
docker inspect web
```

C'est un gros bloc JSON — plusieurs centaines de lignes. Personne ne le lit en entier : on en extrait un champ précis. Cherchez l'option qui le permet :

```
docker inspect --help
```

Elle attend un chemin dans le JSON, entre doubles accolades. Essayez :

```
docker inspect --format '{{.State.Status}}' web
docker inspect --format '{{.State.Pid}}' web
docker inspect --format '{{NetworkSettings.Networks.bridge.IPAddress}}' web
```

- Quelle adresse IP le conteneur possède-t-il ?
- Essayez d'y accéder depuis votre navigateur, à la place de `localhost:8080`. Que se passe-t-il ?

 **Cette adresse ne vous sert à rien pour l'instant**

Elle appartient à un réseau interne géré par le démon Docker. Depuis macOS ou Windows, elle n'existe même pas sur votre machine mais dans la VM Linux. C'est précisément ce que le chapitre sur le réseau démêlera — et c'est la raison d'être du `-p` de la Partie 4.

La consommation

```
docker stats
```

L'affichage se rafraîchit en continu. Quittez avec `Ctrl-C`, puis cherchez l'option qui donne un instantané unique :

```
docker stats --help
```

Les processus

```
docker top web
```

- Combien de processus tournent dans le conteneur ? Quel est le premier de la liste ?
- Si vous êtes sur Linux, cherchez ces mêmes processus sur votre machine avec `ps -ef | grep nginx`. Les trouvez-vous ? Et si vous êtes sur macOS ou Windows ?

Validation — vous savez obtenir, pour n'importe quel conteneur : ses journaux, son état via `--format`, sa consommation mémoire et la liste de ses processus.

Partie 6 — Intervenir dans un conteneur

Observer ne suffit pas toujours : il faut parfois ouvrir un shell dans un conteneur qui tourne, ou y déposer un fichier.

Ouvrir un shell : `exec`

```
docker exec -it web sh
```

Vous êtes dans `web`. Regardez d'abord sur quelle distribution est construite l'image Nginx :

```
cat /etc/os-release
```

- Est-ce la même qu'Alpine en Partie 4 ? Qu'en déduisez-vous sur les images Docker ?

Remplacez la page d'accueil du serveur. Le fichier servi par défaut est `/usr/share/nginx/html/index.html` :

```
echo "<h1>Bonjour depuis le conteneur</h1>" > /usr/share/nginx/html/index.html
exit
```

Rechargez <http://localhost:8080>.

- La page a-t-elle changé ?
- Vous venez de taper `exit`. Le conteneur est-il arrêté ? Vérifiez avec `docker ps`.

`exec` et `attach` ne font pas la même chose

`attach` vous branche sur le processus principal du conteneur — le quitter peut l'arrêter. `exec` lance un nouveau processus à côté : en sortir ne touche pas au processus principal. C'est pour ça que `docker exec -it <conteneur> sh` est la façon normale d'entrer dans un conteneur en cours d'exécution.

Déplacer des fichiers : `cp`

`docker cp` fonctionne dans les deux sens, avec la syntaxe `conteneur:chemin` d'un côté.

Récupérez le fichier de configuration de Nginx sur votre machine :

```
docker cp web:/etc/nginx/nginx.conf .
ls -l nginx.conf
```

Maintenant l'inverse. Fabriquez une page sur votre machine, envoyez-la dans le conteneur :

```
echo "<h1>Page envoyée depuis l-hôte</h1>" > index.html
docker cp index.html web:/usr/share/nginx/html/index.html
```

Rechargez la page dans le navigateur.

Validation — le navigateur affiche votre dernière page, et `nginx.conf` est présent dans votre répertoire courant.

⚠ Tout ça est éphémère

Les modifications que vous venez de faire vivent dans ce conteneur et nulle part ailleurs. Supprimez-le, relancez `nginx`, et la page d'accueil d'origine revient. C'est le sujet du chapitre sur les données et la persistance.

Partie 7 — Retrouver le conteneur fabriqué à la main

Au chapitre précédent, vous avez empilé `chroot` et des namespaces à la main, puis constaté ce que chaque brique changeait. Refaisons exactement les mêmes observations, mais dans un conteneur Docker.

Rappelez-vous d'abord ce que votre machine affiche. Dans votre terminal :

```
hostname
id
ps -e | wc -l
```

✍ Sur macOS

`ps -e | wc -l` fonctionne, mais vous comptez les processus de macOS, pas ceux d'un Linux. La comparaison reste parlante : ce qui compte est l'ordre de grandeur, quelques dizaines ou centaines contre une poignée.

Maintenant, les mêmes commandes dans un conteneur :

```
docker run -it --rm alpine sh
```

À l'intérieur :

```
hostname
id
ps -e
ip link
ls /
```

 Si `ip link` n'affiche rien d'exploitable

Listez les interfaces autrement : `ls /sys/class/net`.

Répondez, en comparant point par point avec le chapitre précédent :

1. `hostname` affiche une suite de caractères inhabituelle. À quoi correspond-elle ? Comparez avec la sortie de `docker ps`.
2. Combien de processus voyez-vous ? Quel est le PID du shell ? Quel namespace du chapitre précédent reconnaissez-vous ?
3. Combien d'interfaces réseau, et lesquelles ? Qu'est-ce qui a été créé pour ce conteneur ?
4. Le contenu de `/` est-il celui de votre machine ? Quelle brique du chapitre précédent est à l'œuvre ?
5. `id` affiche `uid=0(root)`. Êtes-vous root sur votre machine pour autant ?

Sortez avec `exit`.

Docker n'invente rien

Un `docker run` fait, en une commande, ce que vous avez empilé à la main : nouveau système de fichiers, nouveau nom de machine, nouvelle arborescence de processus, nouvel espace réseau, et un root aux pouvoirs réduits. Les commandes de ce TP ne sont qu'une interface confortable au-dessus des mêmes mécanismes du noyau.

Validation — vous pouvez associer chacune des cinq observations ci-dessus à la brique correspondante du chapitre précédent.

Partie 8 — Arrêter : proprement ou brutalement

`docker stop` et `docker kill` arrêtent tous les deux un conteneur, mais pas de la même façon. La différence se mesure.

`docker stop` demande d'abord poliment au processus principal de se terminer, en lui envoyant le signal `SIGTERM`. S'il n'a pas obtempéré au bout d'un délai, Docker le tue sans sommation avec `SIGKILL`, auquel aucun processus ne peut résister. `docker kill` envoie `SIGKILL` directement.

Lancez un conteneur qui refuse explicitement d'obéir à `SIGTERM` :

```
docker run -d --name tetu alpine sh -c 'trap "" TERM; sleep 300'
```

Arrêtez-le en mesurant le temps que ça prend :

```
time docker stop tetu
```

- Combien de temps la commande a-t-elle mis ?

Comparez avec `web`, dont le processus principal (Nginx) sait réagir à `SIGTERM` :

```
time docker stop web
```

- Et cette fois, combien de temps ?

Regardez comment chacun des deux s'est terminé :

```
docker inspect --format '{{.Name}} {{.State.ExitCode}}' tetu web
```

- Les deux codes de sortie sont-ils identiques ? Lequel correspond à une terminaison normale, lequel à un processus tué ?

🔗 137, et pourquoi PID 1 est un cas à part

Un code 137 vaut $128 + 9$: le processus a été tué par le signal 9, `SIGKILL`. C'est la signature d'un conteneur que `docker stop` a dû achever de force.

Attention à une subtilité que vous rencontrerez vite : dans un namespace PID, le processus numéro 1 **ignore** les signaux pour lesquels il n'a pas installé de gestionnaire. Un conteneur dont le processus principal ne gère pas `SIGTERM` mettra donc systématiquement le délai complet à s'arrêter, même s'il n'a aucune raison de résister. C'est le cas de la plupart des conteneurs lancés avec un `sh -c "..."` improvisé.

Le délai avant le coup de grâce est réglable. Trouvez l'option :

```
docker stop --help
```

Relancez `tetu` et arrêtez-le avec un délai de deux secondes :

```
docker start tetu
time docker stop -t 2 tetu
```

Enfin, comparez avec l'arrêt brutal :

```
docker start tetu
time docker kill tetu
docker inspect --format '{{.State.ExitCode}}' tetu
```

- Quelle différence de durée avec `docker stop` ?
- Dans quel cas est-il raisonnable d'utiliser `kill` plutôt que `stop` ?

Validation — vous savez expliquer pourquoi `docker stop` est parfois instantané et parfois lent, et ce que signifie un code de sortie 137.

Partie 9 — Faire le ménage



queen-i-want-to-break-free.webp

Vous avez accumulé des conteneurs arrêtés tout au long du TP. Ils occupent du disque pour rien.

Dressez l'inventaire :

```
docker ps -a
```

- Combien de conteneurs reste-t-il ? Dans quels états ?

Supprimez-en un nommé `tetu`, pour vous assurer que la commande fonctionne encore :

```
docker rm tetu
```

Puis, pour le reste, il existe une commande qui supprime d'un coup tous les conteneurs arrêtés. Cherchez-la :

```
docker container --help
```

Lancez-la. Elle demande confirmation avant d'agir : lisez le message, il vous dit exactement ce qui va être supprimé.

```
docker container prune  
docker ps -a
```

⚠ `prune` ne demande pas deux fois

Les commandes `prune` suppriment sans retour possible. Ici, le risque est nul — ce sont vos conteneurs de TP. En usage réel, lisez toujours le message de confirmation avant de taper `y` : `docker container prune` emporte **tous** les conteneurs arrêtés de la machine, y compris ceux que vous comptiez redémarrer demain.

Vérifiez pour finir ce que le ménage n'a *pas* touché :

```
docker images
```

- Les images sont-elles encore là ? Supprimer un conteneur supprime-t-il l'image dont il était issu ?

Les images ont leur propre cycle de vie

Elles survivent aux conteneurs, s'empilent et finissent par occuper beaucoup de place. Comment elles sont construites, stockées et nettoyées : c'est tout le chapitre suivant.

Validation — `docker ps -a` ne liste plus aucun conteneur, et `docker images` en liste encore au moins deux.

Mémo des commandes

Ce que vous avez utilisé, regroupé par usage. Gardez cette liste sous les yeux pour les chapitres suivants.

- **Se renseigner** : `docker version`, `docker info`, `docker <commande> --help`
- **Lancer** : `docker run (-d, -it, --rm, --name, -p)`, `docker create`
- **Lister** : `docker ps`, `docker ps -a`, `docker images`
- **Cycle de vie** : `docker start`, `docker stop (-t)`, `docker restart`, `docker kill`
- **Observer** : `docker logs (-f, --tail)`, `docker inspect (--format)`, `docker stats`, `docker top`, `docker events`
- **Intervenir** : `docker exec -it`, `docker attach`, `docker cp`
- **Nettoyer** : `docker rm`, `docker rmi`, `docker container prune`

Pour finir

Vous venez de faire le tour des commandes que vous utiliserez tous les jours. Beaucoup de questions restent ouvertes : d'où viennent les images et comment elles sont faites, pourquoi vos modifications disparaissent avec le conteneur, ce que `-p 8080:80` fabrique réellement. Chacune est un chapitre à venir.

Pour l'instant, retenez surtout deux choses : la commande `docker` ne fait que transmettre des demandes à un démon, et ce que ce démon fabrique, vous l'avez déjà construit à la main au chapitre précédent.