🐳 Docker — Le Dockerfile

Objectifs

À la fin de cette page, tu seras capable de :

- Expliquer le rôle d'un Dockerfile
- Utiliser les instructions essentielles : `FROM`, `WORKDIR`, `COPY`, `RUN`, `EXPOSE`, `CMD`, `ENV`
- Construire une image personnalisée avec `docker build`
- Créer une image d'un site HTML statique servi par Nginx
- Créer une image d'une application PHP

---

5 notions-clés

1. `FROM` — L'image de base dont on hérite (point de départ obligatoire)
2. `COPY` — Copie des fichiers de ta machine vers l'image en construction
3. `RUN` — Exécute une commande shell **pendant la construction** de l'image
4. `CMD` — Définit la commande qui s'exécute **au démarrage** du conteneur
5. `docker build` — La commande qui lit le Dockerfile et fabrique l'image

---

Qu'est-ce qu'un Dockerfile ?

Jusqu'ici, on a utilisé des images existantes (`nginx`, `mysql`, `php`). Mais que faire quand on veut :

- Un serveur web qui contient **déjà notre application**
- Une image PHP avec des **extensions supplémentaires** installées
- Une image Node.js avec nos **dépendances npm** déjà installées

C'est là qu'intervient le **Dockerfile** : un simple fichier texte qui décrit, **étape par étape**, comment construire une image sur mesure.

```
Mon code source  +  Dockerfile  →  docker build  →  Mon Image  →  docker run  →  Conteneur
```

Docker lit le Dockerfile de haut en bas et crée une **nouvelle couche** à chaque instruction. Ces couches sont mises en cache : si rien n'a changé, Docker ne les reconstruit pas.

Les instructions essentielles

`FROM` — L'image de base

```dockerfile
FROM nginx:1.25
FROM php:8.3-apache
FROM node:20-alpine
FROM ubuntu:22.04
```

- **Obligatoire** et toujours en première instruction
- On part toujours d'une image existante, jamais de zéro (ou presque)
- `alpine` désigne des images très légères basées sur Alpine Linux (quelques Mo)

`WORKDIR` — Le répertoire de travail

```dockerfile
WORKDIR /var/www/html
```

Définit le répertoire courant pour les instructions suivantes (`COPY`, `RUN`, `CMD`). Crée le dossier s'il n'existe pas.

```dockerfile
FROM php:8.3-apache
WORKDIR /var/www/html
# Maintenant, toutes les instructions suivantes
# s'exécutent dans /var/www/html
```

`COPY` — Copier des fichiers

```dockerfile
# Copier un fichier
COPY index.php .

# Copier un dossier entier
COPY src/ .

# Copier avec un chemin destination explicite
COPY config/php.ini /usr/local/etc/php/php.ini
```

Syntaxe : `COPY source destination`

- `source` : chemin relatif sur **ta machine** (à côté du Dockerfile)
- `destination` : chemin dans **l'image en construction**

`RUN` — Exécuter une commande lors de la construction

```dockerfile
# Installer des paquets Linux
RUN apt-get update && apt-get install -y curl zip

# Installer une extension PHP
RUN docker-php-ext-install pdo pdo_mysql

# Installer des dépendances npm
RUN npm install

# Créer un dossier
RUN mkdir -p /var/log/monapp
```

💡 **Bonne pratique** : Enchaîner les commandes avec `&&` dans un seul `RUN` pour limiter le nombre de couches et réduire la taille de l'image.

```
# ❌ Crée 3 couches inutiles
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y zip

# ✅ Une seule couche
RUN apt-get update && apt-get install -y curl zip
```

`EXPOSE` — Documenter le port

```
EXPOSE 80
EXPOSE 3000
EXPOSE 8080
```

Cette instruction **documente** le port sur lequel l'application écoute. Elle ne publie pas le port sur la machine hôte (ça, c'est `-p` dans `docker run`). C'est une indication pour les utilisateurs de l'image.

`ENV` — Variables d'environnement

```
ENV APP_ENV=production
ENV PORT=3000
ENV DB_HOST=localhost
```

Ces variables sont disponibles dans le conteneur pendant son exécution.

```
FROM node:20-alpine
ENV NODE_ENV=production
ENV PORT=3000
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["node", "server.js"]
```

`CMD` — La commande de démarrage

```
CMD ["nginx", "-g", "daemon off;"]
CMD ["php", "-S", "0.0.0.0:80"]
CMD ["node", "server.js"]
```

- S'exécute **quand le conteneur démarre** (pas à la construction)
- Il ne peut y avoir **qu'un seul** `CMD` par Dockerfile
- Si tu passes une commande à `docker run`, elle remplace `CMD`
- Utilise la syntaxe avec tableau `["commande", "argument"]` (forme `exec`) — plus fiable que la forme texte

💡 Les images officielles définissent déjà un `CMD`. Par exemple, `nginx` démarre nginx, `mysql` démarre le serveur MySQL. Tu n'as souvent pas besoin de le redéfinir.

`ENTRYPOINT` — Point d'entrée fixe (avancé)

```
ENTRYPOINT ["python", "app.py"]
```

Similaire à `CMD` mais **ne peut pas être remplacé** par `docker run`. On l'utilise quand l'image est dédiée à un seul outil. Moins courant pour les débutants.

`ADD` — Copier (avec des super-pouvoirs)

```
ADD archive.tar.gz /var/www/html/
ADD https://exemple.com/fichier.txt /tmp/
```

Comme `COPY`, mais peut aussi **décompresser** des archives et télécharger depuis une URL. Dans la pratique, **préfère `COPY`** pour la lisibilité — `ADD` est réservé aux cas où on a vraiment besoin de ses fonctionnalités supplémentaires.

`ARG` — Argument de construction

```
ARG VERSION=1.0
ARG ENV=dev
```

Comme `ENV`, mais disponible **uniquement pendant la construction** (pas à l'exécution). Utile pour paramétrer un build.

```
docker build --build-arg VERSION=2.0 -t mon-app .
```

Récapitulatif des instructions

Instruction	Moment d'exécution	Rôle principal
`FROM`	Construction	Image de base
`WORKDIR`	Construction	Répertoire de travail
`COPY`	Construction	Copier des fichiers locaux
`ADD`	Construction	Copier + décompresser / URL
`RUN`	Construction	Exécuter une commande
`ENV`	Construction + Exécution	Variable d'environnement
`ARG`	Construction uniquement	Paramètre de build
`EXPOSE`	— (documentation)	Documenter le port
`CMD`	Démarrage du conteneur	Commande par défaut
`ENTRYPOINT`	Démarrage du conteneur	Point d'entrée fixe

La commande `docker build`

```
# Syntaxe générale
docker build -t nom:tag chemin

# Construire depuis le dossier courant (le plus courant)
docker build -t mon-site:1.0 .

# Construire avec un Dockerfile ailleurs
docker build -t mon-site -f chemin/vers/Dockerfile .

# Forcer la reconstruction sans cache
docker build --no-cache -t mon-site .
```

Option	Signification
`-t nom:tag`	Donner un nom et une version à l'image
`.`	Le **contexte de build** : le dossier que Docker envoie au moteur
`-f`	Spécifier un Dockerfile à un autre emplacement
`--no-cache`	Ne pas utiliser le cache des couches précédentes

Exercice C — Serveur web avec une page HTML personnalisée

Objectif : Créer une image Docker qui sert une page HTML custom via Nginx.

Ce que tu vas apprendre : `FROM`, `COPY`, `docker build`, `docker run`, le contexte de build

---

Structure du projet

Crée le dossier suivant sur ton Bureau :

```
docker-html/
├── Dockerfile
└── html/
    └── index.html
```

Étape 1 — Créer la page HTML

Crée `html/index.html` :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Mon premier conteneur</title>
 <style>
      body {
```

```
        font-family: sans-serif;
        background: #1a1a2e;
        color: #e0e0e0;
        display: flex;
        flex-direction: column;
        align-items: center;
        justify-content: center;
        height: 100vh;
        margin: 0;
      }
      h1 { color: #00d4ff; font-size: 3rem; }
      p { font-size: 1.2rem; opacity: 0.8; }
  </style>
  </head>
  <body>
    <h1>🐬 Hello Docker !</h1>
    <p>Ce site est servi depuis un conteneur Docker.</p>
    <p>Image : Nginx 1.25 — construite avec mon Dockerfile</p>
  </body>
  </html>
```

Étape 2 — Écrire le Dockerfile

Crée le fichier `Dockerfile` (sans extension) à la racine du projet :

```
# On part de l'image officielle Nginx
FROM nginx:1.25

# On copie notre dossier html/ dans le répertoire
# par défaut que Nginx utilise pour servir les fichiers
COPY html/ /usr/share/nginx/html/

# Documentation : Nginx écoute sur le port 80
EXPOSE 80

# Nginx est déjà configuré comme CMD dans l'image officielle,
# pas besoin de le redéfinir ici
```

Étape 3 — Construire l'image

Ouvre un terminal dans le dossier `docker-html/` :

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

Analyse de la sortie :

```
[1/2] FROM docker.io/library/nginx:1.25    ← téléchargement de l'image de base
[2/2] COPY html/ /usr/share/nginx/html/   ← copie de notre fichier
Successfully built abc123def456
Successfully tagged mon-site:1.0           ← image créée avec succès ✅
```

Vérifie que l'image apparaît dans la liste :

```
docker images
```

Tu dois voir `mon-site` avec le tag `1.0`.

Étape 4 — Lancer un conteneur depuis ton image

```
docker run -d --name test-site -p 8080:80 mon-site:1.0
```

Ouvre **http://localhost:8080** → tu vois ta page HTML 🎉

Étape 5 — Modifier et reconstruire

Modifie le `<h1>` dans `index.html` (change le texte).

Reconstruction :

```
docker build -t mon-site:1.1 .
```

Arrête l'ancien conteneur et lance le nouveau :

```
docker stop test-site
docker rm test-site
docker run -d --name test-site -p 8080:80 mon-site:1.1
```

Recharge la page : la modification est visible.

💡 **Observation** : La reconstruction est quasi-instantanée. Docker réutilise le cache de la couche `FROM nginx:1.25` (inchangée) et ne recalcule que la couche `COPY`.

Étape 6 — Nettoyage

```
docker stop test-site
docker rm test-site
docker rmi mon-site:1.0 mon-site:1.1
```

Exercice A — Packager une application PHP

Objectif : Créer une image Docker qui contient une vraie mini-application PHP (plusieurs fichiers, connexion à une base de données).

Ce que tu vas apprendre : `RUN` pour installer des extensions, `ENV`, image PHP officielle, variables d'environnement au runtime

---

Structure du projet

```
docker-php-app/
├── Dockerfile
├── docker-compose.yml
├── .env
└── src/
    ├── index.php
    ├── config.php
    └── pages/
        ├── liste.php
        └── ajouter.php
```

Étape 1 — L'application PHP

`src/config.php` — connexion à la base de données :

```php
<?php
// On lit les variables d'environnement injectées par Docker
$host = getenv('DB_HOST') ?: 'db';
$dbname = getenv('DB_NAME') ?: 'apptest';
$user = getenv('DB_USER') ?: 'root';
$password = getenv('DB_PASSWORD') ?: '';

try {
    $pdo = new PDO(
        "mysql:host=$host;dbname=$dbname;charset=utf8",
        $user,
        $password,
        [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]
    );
} catch (PDOException $e) {
    die("Erreur de connexion : " . $e->getMessage());
}
?>
```

`src/index.php` — page d'accueil :

```php
<?php
require 'config.php';

// Création de la table si elle n'existe pas
$pdo->exec("CREATE TABLE IF NOT EXISTS messages (
    id INT AUTO_INCREMENT PRIMARY KEY,
    texte VARCHAR(255) NOT NULL,
    cree_le DATETIME DEFAULT CURRENT_TIMESTAMP
)");
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>App Docker PHP</title>
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css">
</head>
<body class="bg-dark text-light p-4">
    <div class="container">
```

```
        <h1 class="mb-4">🐬 App PHP dans Docker</h1>

        <!-- Formulaire d'ajout -->
        <form method="POST" action="pages/ajouter.php" class="mb-4">
            <div class="input-group">
          <input type="text" name="texte" class="form-control"
                 placeholder="Ton message..." required>
                <button type="submit" class="btn btn-primary">Envoyer</button>
            </div>
        </form>

        <!-- Liste des messages -->
        <h2 class="h5 mb-3">Messages enregistrés :</h2>
    <?php
        $stmt = $pdo->query("SELECT * FROM messages ORDER BY cree_le DESC");
        $messages = $stmt->fetchAll(PDO::FETCH_ASSOC);

        if (empty($messages)) {
            echo '<p class="text-muted">Aucun message pour l\'instant.</p>';
        } else {
            foreach ($messages as $msg) {
                echo '<div class="card bg-secondary mb-2">';
                echo '  <div class="card-body py-2">';
                echo '    <span>' . htmlspecialchars($msg['texte']) . '</span>';
                echo '    <small class="text-muted ms-3">' . $msg['cree_le'] . '</small>';
                echo '  </div>';
                echo '</div>';
            }
        }
    ?>

        <p class="mt-4 text-muted small">
            Connecté à : <?= $host ?> / <?= $dbname ?>
        </p>
    </div>
</body>
</html>
```

`src/pages/ajouter.php` :

```
<?php
require '../config.php';

if ($_SERVER['REQUEST_METHOD'] === 'POST' && !empty($_POST['texte'])) {
    $stmt = $pdo->prepare("INSERT INTO messages (texte) VALUES (?)");
    $stmt->execute([htmlspecialchars($_POST['texte'])]);
}

header('Location: ../index.php');
exit;
?>
```

Étape 2 — Le Dockerfile

```dockerfile
# Image officielle PHP avec Apache intégré
FROM php:8.3-apache

# Installer les extensions nécessaires
# pdo_mysql : permet à PHP de se connecter à MySQL via PDO
RUN docker-php-ext-install pdo pdo_mysql

# Définir le répertoire de travail
WORKDIR /var/www/html

# Copier le code source de l'application
COPY src/ .

# Apache écoute sur le port 80 (déjà défini dans l'image de base)
EXPOSE 80

# CMD est déjà défini dans php:8.3-apache
# Il démarre Apache automatiquement
```

Pourquoi `docker-php-ext-install` ?
L'image `php:8.3-apache` est volontairement minimale. Les extensions comme `pdo_mysql` ne sont pas incluses par défaut pour alléger l'image. La commande `docker-php-ext-install` est un script fourni par l'image officielle PHP pour les installer proprement.

Étape 3 — Le docker-compose.yml

On ne lance pas le conteneur PHP seul : il a besoin d'une base de données MySQL.

```yaml
services:

  # Le service base de données
  db:
    image: mysql:8.0
    container_name: phpapp_db
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: ${DB_PASSWORD}
      MYSQL_DATABASE: ${DB_NAME}
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - appnet

  # Notre application PHP (construite depuis le Dockerfile)
  app:
    build: .                    # ← Docker va lire le Dockerfile du dossier courant
    container_name: phpapp_web
    restart: unless-stopped
    ports:
      - "8080:80"
    environment:
      DB_HOST: db               # ← le nom du service MySQL dans le réseau Docker
      DB_NAME: ${DB_NAME}
      DB_USER: root
      DB_PASSWORD: ${DB_PASSWORD}
```

```
    networks:
      - appnet
    depends_on:
      - db

volumes:
  db_data:

networks:
  appnet:
```

Étape 4 — Le fichier .env

```
DB_NAME=apptest
DB_PASSWORD=monmotdepasse
```

Étape 5 — Construire et lancer

```
# Se placer dans le dossier docker-php-app/
cd docker-php-app

# Construire l'image PHP et démarrer tous les services
docker compose up -d --build
```

L'option `--build` force Docker à (re)construire l'image depuis le Dockerfile avant de démarrer.

Surveille le démarrage :

```
docker compose logs -f
```

Attends que MySQL soit prêt (tu verras `ready for connections` dans les logs). Appuie sur `Ctrl+C` pour quitter les logs sans arrêter les conteneurs.

Étape 6 — Tester l'application

Ouvre **http://localhost:8080**

- Tu dois voir la page de l'application
- Saisis un message et envoie
- Actualise la page : le message est enregistré en base de données ✅

Étape 7 — Vérifier que les données persistent

```
# Arrêter et supprimer les conteneurs
docker compose down

# Relancer (sans --build cette fois, l'image est déjà construite)
docker compose up -d
```

```
6
7  # Ouvrir http://localhost:8080 → les messages sont toujours là ✅
```

Étape 8 — Modifier l'application et reconstruire

Modifie le titre `<h1>` dans `src/index.php`.

```
1  # Reconstruire uniquement le service "app" et redémarrer
2  docker compose up -d --build app
```

Recharge la page → la modification est visible.

Étape 9 — Nettoyage complet

```
1  # Arrêter les services, supprimer les conteneurs, le réseau ET les volumes
2  docker compose down -v
3
4  # Supprimer les images construites
5  docker rmi docker-php-app-app
```

Bonne pratique : le fichier `.dockerignore`

Comme `.gitignore`, le fichier `.dockerignore` dit à Docker quels fichiers **ne pas inclure** dans le contexte de build. Ça accélère la construction et évite d'envoyer des fichiers inutiles (node_modules, .env, .git...).

Crée `.dockerignore` à côté du `Dockerfile` :

```
.env
.git
*.log
node_modules
```