

Docker for Teenagers

**CONVIENT AUSSI AUX
ADULTES**



Installation et vérification



```
● ● ●

# Installation Debian
sudo apt install docker.io

# Installation Red Hat (Fedora, CentOS, etc)
sudo yum install docker

# Installation MacOS (via Homebrew)
brew install docker

# Installation Windows (via Chocolatey)
choco install docker-desktop
```

```
● ● ●

# Version
docker --version
$> Docker version 20.10.7, build f0df350
```

```
● ● ●

# Sur Linux
sudo systemctl start docker

# Sur MacOS, Windows, lancez l'application Docker Desktop
```

DockerFile

Exemple : nginx



Un Dockerfile est une configuration scriptée utilisée pour créer des images de conteneurs Docker. Il utilise des commandes spécifiques à Docker, comme FROM, RUN, COPY, et EXPOSE, pour définir les étapes de construction d'une image.

Dans le contexte de Nginx, un Dockerfile typique inclurait l'installation de Nginx sur une image de base Linux, en exposant les ports nécessaires et en configurant les commandes pour démarrer le serveur web.

Tips pour Nginx

- **Image de Base Légère:** Préférez des images légères comme base pour réduire la taille de l'image Docker.
- **Nettoyage Après Installation:** Nettoyez les fichiers temporaires après l'installation pour éviter une image surdimensionnée.
- **Minimisation des Couches:** Regroupez les commandes RUN pour réduire le nombre de couches dans l'image.

DockerFile

Exemple : nginx



```
● ● ●

# Utilise une image de base pour créer la nouvelle image.
# Exemple : Ubuntu, Alpine, etc
FROM ubuntu:latest

# Définit une variable d'environnement utilisée dans le
# Dockerfile
ENV APP_HOME /app

# Définit le répertoire de travail dans le conteneur
WORKDIR $APP_HOME

# Copie des fichiers ou répertoires du contexte de
# construction local vers l'image
COPY . $APP_HOME

# Exécute des commandes pour installer des logiciels ou
# configurer l'image
RUN apt-get update && apt-get install -y nginx

# Informe Docker que le conteneur écoute sur les ports
# spécifiés lors de l'exécution
EXPOSE 80

# Définit la commande par défaut à exécuter lorsque le
# conteneur démarre
# Si vous avez besoin que votre conteneur s'exécute comme un
# exécutable, utilisez ENTRYPPOINT à la place de CMD.
CMD ["nginx", "-g", "daemon off;"]

# Crée un point de montage pour attacher un volume
VOLUME /var/www/html
```

DockerFile

Exemple : build .Net App



Le Dockerfile pour une application .NET établit un environnement uniforme en installant les dépendances .NET, en copiant les fichiers sources, et en spécifiant la commande pour exécuter l'application. En plus, il est important de noter qu'avec Docker, il est possible de compiler pratiquement n'importe quel type d'application, pas seulement des applications .NET, grâce à la flexibilité et à la variété des images disponibles.

Tips pour les Applications .NET

- **Builds Multi-étapes:** Optez pour des constructions en plusieurs étapes pour réduire la taille de l'image finale.
- **Versions Spécifiques:** Utilisez des versions précises pour les images de base et les dépendances pour la sécurité et la compatibilité.
- **Optimisation des Couches:** Organisez les instructions pour maximiser l'efficacité du cache des couches Docker et accélérer le build.

Ces conseils visent à optimiser la création d'images Docker, que ce soit pour un serveur web comme Nginx ou pour une application complexe comme une application .NET, en garantissant efficacité, sécurité et performance.

DockerFile

Exemple : build .Net App



```
● ● ●

# Utilise l'image SDK .NET pour construire l'application
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build-env
WORKDIR /app

# Copie le fichier csproj et restaure les dépendances
COPY *.csproj ./
RUN dotnet restore

# Copie les autres fichiers du projet et construit
l'application
COPY . ./
RUN dotnet publish -c Release -o out

# Utilise l'image runtime .NET pour exécuter l'application
FROM mcr.microsoft.com/dotnet/aspnet:5.0
WORKDIR /app
COPY --from=build-env /app/out .

# Expose le port sur lequel l'application va s'exécuter
EXPOSE 80

# Définit la commande pour démarrer l'application
CMD ["dotnet", "VotreApplication.dll"]
```

CLI



Les interfaces en ligne de commande (CLI) pour Docker sont essentielles pour interagir avec Docker, fournissant un contrôle détaillé et précis sur la gestion des conteneurs, des images, des réseaux et des volumes. Bien que moins visuelles que les interfaces graphiques utilisateur (GUI), les CLI offrent une flexibilité et une puissance supérieures pour les utilisateurs expérimentés.

Avantages des CLI Docker

- 1. Contrôle Précis:** Permettent des opérations détaillées et spécifiques sur les conteneurs et les images.
- 2. Automatisation:** Facilitent l'automatisation des tâches Docker via des scripts.
- 3. Utilisation en Environnement Headless:** Idéales pour les serveurs sans interface graphique.
- 4. Intégration avec les Outils de DevOps:** Se marient bien avec des outils de CI/CD, de versioning, et d'autres outils de développement.

Points Clés

- **Puissance et Flexibilité:** Offrent une gamme étendue de commandes pour toutes les fonctions de Docker.
- **Apprentissage:** Nécessitent un apprentissage pour maîtriser, mais offrent une compréhension profonde de Docker.
- **Préférées des Développeurs et DevOps:** Sont souvent le choix de prédilection pour les professionnels techniques en raison de leur efficacité et de leur intégrabilité dans des flux de travail complexes.

En résumé, les CLI pour Docker sont un outil puissant pour les utilisateurs avancés, permettant une gestion précise et efficace des environnements Docker, tout en offrant les capacités nécessaires pour l'automatisation et l'intégration dans des systèmes plus larges de développement et d'opérations.

Gestion des conteneurs



```
# Créer et lancer un conteneur à partir d'une image.  
# docker run [options] [nom_de_l'image]  
# -d : exécute le conteneur en arrière-plan (mode détaché).  
# -p 80:80 : mappe le port 80 du conteneur sur le port 80 de  
l'hôte.  
docker run -d -p 80:80 nginx  
  
# Affiche la liste des conteneurs actifs avec des détails  
comme ID, image, statut, ports, etc.  
docker ps  
  
# Arrêter un conteneur en cours d'exécution.  
# docker stop [ID_ou_nom_du_conteneur]  
docker stop monconteneur  
  
# Redémarrer un conteneur arrêté.  
# docker restart [ID_ou_nom_du_conteneur]  
docker restart monconteneur  
  
# Supprimer un conteneur spécifique.  
# docker rm [ID_ou_nom_du_conteneur]  
docker rm monconteneur
```

Gestion des images



```
# Construire une image à partir d'un Dockerfile.  
# docker build -t [nom_de_l'image] [chemin_du_dossier]  
# -t : nomme l'image construite.  
# [chemin_du_dossier] : spécifie le dossier contenant le  
Dockerfile, . pour le dossier courant.  
docker build -t monimage .  
  
# Affiche la liste des images avec des détails comme ID,  
taille, tag, etc.  
docker images  
  
# Supprimer une image Docker spécifique.  
# docker rmi [ID_ou_nom_de_l'image]  
docker rmi monimage  
  
# Liste détaillée des couches de l'image et de leurs  
changements.  
# docker history [nom_de_l'image]  
docker history nginx
```

Gestion du réseau



```
# Affiche la liste des réseaux avec des détails comme ID,  
nom, driver, etc.  
docker network ls  
  
# Créer un nouveau réseau Docker.  
# docker network create [options] [nom_du_réseau]  
# --driver : spécifie le type de réseau, comme bridge,  
overlay, none, etc.  
docker network create --driver bridge monreseau  
  
# Connecter un conteneur à un réseau Docker.  
# docker network connect [nom_du_réseau] [nom_du_conteneur]  
docker network connect monreseau monconteneur  
  
# Inspecter les détails d'un réseau Docker.  
# docker network inspect [nom_du_réseau]  
docker network inspect monreseau
```

Gestion des données



```
● ● ●

# Créer un nouveau volume Docker pour le stockage persistant.
# docker volume create [nom_du_volume]
docker volume create monvolume

# Affiche la liste des volumes avec des détails comme le nom,
# le driver, etc.
docker volume ls

# Attacher un volume à un conteneur pour le stockage
# persistant.
# docker run -v [nom_du_volume]:[chemin_dans_le_conteneur]
# [nom_de_l'image]
# -v : attache le volume nommé à un chemin spécifique dans le
# conteneur.
# [chemin_dans_le_conteneur] : le chemin où le volume sera
# monté dans le conteneur.
docker run -v monvolume:/data nginx

# Supprimer un volume Docker spécifique.
# docker volume rm [nom_du_volume]
docker volume rm monvolume
```

Monitoring et logs



● ● ●

```
# Affiche les logs de sortie du conteneur spécifié.  
# docker logs [ID_ou_nom_du_conteneur]  
docker logs monconteneur  
  
# Surveiller l'utilisation des ressources par les conteneurs  
en temps réel, y compris l'utilisation du CPU, de la mémoire,  
du réseau, etc.  
docker stats  
  
# Ouvre un shell interactif dans le conteneur, si bash est  
utilisé comme commande.  
# docker exec [options] [ID_ou_nom_du_conteneur] [commande]  
# -it : attache une session interactive TTY.  
docker exec -it monconteneur bash  
  
# Affiche un JSON détaillé avec des informations complètes  
sur le conteneur, y compris l'état du réseau, les volumes  
montés, les paramètres de configuration, etc.  
# docker inspect [ID_ou_nom_du_conteneur]  
docker inspect monconteneur
```

Débogage



```
# Affiche un JSON détaillé avec des informations complètes
# sur le conteneur, y compris l'état du réseau, les volumes
# montés, les paramètres de configuration, etc.
# docker inspect [ID_ou_nom_du_conteneur]
docker inspect monconteneur

# Sauvegarder un conteneur dans une nouvelle image. Utile
# pour créer une image à partir d'un conteneur modifié.
# docker commit [ID_ou_nom_du_conteneur] [nom_de_l'image]
docker commit monconteneur monimage

# Copier des fichiers ou dossiers depuis et vers un
# conteneur. Très utile pour le transfert de données.
## Pour copier du conteneur vers l'hôte:
# docker cp [ID_ou_nom_du_conteneur]:
# [chemin_dans_le_conteneur] [chemin_local]
docker cp monconteneur:/fichier.txt /mon/dossier/local

## Pour copier de l'hôte vers le conteneur:
# docker cp [chemin_local] [ID_ou_nom_du_conteneur]:
# [chemin_dans_le_conteneur]
docker cp /mon/dossier/local/fichier.txt monconteneur:/

# Liste les fichiers et dossiers qui ont été ajoutés,
# modifiés ou supprimés dans le conteneur.
# docker diff [ID_ou_nom_du_conteneur]
docker diff monconteneur
```

Nettoyage



```
# Nettoyer les ressources inutilisées telles que les conteneurs arrêtés, les réseaux non utilisés, et les images en cache.
```

```
docker system prune
```

```
# Supprimer les conteneurs arrêtés, les volumes non utilisés, les réseaux non utilisés, et les images non utilisées (y compris les images en cache).
```

```
docker system prune -a
```

```
# Supprimer toutes les images non utilisées, pas seulement les images en cache.
```

```
docker image prune -a
```

```
# Supprimer tous les conteneurs arrêtés.
```

```
docker container prune
```

```
# Supprimer tous les volumes non utilisés.
```

```
docker volume prune
```

Gestion des registres

Exemple : DockerHub



```
# Connexion à DockerHub pour permettre le push et le pull
# d'images.
# docker login -u [nom_utilisateur] -p [mot_de_passe]
docker login -u monnomutilisateur -p monmotdepasse
# docker login -u [nom_utilisateur] -p [token_d'accès]
docker login -u monnomutilisateur -p 9f86d081884c7d659a2feaa0

# Télécharger une image depuis DockerHub sur le système
# local.
# docker pull [nom_de_l'image]
docker pull ubuntu

# Pousser une image locale sur DockerHub.
# docker push [nom_utilisateur/nom_de_l'image]
docker push monutilisateur/monimage

# Recherche d'images sur DockerHub.
# docker search [terme_de_recherche]
docker search mysql
```

Docker Compose



Docker Compose est un outil de Docker utilisé pour définir et gérer des applications multi-conteneurs. Il se base sur un fichier YAML pour configurer les services de l'application, ce qui simplifie le déploiement et la gestion de conteneurs interdépendants.

Avantages de Docker Compose

1. Configuration Simple: Utilisation d'un fichier YAML pour une configuration claire et facile.
2. Gestion Intégrée: Lance et gère plusieurs conteneurs comme un seul service groupé.
3. Isolation des Environnements: Assure la cohérence entre les environnements de développement, de test et de production.

Utilisation

Docker Compose est idéal pour le développement local, les tests automatisés et le déploiement en production d'applications composées de plusieurs services interconnectés.

En bref, Docker Compose est essentiel pour gérer facilement des applications complexes sur Docker, offrant une approche structurée et efficace pour orchestrer des services multiples.

Dans l'exemple suivant, le fichier `docker-compose.yaml` permet de lancer deux conteneurs en même temps sur les ports 8080 (qui se map sur le port 80 du conteneur) et le port 1433 tout en passant quelques variables d'environnement.

Docker Compose



```
version: '3.8'

services:
  webapp:
    image: monappnet:latest
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8000:80"
    environment:
      -
      "ConnectionStrings:DefaultConnection=Server=db;Database=MyDb;
      User=sa;Password=Your_password123;"
    depends_on:
      - db

  db:
    image: mcr.microsoft.com/mssql/server:2019-latest
    environment:
      SA_PASSWORD: "Your_password123"
      ACCEPT_EULA: "Y"
    ports:
      - "1433:1433"
```

Gestion de Docker Compose



```
● ● ●

# Démarrer les services spécifiés dans le fichier docker-compose.yml.
# -d : exécute le conteneur en arrière-plan (mode détaché)
docker compose up -d

# Arrête les services et nettoie les ressources associées.
docker compose down

# Afficher les logs de tous les services.
docker compose logs

# Exécuter une commande unique dans un service.
# docker compose run [nom_du_service] [commande]
docker compose run webapp echo "Hello World"

# Mettre à jour et reconstruire un service spécifique.
# docker compose up -d --no-deps [nom_du_service]
docker compose up -d --no-deps webapp

# Arrêter un service spécifique.
# docker compose stop [nom_du_service]
docker compose stop webapp
```