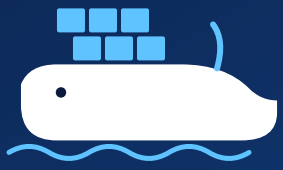
GUIDE D'APPRENTISSAGE



Docker
pour Débutants

Le guide complet pour comprendre, installer
et utiliser Docker — sans aucun prérequis

Niveau : Débutant · Aucune connaissance préalable requise

Édition 2026 · Docker Engine 29

Table des matières

Avant de commencer

> *Ce guide part de zéro. Vous n'avez besoin d'aucune expérience préalable de Docker, ni d'être développeur·euse confirmé·e. Une aisance de base avec un terminal (la « ligne de commande ») suffit largement, et nous reverrons l'essentiel au fur et à mesure.*

Comment ce guide est organisé

Le cours suit une progression volontairement lente et logique. Chaque module s'appuie sur le précédent : nous commençons par **comprendre** ce qu'est Docker et le problème qu'il résout, puis nous **installons** l'outil, ensuite nous **manipulons** des conteneurs existants, et enfin nous **créons** les nôtres avant d'**orchestrer** plusieurs services ensemble. Le tout se termine par un projet complet que vous réaliserez de A à Z.

Les encadrés que vous rencontrerez

À RETENIR

L'essentiel à mémoriser. Si vous ne deviez retenir qu'une chose d'une section, ce serait ceci.

ASTUCE

Un conseil pratique pour gagner du temps ou éviter une erreur courante.

ATTENTION

Un point délicat ou un piège classique de débutant à éviter.

ANALOGIE

Une comparaison avec le monde réel pour ancrer une idée abstraite.

ASTUCE — ENTRAÎNEZ-VOUS EN TAPANT VRAIMENT

On n'apprend pas Docker en lisant : on l'apprend en tapant les commandes. Gardez un terminal ouvert pendant votre lecture et reproduisez chaque exemple. Les erreurs font partie de l'apprentissage — elles sont sans danger ici.

MODULE 1

Qu'est-ce que Docker ?

> *Avant d'apprendre comment utiliser Docker, il faut comprendre pourquoi il existe. Docker répond à un problème très concret que rencontre toute personne qui crée ou installe des logiciels.*

Le problème : « pourtant, ça marche sur ma machine ! »

Imaginez que vous développez une application. Sur votre ordinateur, tout fonctionne parfaitement. Vous l'envoyez à un collègue… et chez lui, rien ne marche. Pourquoi ? Parce que son ordinateur n'a pas la même version du langage, pas les mêmes bibliothèques installées, pas le même système d'exploitation, pas les mêmes réglages.

Un logiciel ne tourne jamais seul : il dépend d'un **environnement** entier (le système, des bibliothèques, des variables de configuration, des versions précises). Recréer cet environnement à l'identique sur chaque machine — celle des collègues, le serveur de test, le serveur de production — est long, fastidieux et source d'innombrables bugs.

ANALOGIE — LE TRANSPORT MARITIME

Avant les années 1950, charger un bateau était un cauchemar : chaque marchandise (sacs, tonneaux, caisses de tailles différentes) demandait une manutention spécifique. Puis on a inventé le **conteneur maritime standardisé** : une boîte de taille fixe que l'on peut empiler, charger sur n'importe quel bateau, camion ou train, sans se soucier de ce qu'il y a dedans. Docker fait exactement la même chose pour les logiciels : il met votre application *et tout son environnement* dans une « boîte » standard qui tourne de la même façon partout. Ce n'est pas un hasard si le logo de Docker est une baleine qui transporte des conteneurs !

La solution apportée par Docker

Docker permet d'emballer une application avec **tout ce dont elle a besoin pour fonctionner** — le code, le langage, les bibliothèques, les fichiers de configuration — dans une unité unique et portable appelée **conteneur**. Ce conteneur tournera de manière identique sur votre machine, sur celle de vos collègues et sur un serveur, quel que soit le système installé en dessous.

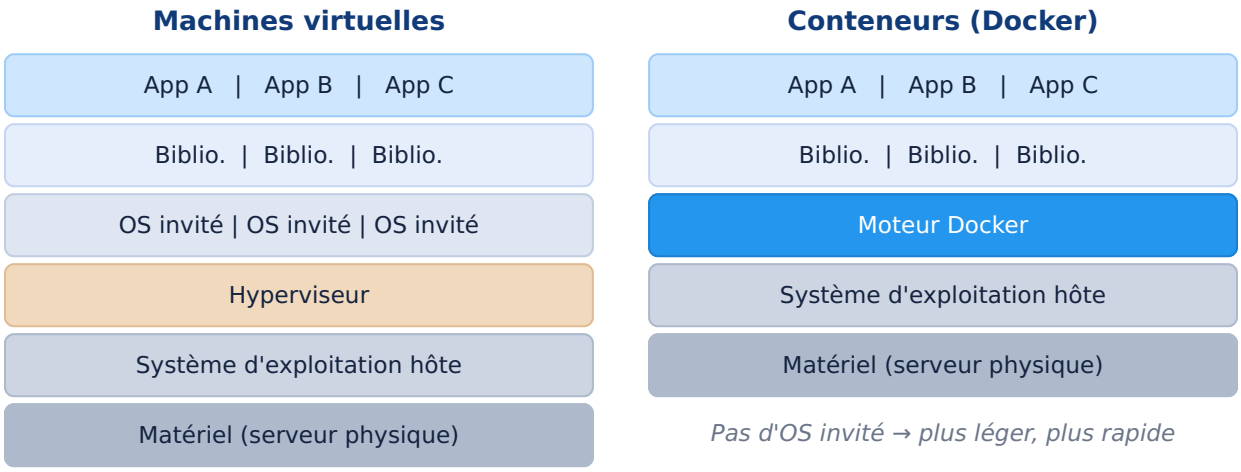
Ce que Docker vous apporte concrètement

- **Portabilité** — « si ça marche chez moi, ça marchera partout », car l'environnement voyage *avec* l'application.
- **Reproductibilité** — vous pouvez recréer exactement le même environnement autant de fois que vous voulez, en une commande.
- **Isolation** — chaque conteneur est cloisonné : deux applications avec des besoins contradictoires (par ex. deux versions différentes du même langage) cohabitent sans se gêner.
- **Rapidité** — démarrer un conteneur prend quelques secondes, là où installer manuellement un environnement prend des heures.
- **Légèreté** — les conteneurs consomment beaucoup moins de ressources que les machines virtuelles classiques (nous allons voir pourquoi).

Conteneurs et machines virtuelles : quelle différence ?

C'est *la* question clé pour comprendre Docker. Avant Docker, pour isoler des applications, on utilisait des **machines virtuelles (VM)**. Une VM simule un ordinateur complet, avec son propre système d'exploitation, à l'intérieur de votre ordinateur. C'est puissant, mais lourd : chaque VM embarque un système d'exploitation entier (plusieurs gigaoctets) et met des minutes à démarrer.

Un conteneur, lui, **partage le système d'exploitation** de la machine hôte. Il n'embarque que l'application et ses dépendances directes, pas un système complet. Résultat : il est minuscule, démarre en quelques secondes, et on peut en faire tourner des dizaines là où on ne pourrait lancer que quelques VM.



Une VM empile un système d'exploitation complet par application. Un conteneur partage celui de l'hôte : il ne reste que l'application et ses bibliothèques.

Critère	Machine virtuelle	Conteneur Docker
Taille	Plusieurs Go (OS complet)	Quelques Mo à quelques centaines de Mo
Démarrage	Dizaines de secondes à minutes	Quelques secondes, voire moins
Isolation	Très forte (OS séparé)	Forte (processus isolés, OS partagé)
Consommation	Élevée	Faible
Densité	Quelques VM par machine	Des dizaines de conteneurs par machine

À RETENIR

Docker emballe une application **avec son environnement** dans un **conteneur** léger et portable, qui tourne de façon identique partout. Contrairement à une machine virtuelle, un conteneur **partage le système d'exploitation** de l'hôte : il est donc beaucoup plus léger et démarre en quelques secondes.

ATTENTION — VM ET CONTENEURS NE SONT PAS ENNEMIS

On les compare souvent, mais ils sont complémentaires. En pratique, on fait très souvent tourner des conteneurs Docker… *à l'intérieur* de machines virtuelles, dans le cloud. Docker ne remplace pas tout : il résout un problème précis, celui de la portabilité et de l'isolation des applications.

MODULE 2

Les concepts fondamentaux

> *Docker repose sur un petit nombre de notions qui reviennent en permanence. Les maîtriser maintenant rendra tout le reste du guide limpide. Prenez le temps de bien assimiler ce vocabulaire.*

Les six mots à connaître

1. L'image

Une **image** est un modèle figé, en lecture seule, qui contient tout le nécessaire pour faire tourner une application : le système de fichiers, le code, les bibliothèques, la configuration. C'est un *plan*, une *recette*. Une image ne « tourne » pas ; elle sert de base à la création de conteneurs.

ANALOGIE

Une image, c'est comme le **moule à gâteau** (ou la recette). Le conteneur, c'est le **gâteau** que vous en sortez. Avec un seul moule, vous pouvez faire autant de gâteaux identiques que vous voulez.

2. Le conteneur

Un **conteneur** est une *instance en cours d'exécution* d'une image. C'est l'image « mise en marche ». Vous pouvez démarrer, arrêter, supprimer un conteneur, et en lancer plusieurs à partir de la même image. Le conteneur est isolé du reste du système et possède sa propre vie.

3. Le Dockerfile

Un **Dockerfile** est un simple fichier texte contenant la liste des instructions pour *construire* une image, étape par étape. C'est la recette écrite : « pars de telle base, copie ces fichiers, installe ces dépendances, lance cette commande ». Nous y consacrerons un module entier.

4. Le registre (registry) et Docker Hub

Un **registre** est un entrepôt en ligne où l'on stocke et partage des images. Le plus connu est **Docker Hub** : une immense bibliothèque publique où vous trouvez des images officielles

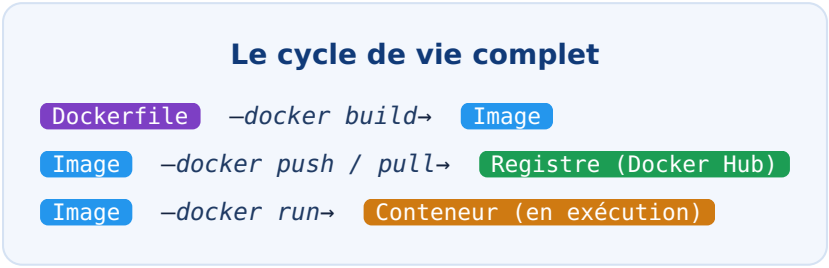
toutes prêtes (Python, Node.js, PostgreSQL, Nginx, etc.). On y « tire » (`pull`) des images, et on peut y « pousser » (`push`) les siennes.

5. Le volume

Un conteneur est **éphémère** : quand on le supprime, tout ce qu'il contenait disparaît. Pour *conserver* des données (une base de données, des fichiers téléversés…), on utilise un **volume** : un espace de stockage géré par Docker qui survit à la suppression du conteneur. Module 7.

6. Le réseau

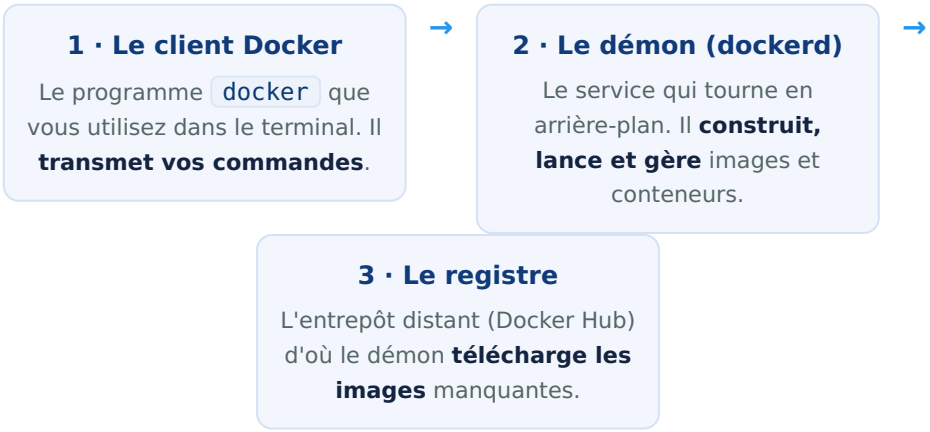
Les **réseaux** Docker permettent aux conteneurs de communiquer entre eux et avec l'extérieur. Par exemple, faire dialoguer un conteneur « application web » avec un conteneur « base de données ». Module 8.



On écrit un Dockerfile → on construit une image → on la partage via un registre → on la lance sous forme de conteneur.

L'architecture de Docker : client et démon

Quand vous tapez une commande comme `docker run`, deux acteurs entrent en jeu. Comprendre cette séparation évite bien des confusions plus tard.



Le client envoie vos ordres au démon, qui fait le vrai travail et va chercher les images dans un registre si besoin.

Le **client** (la commande `docker`) et le **démon** (`dockerd`) communiquent via une API. Sur votre machine, ils tournent généralement ensemble, mais ils pourraient être sur deux machines différentes. C'est le démon qui exécute réellement tout le travail : c'est lui le « cerveau » de Docker.

À RETENIR

Image = modèle figé (la recette). **Conteneur** = image en cours d'exécution (le plat). **Dockerfile** = recette écrite pour fabriquer une image. **Registre / Docker Hub** = bibliothèque d'images en ligne. **Volume** = stockage qui survit au conteneur. Le **client** donne les ordres, le **démon** les exécute.

MODULE 3

Installer Docker

> *Pour la plupart des débutants, l'installation se résume à installer **Docker Desktop**, une application avec interface graphique disponible sur Windows, macOS et Linux. Elle inclut tout le nécessaire : le moteur Docker, la ligne de commande et Docker Compose.*

Sur Windows

1. Rendez-vous sur le site officiel **docs.docker.com** et téléchargez **Docker Desktop for Windows**.
2. Docker s'appuie sur **WSL 2** (le sous-système Linux de Windows). L'installateur vous guide pour l'activer si nécessaire ; il faut généralement redémarrer une fois.
3. Lancez l'installateur, suivez les étapes, puis démarrez Docker Desktop. Une icône de baleine apparaît dans la barre des tâches une fois Docker prêt.

Sur macOS

1. Téléchargez **Docker Desktop for Mac** en choisissant la bonne puce : **Apple Silicon** (M1/M2/M3/M4…) ou **Intel**.
2. Ouvrez le fichier `.dmg` et glissez Docker dans le dossier Applications.
3. Lancez Docker depuis les Applications ; attendez que l'icône de baleine indique que le moteur est démarré.

Sur Linux

Sur Linux, vous avez deux options. Docker Desktop existe, mais beaucoup préfèrent installer directement **Docker Engine** (le moteur seul, en ligne de commande). La méthode recommandée est le script officiel ou les dépôts de paquets de votre distribution :

terminal — installation rapide (à des fins de test)

```
# Script d'installation officiel (Ubuntu, Debian, etc.)
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh

# Permettre d'utiliser docker sans « sudo » (puis se reconnecter)
sudo usermod -aG docker $USER
```

ATTENTION — LICENCE DOCKER DESKTOP

Docker Desktop est **gratuit** pour un usage personnel, l'éducation et les petites entreprises. En revanche, son usage commercial dans les **grandes entreprises** (plus de 250 employés *ou* plus de 10 millions de dollars de chiffre d'affaires annuel) nécessite un abonnement payant. Le moteur Docker Engine seul, sous licence Apache 2.0, reste libre.

Vérifier que tout fonctionne

Une fois l'installation terminée, ouvrez un terminal et tapez :

terminal

```
docker --version
# Affiche par exemple : Docker version 29.x.x

docker info
# Affiche des détails sur votre installation (client + démon)
```

Puis lancez le conteneur de bienvenue, un grand classique :

terminal

```
docker run hello-world
```

Si tout va bien, Docker télécharge une petite image, lance un conteneur, et affiche un message confirmant que votre installation fonctionne. Que s'est-il passé ? Le client a demandé au démon de lancer l'image `hello-world` ; ne la trouvant pas en local, le démon l'a téléchargée depuis Docker Hub, puis l'a exécutée. Vous venez de réaliser tout le cycle !

ASTUCE — LE DÉMON DOIT TOURNER

Erreur fréquente : `Cannot connect to the Docker daemon`. Cela signifie presque toujours que **Docker Desktop n'est pas démarré** (sur Windows/Mac) ou que le service n'est pas lancé (sur Linux : `sudo systemctl start docker`). Lancez Docker, attendez l'icône de baleine, et ré-essayez.

MODULE 4

Vos premières commandes

> *C'est ici que Docker devient concret. Nous allons manipuler de vrais conteneurs avec une poignée de commandes. Toutes commencent par le mot `docker`, suivi d'un verbe (l'action) et d'arguments.*

Lancer un conteneur : `docker run`

La commande la plus importante. Elle prend une image et en crée un conteneur en cours d'exécution. Essayons avec un serveur web Nginx :

terminal

```
docker run -d -p 8080:80 --name mon-serveur nginx
```

Décortiquons chaque morceau, car cette ligne contient des options que vous réutiliserez sans cesse :

Élément	Signification
`docker run`	Crée et démarre un conteneur.
`-d`	*detached* : le conteneur tourne en arrière-plan et vous rend la main dans le terminal.
`-p 8080:80`	Relie le port **8080 de votre machine** au port **80 du conteneur**. Format : `hôte:conteneur`.
`--name mon-serveur`	Donne un nom lisible au conteneur (sinon Docker en invente un aléatoire).
`nginx`	L'image à utiliser. Absente en local ? Docker la télécharge depuis Docker Hub.

Ouvrez maintenant votre navigateur sur `http://localhost:8080` : la page d'accueil de Nginx s'affiche. Votre conteneur fonctionne !

ATTENTION — LA REDIRECTION DE PORTS

Sans l'option `-p`, un conteneur est totalement isolé : même s'il fait tourner un serveur web, vous ne pourrez pas y accéder depuis votre navigateur. `-p` est le « pont » entre l'intérieur du conteneur et votre machine. C'est l'un des oublis les plus fréquents chez les débutants.

Voir les conteneurs : `docker ps`

terminal

```
docker ps
# Liste les conteneurs EN COURS d'exécution

docker ps -a
# Liste TOUS les conteneurs, y compris ceux qui sont arrêtés
```

Vous y voyez l'identifiant, l'image, le statut, les ports et le nom de chaque conteneur.

Arrêter, redémarrer, supprimer

terminal

```
docker stop mon-serveur      # arrête le conteneur
docker start mon-serveur     # le redémarre
docker restart mon-serveur   # arrêt + redémarrage
docker rm mon-serveur        # le supprime (doit être arrêté d'abord)
docker rm -f mon-serveur     # force la suppression même s'il tourne
```

Voir les logs et entrer dans un conteneur

terminal

```
docker logs mon-serveur       # affiche la sortie du conteneur
docker logs -f mon-serveur    # suit les logs en direct (-f = follow)

# Ouvrir un terminal À L'INTÉRIEUR du conteneur :
docker exec -it mon-serveur bash
```

L'option `-it` (interactif + terminal) vous donne un shell *dans* le conteneur, comme si vous étiez connecté à une petite machine. Tapez `exit` pour en sortir.

ASTUCE — LE MODE INTERACTIF

Pour tester rapidement une image Linux, lancez-la directement en interactif : `docker run -it ubuntu bash`. Vous obtenez un terminal Ubuntu jetable en deux secondes, parfait pour expérimenter sans rien installer sur votre machine.

Faire le ménage

Les conteneurs et images s'accumulent vite et occupent de l'espace disque. Pour nettoyer :

terminal

```
docker system prune          # supprime les éléments inutilisés (conteneurs arrêtés,
etc.)
docker system prune -a       # nettoyage plus agressif (images non utilisées incluses)
```

ATTENTION

`prune` supprime définitivement. Vérifiez qu'aucun conteneur ou volume important n'est concerné avant de confirmer.

MODULE 5

Travailler avec les images

> *Les images sont la matière première de Docker. Avant de fabriquer les vôtres, apprenons à trouver, télécharger, lister et gérer celles qui existent déjà.*

Trouver et télécharger une image

terminal

```
docker search postgres        # cherche des images sur Docker Hub
docker pull postgres          # télécharge l'image (dernière version)
docker pull postgres:16       # télécharge une version PRÉCISE (tag 16)
```

Les tags : choisir une version

Une image porte un nom suivi, après deux-points, d'un **tag** qui indique la version : `python:3.12`, `node:22`, `nginx:1.27`. Si vous ne précisez rien, Docker utilise le tag `latest` (la dernière version par défaut).

ATTENTION — N'ABUSEZ PAS DE `LATEST`

`latest` n'est pas une version figée : ce qu'elle désigne change avec le temps. Pour des projets sérieux, indiquez toujours un tag précis (par ex. `postgres:16`) afin que votre environnement reste reproductible et ne casse pas du jour au lendemain.

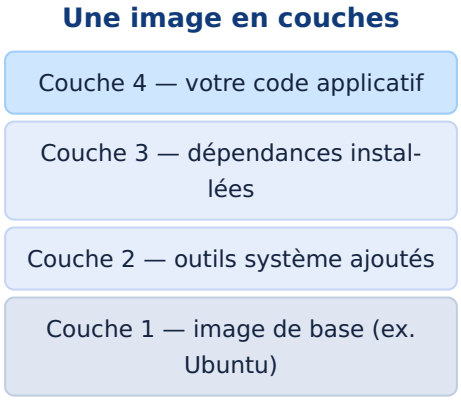
Lister et supprimer des images

terminal

```
docker images               # liste les images présentes en local
docker rmi postgres:16       # supprime une image (rmi = remove image)
docker image prune          # supprime les images orphelines (sans tag)
```

Comprendre les couches (layers)

Une image n'est pas un bloc monolithique : elle est constituée d'une **pile de couches** empilées, chacune correspondant à une étape de sa construction. C'est un détail clé pour comprendre pourquoi Docker est si rapide et économe.



Chaque instruction du Dockerfile ajoute une couche. Les couches identiques sont mises en cache et partagées entre images.

Deux conséquences très pratiques :

- **Le cache accélère tout.** Si une couche n'a pas changé depuis la dernière construction, Docker la réutilise au lieu de la refaire. Les constructions suivantes sont quasi instantanées.
- **Le partage économise l'espace.** Si dix images reposent sur la même base Ubuntu, cette couche n'est stockée qu'une seule fois sur votre disque.

À RETENIR

On **tire** une image avec `pull`, on la **liste** avec `docker images`, on la **supprime** avec `rmi`. Précisez toujours un **tag** de version pour la reproductibilité. Une image est faite de **couches** mises en cache et partagées : c'est le secret de la rapidité de Docker.

MODULE 6

Créer ses propres images : le Dockerfile

> *Jusqu'ici nous avons utilisé des images toutes prêtes. Maintenant, le cœur de Docker : emballer votre application dans votre image, grâce à un **Dockerfile**.*

Qu'est-ce qu'un Dockerfile ?

C'est un fichier texte nommé exactement `Dockerfile` (sans extension), placé à la racine de votre projet. Il contient, ligne par ligne, les instructions que Docker exécute pour construire l'image. Chaque instruction crée une couche.

Exemple complet : une petite application Python

Supposons un mini-projet avec deux fichiers : un programme `app.py` et la liste de ses dépendances `requirements.txt`. Voici un Dockerfile pour l'emballer :

Dockerfile

```
FROM python:3.12-slim
# 1. On part d'une image de base : Python 3.12 en version allégée

WORKDIR /app
# 2. On définit le dossier de travail à l'intérieur de l'image

COPY requirements.txt .
# 3. On copie d'abord la liste des dépendances (astuce de cache, voir plus bas)

RUN pip install --no-cache-dir -r requirements.txt
# 4. On installe les dépendances

COPY . .
# 5. On copie tout le reste du code dans l'image

EXPOSE 5000
# 6. On documente le port utilisé par l'application

CMD ["python", "app.py"]
# 7. La commande lancée au démarrage du conteneur
```

Les instructions essentielles, une par une

Instruction	Rôle
`FROM`	L'image de base sur laquelle on construit. **Toujours la première ligne.**
`WORKDIR`	Définit le répertoire de travail dans l'image ; les commandes suivantes s'y exécutent.
`COPY`	Copie des fichiers de votre machine vers l'image.
`ADD`	Comme COPY, mais sait aussi décompresser une archive ou télécharger une URL. Préférez COPY par défaut.
`RUN`	Exécute une commande **pendant la construction** (installer un paquet, par ex.). Crée une couche.
`ENV`	Définit une variable d'environnement dans l'image.
`EXPOSE`	Documente le port que l'application écoute (informatif ; ne publie pas le port à lui seul).
`CMD`	La commande exécutée **au démarrage du conteneur**. Une seule par Dockerfile.
`ENTRYPOINT`	Comme CMD, mais fixe l'exécutable principal ; souvent combiné avec CMD pour les arguments.

ATTENTION — RUN, CMD ET ENTRYPOINT NE S'EXÉCUTENT PAS AU MÊME MOMENT

`RUN` s'exécute **pendant la construction** de l'image (une fois, pour la fabriquer). `CMD` et `ENTRYPOINT` s'exécutent **au lancement** de chaque conteneur. Confondre les deux est une source classique d'erreurs.

Construire l'image : `docker build`

terminal

```
docker build -t mon-app:1.0 .
```

Élément	Signification
`-t mon-app:1.0`	Donne un nom et un tag à l'image (*t* pour *tag*).
`. (le point)`	Le « contexte de construction » : le dossier courant, où se trouvent le Dockerfile et le code.

Une fois construite, lancez-la comme n'importe quelle image : `docker run -d -p 5000:5000 mon-app:1.0`.

Le fichier `.dockerignore`

À côté du Dockerfile, créez un fichier `.dockerignore` pour exclure de la construction les fichiers inutiles (dépendances locales, fichiers temporaires, secrets). Cela allège l'image et accélère la construction :

.dockerignore

```
# Ne pas copier ces éléments dans l'image
.git
node_modules
__pycache__
*.log
.env
```

ASTUCE — L'ORDRE DES INSTRUCTIONS COMPTE POUR LE CACHE

Dans l'exemple Python, on copie `requirements.txt` et on installe les dépendances *avant* de copier le reste du code. Pourquoi ? Parce que le code change souvent, mais les dépendances rarement. Grâce au cache des couches, tant que `requirements.txt` ne change pas, Docker saute l'étape d'installation (longue) et réutilise la couche existante. Placez toujours ce qui change le moins en premier.

MODULE 7

Persister les données : les volumes

> *Un conteneur est volontairement **éphémère** : quand vous le supprimez, tout ce qui était écrit à l'intérieur disparaît. Pour une base de données ou des fichiers à conserver, c'est inacceptable. La solution : les volumes.*

Le problème de l'éphémère

Lancez une base de données dans un conteneur, ajoutez-y des données, supprimez le conteneur… et tout est perdu. C'est *voulu* : cette légèreté fait la force de Docker. Mais il faut un mécanisme pour stocker durablement ce qui doit l'être, en dehors du cycle de vie du conteneur.

ANALOGIE

Le conteneur est comme une **chambre d'hôtel** : pratique, jetable, on en change facilement. Un volume, c'est votre **valise** : elle reste à vous quelle que soit la chambre, et vous l'emportez d'une chambre à l'autre. On range les affaires importantes dans la valise, pas dans la chambre.

Deux façons de stocker hors du conteneur

1. Les volumes nommés (recommandé)

Docker crée et gère un espace de stockage indépendant, identifié par un nom. C'est la méthode de choix pour les données d'application (bases de données, etc.).

terminal

```
# Créer un volume nommé
docker volume create mes-donnees

# L'attacher à un conteneur : volume:chemin_dans_le_conteneur
docker run -d -v mes-donnees:/var/lib/postgresql/data postgres:16

docker volume ls          # lister les volumes
docker volume rm mes-donnees # supprimer un volume
```

Même si vous supprimez le conteneur Postgres puis en relancez un nouveau attaché au même volume, vos données sont toujours là.

2. Les montages liés (bind mounts)

Ici, vous reliez directement un **dossier de votre machine** à un dossier du conteneur. Très utile en développement : vous modifiez un fichier sur votre ordinateur et le changement est immédiatement visible dans le conteneur.

terminal

```
# chemin_sur_la_machine:chemin_dans_le_conteneur
docker run -d -v $(pwd)/site:/usr/share/nginx/html -p 8080:80 nginx
```

Ici, le contenu du dossier `site` du répertoire courant est servi par Nginx. Modifiez un fichier HTML : rechargez la page, c'est à jour.

	Volume nommé	Montage lié (bind mount)
Géré par	Docker	Vous (chemin sur votre disque)
Usage idéal	Données de production (BDD)	Développement, code source
Portabilité	Haute	Dépend de l'arborescence locale

À RETENIR

Sans volume, les données d'un conteneur disparaissent à sa suppression. Utilisez un **volume nommé** (`-v nom:/chemin`) pour les données durables comme les bases de données, et un **bind mount** (`-v ./dossier:/chemin`) pour travailler sur votre code en direct pendant le développement.

MODULE 8

Les réseaux Docker

> *Une application réelle, c'est rarement un seul conteneur. C'est souvent un serveur web qui parle à une base de données, qui parle à un cache... Les réseaux Docker permettent à ces conteneurs de se trouver et de communiquer.*

Les trois types de réseaux à connaître

Réseau	Comportement
`bridge`	Le mode par défaut. Crée un réseau privé isolé où les conteneurs peuvent communiquer entre eux. C'est celui que vous utiliserez 95 % du temps.
`host`	Le conteneur partage directement le réseau de la machine hôte, sans isolation. Plus rapide, mais moins cloisonné.
`none`	Aucun réseau : le conteneur est totalement isolé. Utile pour des tâches sans besoin réseau.

Faire communiquer deux conteneurs

L'astuce essentielle : sur un **réseau bridge personnalisé**, les conteneurs peuvent s'appeler entre eux **par leur nom**, sans connaître d'adresse IP. Docker fournit une résolution de noms automatique.

terminal

```
# 1. Créer un réseau dédié
docker network create mon-reseau

# 2. Lancer une base de données sur ce réseau
docker run -d --name base --network mon-reseau postgres:16

# 3. Lancer l'application sur le MÊME réseau
docker run -d --name appli --network mon-reseau mon-app:1.0
```

Désormais, depuis le conteneur `appli`, on peut joindre la base simplement à l'adresse `base` (son nom) sur le port de Postgres. Plus besoin de gérer des adresses IP qui changent.

terminal — commandes utiles

```
docker network ls                # lister les réseaux
docker network inspect mon-reseau # voir les détails et les conteneurs connectés
docker network rm mon-reseau     # supprimer un réseau
```

ASTUCE — DOCKER COMPOSE GÈRE LE RÉSEAU POUR VOUS

Bonne nouvelle : dès le module suivant, vous découvrirez Docker Compose, qui crée **automatiquement** un réseau commun pour vos services. Vous n'aurez donc presque jamais à taper ces commandes réseau à la main dans la vraie vie — mais comprendre le principe est indispensable.

MODULE 9

Docker Compose

> *Lancer un seul conteneur à la main, c'est faisable. Mais une application réelle en demande plusieurs, avec leurs ports, volumes et réseaux. Tout taper à la main devient vite ingérable. **Docker Compose** décrit toute votre application dans un seul fichier et la lance d'une commande.*

Le principe

Au lieu de mémoriser de longues commandes `docker run`, vous écrivez un fichier `compose.yaml` qui déclare tous vos **services** (vos conteneurs), leurs réglages et leurs liens. Puis : une commande pour tout démarrer, une pour tout arrêter.

À RETENIR — LA COMMANDE MODERNE

Compose est aujourd'hui intégré à Docker. On l'invoque avec `docker compose` (deux mots, sans tiret). L'ancienne commande `docker-compose` (avec tiret) est dépassée. De même, le fichier s'appelle désormais `compose.yaml` et n'a plus besoin de l'ancienne ligne `version:` en tête.

Exemple : une application web + sa base de données

Voici un fichier Compose qui lance ensemble une application web (construite depuis votre Dockerfile) et une base de données PostgreSQL, reliées entre elles :

compose.yaml

```yaml
services:

  web:
    build: .                  # construit l'image depuis le Dockerfile local
    ports:
      - "8080:5000"           # port_hôte:port_conteneur
    environment:
      DATABASE_URL: postgres://user:secret@base:5432/madb
    depends_on:
      - base                  # démarre « base » avant « web »

  base:
    image: postgres:16        # utilise une image toute prête
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: madb
    volumes:
      - donnees-bdd:/var/lib/postgresql/data   # persistance

volumes:
  donnees-bdd:                # déclare le volume nommé utilisé ci-dessus
```

Remarquez la puissance de ce fichier : l'application web joint sa base de données simplement par son nom de service, `base` (dans `DATABASE_URL`), parce que Compose place automatiquement tous les services sur un réseau commun.

Les commandes de Compose

terminal (dans le dossier du fichier compose.yaml)

```
docker compose up -d       # construit (si besoin) et démarre tout, en arrière-plan
docker compose ps          # liste les services de l'application
docker compose logs -f     # suit les logs de tous les services
docker compose stop        # arrête sans supprimer
docker compose down        # arrête ET supprime conteneurs et réseau
docker compose down -v     # idem + supprime aussi les volumes (perte des données !)
```

ASTUCE — COMPOSE, VOTRE MEILLEUR ALLIÉ

Dans la pratique, presque tous les projets Docker utilisent Compose, même pour un seul service : le fichier sert de documentation vivante de votre application. Quelqu'un qui récupère votre projet n'a qu'à taper `docker compose up` pour tout lancer, sans rien installer d'autre.

MODULE 10

Les bonnes pratiques

> *Faire tourner un conteneur, c'est facile. Le faire bien — léger, sûr, maintenable — demande quelques réflexes. Voici les plus importants pour bien démarrer.*

Garder des images légères

- **Choisissez une base mince.** Préférez les variantes `-slim` ou `-alpine` (par ex. `python:3.12-slim`). Alpine est une distribution Linux minuscule : l'image finale peut être dix fois plus petite.
- **Utilisez les constructions multi-étapes (multi-stage builds).** On compile dans une première étape « lourde », puis on ne copie que le résultat final dans une image « légère ». L'outillage de compilation ne se retrouve pas dans l'image livrée.
- **Soignez le `.dockerignore`** pour ne pas embarquer de fichiers inutiles.

Dockerfile — exemple de multi-stage (application compilée)

```
# Étape 1 : construction (image complète avec les outils)
FROM golang:1.22 AS build
WORKDIR /src
COPY . .
RUN go build -o /app/serveur

# Étape 2 : image finale, minuscule, ne contenant que le binaire
FROM alpine:3.20
COPY --from=build /app/serveur /serveur
CMD ["/serveur"]
```

Sécurité

- **Ne mettez jamais de secrets dans une image.** Mots de passe, clés d'API, jetons : ils ne doivent pas figurer dans le Dockerfile ni être copiés dans l'image. Passez-les par des variables d'environnement ou un gestionnaire de secrets.
- **Ne lancez pas vos conteneurs en root.** Par défaut, le processus tourne en root dans le conteneur. Créez et utilisez un utilisateur dédié avec l'instruction `USER` pour limiter les dégâts en cas de faille.
- **Épinglez vos versions.** Des tags précis (`postgres:16` plutôt que `latest`) évitent qu'une mise à jour surprise casse votre application.
- **Mettez Docker à jour.** Les versions récentes corrigent régulièrement des failles de sécurité. Utilisez une version maintenue du moteur.

ERREUR À NE JAMAIS COMMETTRE

Ne publiez jamais sur un registre public une image contenant un fichier `.env`, une clé privée ou un mot de passe en clair. Une fois poussée, l'image — et tout ce qu'elle contient — peut être téléchargée par n'importe qui. Vérifiez toujours votre `.dockerignore` avant de construire.

Maintenabilité

- **Un conteneur, une responsabilité.** Ne mettez pas l'application *et* sa base de données dans le même conteneur : séparez-les en services distincts (c'est tout l'intérêt de Compose).
- **Rendez vos conteneurs « jetables ».** Ils doivent pouvoir être détruits et recréés à tout moment sans perte : toutes les données importantes vivent dans des volumes, pas dans le conteneur.
- **Versionnez vos Dockerfile et compose.yaml** avec votre code (dans Git). Ils font partie intégrante du projet.

À RETENIR

Images **légères** (bases slim/alpine, multi-stage), **sécurité** (pas de secrets, pas de root, versions épinglées), **conteneurs jetables** avec données dans des volumes, et un conteneur = un rôle. Ces réflexes distinguent un usage amateur d'un usage professionnel.

MODULE 11 · PROJET PRATIQUE

Conteneurisez votre première application

> *Mettons tout en pratique. Objectif : conteneuriser de A à Z une petite application web Python, puis l'orchestrer avec Compose. Suivez les étapes une à une — tout ce dont vous avez besoin a été vu dans les modules précédents.*

Étape 1 — Créer le projet

Créez un dossier `mon-projet` et, à l'intérieur, trois fichiers.

app.py – une mini-application web avec Flask

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def accueil():
    return "Bonjour depuis mon conteneur Docker !"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

ATTENTION — HOST="0.0.0.0"

À l'intérieur d'un conteneur, une application qui n'écoute que sur `localhost` (127.0.0.1) sera **injoignable depuis l'extérieur**. Il faut écouter sur `0.0.0.0` pour accepter les connexions venant de la machine hôte. C'est un piège très courant.

requirements.txt

```
flask
```

Dockerfile

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

Étape 2 — Construire l'image

terminal (dans le dossier mon-projet)

```
docker build -t mon-projet:1.0 .
```

Docker lit le Dockerfile, télécharge la base Python, installe Flask et empaquette votre code. La première fois prend quelques dizaines de secondes ; les suivantes seront bien plus rapides grâce au cache.

Étape 3 — Lancer et tester

terminal

```
docker run -d -p 8080:5000 --name demo mon-projet:1.0
```

Ouvrez `http://localhost:8080` dans votre navigateur. Le message « Bonjour depuis mon conteneur Docker ! » s'affiche. **Félicitations** — vous venez de conteneuriser votre première application !

Étape 4 — Passer à Docker Compose

Pour préparer la suite (ajout d'une base de données, par exemple) et simplifier le lancement, créez un fichier Compose à côté du Dockerfile :

compose.yaml

```
services:
  web:
    build: .
    ports:
      - "8080:5000"
```

Désormais, tout le projet se lance et s'arrête avec deux commandes :

terminal

```
docker compose up -d     # tout démarrer
docker compose down      # tout arrêter et nettoyer
```

Étape 5 — Pour aller plus loin par vous-même

- ✓ Ajoutez un service `base` (PostgreSQL ou Redis) dans le `compose.yaml` et reliez-le à l'application.
- ✓ Ajoutez un volume nommé pour persister les données de cette base.
- ✓ Modifiez le message dans `app.py`, reconstruisez et observez le cache de Docker à l'œuvre.
- ✓ Créez un `.dockerignore` et vérifiez la taille de votre image avec `docker images`.

VOUS SAVEZ MAINTENANT L'ESSENTIEL

Si vous avez réalisé ce projet, vous maîtrisez le cycle complet : écrire un Dockerfile, construire une image, lancer un conteneur, exposer un port et orchestrer avec Compose. C'est exactement ce que font les équipes en entreprise au quotidien. Le reste n'est qu'approfondissement.

Aide-mémoire des commandes

> *Toutes les commandes essentielles rassemblées sur quelques pages, à garder sous la main.*

Images

Commande	Action
`docker pull <image>`	Télécharger une image depuis Docker Hub
`docker images`	Lister les images locales
`docker build -t nom:tag .`	Construire une image depuis un Dockerfile
`docker rmi <image>`	Supprimer une image
`docker image prune`	Supprimer les images orphelines

Conteneurs

Commande	Action
`docker run -d -p 8080:80 nom`	Lancer un conteneur en arrière-plan, port publié
`docker run -it nom bash`	Lancer un conteneur en mode interactif
`docker ps / docker ps -a`	Lister les conteneurs actifs / tous
`docker stop <nom>`	Arrêter un conteneur
`docker start <nom>`	Redémarrer un conteneur arrêté
`docker rm <nom>`	Supprimer un conteneur
`docker logs -f <nom>`	Afficher / suivre les logs
`docker exec -it <nom> bash`	Ouvrir un terminal dans le conteneur

Volumes & réseaux

Commande	Action
`docker volume create <nom>`	Créer un volume nommé
`docker volume ls`	Lister les volumes
`docker run -v nom:/chemin ...`	Attacher un volume à un conteneur
`docker network create <nom>`	Créer un réseau
`docker network ls`	Lister les réseaux

Docker Compose

Commande	Action
`docker compose up -d`	Construire et démarrer tous les services
`docker compose ps`	Lister les services
`docker compose logs -f`	Suivre les logs de tous les services
`docker compose down`	Arrêter et supprimer conteneurs + réseau
`docker compose down -v`	Idem, en supprimant aussi les volumes

Système

Commande	Action
`docker --version`	Version installée
`docker info`	Détails de l'installation
`docker system prune -a`	Grand nettoyage (à utiliser avec prudence)

Glossaire

Terme	Définition
Image	Modèle figé, en lecture seule, contenant tout le nécessaire pour faire tourner une application. Sert de base aux conteneurs.
Conteneur	Instance en cours d'exécution d'une image. Isolé, léger et éphémère.
Dockerfile	Fichier texte décrivant, étape par étape, comment construire une image.
Registre	Entrepôt en ligne d'images. Docker Hub en est le principal exemple public.
Docker Hub	Le registre public officiel, riche en images prêtes à l'emploi.
Couche (layer)	Strate d'une image correspondant à une instruction du Dockerfile. Mise en cache et partagée.
Tag	Étiquette de version d'une image, après les deux-points (ex. `python:3.12`).
Volume	Espace de stockage géré par Docker qui survit à la suppression d'un conteneur.
Bind mount	Liaison directe entre un dossier de la machine hôte et un dossier du conteneur.
Réseau bridge	Réseau privé par défaut permettant aux conteneurs de communiquer.
Démon (dockerd)	Service tournant en arrière-plan qui exécute réellement le travail de Docker.
Client	La commande `docker` qui transmet vos ordres au démon.
Docker Compose	Outil pour décrire et lancer une application multi-conteneurs via un fichier `compose.yaml`.
Service	Dans Compose, un conteneur (et sa configuration) déclaré dans le fichier.
Build	L'action de construire une image à partir d'un Dockerfile.

Pour aller plus loin

> *Vous tenez maintenant des fondations solides. Voici les prochaines étapes naturelles si vous souhaitez approfondir.*

Sujets à explorer ensuite

- **Docker Compose avancé** — variables d'environnement dans un fichier `.env`, profils, fichiers de surcharge, vérifications de santé (healthchecks).
- **Optimisation des images** — multi-stage builds poussés, choix de bases minimales, analyse de la taille et de la sécurité des images.
- **Publier ses images** — créer un compte Docker Hub, taguer et `push` ses propres images pour les partager.
- **L'orchestration à grande échelle** — quand un seul serveur ne suffit plus, on passe à des orchestrateurs comme **Kubernetes** ou Docker Swarm, qui gèrent des conteneurs sur de nombreuses machines.
- **Intégration continue (CI/CD)** — construire et tester automatiquement vos images à chaque modification du code.

Ressources de référence

- **Documentation officielle** — `docs.docker.com` : claire, à jour, avec des tutoriels guidés. La meilleure source.
- **Docker Hub** — `hub.docker.com` : pour explorer les images officielles et lire leur documentation d'utilisation.
- **La commande d'aide intégrée** — n'oubliez jamais que `docker <commande> --help` documente chaque commande directement dans votre terminal.

LE MOT DE LA FIN

Docker s'apprend par la pratique, pas par la théorie. Reprenez le projet du Module 11, modifiez-le, cassez-le, réparez-le. Chaque erreur rencontrée et résolue vaut dix pages de lecture. Vous avez désormais toutes les clés pour commencer : lancez votre terminal et expérimentez.