🐳 Docker — Bonnes pratiques

Objectifs

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

- Appliquer un workflow Docker progressif et logique
- Nommer et organiser tes conteneurs correctement
- Protéger les mots de passe avec un fichier `.env`
- Choisir la bonne approche selon la situation (run, compose, build)
- Éviter les erreurs classiques des débutants

---

5 notions-clés

1. **Workflow** — La progression logique : `pull → run → compose → build`
2. **Versionning** — Toujours préciser la version d'une image, jamais `latest` en production
3. **Fichier `.env`** — Fichier texte qui contient les variables sensibles (mots de passe, clés API)
4. **Principe du moindre privilège** — Un conteneur = un seul service, pas un système complet
5. **Nettoyage** — Docker accumule des images et conteneurs inutilisés : penser à purger régulièrement

---

La progression naturelle : par où commencer ?

Quand on débute avec Docker, il y a une progression logique à suivre. On ne commence pas à construire ses propres images dès le premier jour.

```
Niveau 1 │ docker pull + docker run
         │ → Utiliser des images existantes
         │
Niveau 2 │ docker-compose.yml
         │ → Orchestrer plusieurs services ensemble
         │
Niveau 3 │ Dockerfile + docker build
         │ → Créer ses propres images personnalisées
         │
Niveau 4 │ Dockerfile + Compose + .env + volumes + réseaux
         │ → Projet complet, propre, reproductible
```

Ne saute pas les étapes. Même les pros commencent par chercher si une image officielle existe avant d'en créer une.

Bonne pratique 1 — Toujours nommer ses conteneurs

Par défaut, Docker donne un nom aléatoire (comme `eloquent_turing` ou `funny_hopper`). C'est illisible.

```
# ❌ À éviter
docker run -d -p 8080:80 nginx

# ✅ Bonne pratique
docker run -d --name site-vitrine -p 8080:80 nginx
```

Avec un nom, les commandes suivantes deviennent lisibles :

```
docker logs site-vitrine
docker stop site-vitrine
docker exec -it site-vitrine bash
```

Bonne pratique 2 — Préciser la version de l'image

L'étiquette `latest` est pratique mais dangereuse : si l'image est mise à jour avec une version majeure, ton application peut se casser sans prévenir.

```
# ❌ Risqué en projet sérieux
docker run mysql

# ✅ Stable et prévisible
docker run mysql:8.0
docker run php:8.3-apache
docker run node:20-alpine
```

Consulte les **tags disponibles** sur hub.docker.com pour chaque image.

💡 **Exception** : En phase d'exploration / apprentissage, `latest` est acceptable. Mais dès qu'un projet est partagé avec d'autres ou mis en production, on fixe la version.

Bonne pratique 3 — Ne jamais mettre les mots de passe dans la commande

```
# ❌ Le mot de passe est visible dans l'historique du terminal
docker run -e MYSQL_ROOT_PASSWORD=monsecret mysql:8.0
```

```
3
4  # ✅ On utilise un fichier .env
5  docker run --env-file .env mysql:8.0
```

Le fichier `.env`

Crée un fichier nommé `.env` dans ton dossier de projet :

```
1  # .env
2  MYSQL_ROOT_PASSWORD=monsecret
3  MYSQL_DATABASE=monprojet
4  MYSQL_USER=dev
5  MYSQL_PASSWORD=devpass
```

Docker Compose charge automatiquement ce fichier s'il se trouve dans le même dossier que `docker-compose.yml`.

⚠️ **Important** : Ajoute `.env` dans ton `.gitignore`. On ne met jamais de mots de passe dans un dépôt Git.

```
1  # .gitignore
2  .env
```

Bonne pratique 4 — Un seul service par conteneur

Un conteneur ne doit faire qu'une seule chose. On ne met pas Apache + MySQL + PHP dans un seul conteneur comme on le ferait avec XAMPP.

```
# ❌ Mauvaise approche
Un conteneur "tout en un" avec Apache + MySQL + PHP

# ✅ Bonne approche
Conteneur 1 : PHP + Apache
Conteneur 2 : MySQL
Conteneur 3 : phpMyAdmin
```

Avantages :

- Chaque service peut être redémarré indépendamment
- On peut mettre à jour MySQL sans toucher à PHP
- On peut réutiliser le conteneur MySQL dans un autre projet

Bonne pratique 5 — Toujours utiliser des volumes pour les données

Un conteneur est **éphémère** : dès qu'on le supprime, toutes ses données disparaissent. Pour les bases de données, les fichiers uploadés, les logs importants, il faut utiliser un volume.

```
# ❌ Les données disparaissent si le conteneur est supprimé
docker run -d --name ma-db mysql:8.0

# ✅ Les données survivent au conteneur
docker run -d --name ma-db \
  -v mysql_data:/var/lib/mysql \
  mysql:8.0
```

La page suivante explique les volumes en détail.

Bonne pratique 6 — Utiliser `--restart unless-stopped` pour les services permanents

Si tu héberges un service qui doit toujours être disponible (un serveur web, une base de données), configure le redémarrage automatique.

```
docker run -d --name mon-site \
  --restart unless-stopped \
  -p 80:80 \
  nginx:1.25
```

Ainsi, si le serveur redémarre ou si Docker redémarre, le conteneur repart automatiquement.

La page suivante détaille toutes les politiques de redémarrage.

Bonne pratique 7 — Nettoyer régulièrement

Docker garde en cache toutes les images téléchargées et les conteneurs arrêtés. Ça peut vite occuper plusieurs dizaines de gigaoctets.

```
# Supprimer les conteneurs arrêtés
docker container prune

# Supprimer les images sans conteneur associé
docker image prune

# Supprimer les volumes non utilisés
docker volume prune

# Tout nettoyer d'un coup (images, conteneurs, réseaux inutilisés)
docker system prune

# Voir l'espace utilisé par Docker
docker system df
```

Le workflow typique d'un projet professionnel

Voici comment un développeur ou technicien aborde un nouveau projet Docker de A à Z.

Étape 1 — Chercher l'image officielle

Avant tout, on vérifie si une image officielle existe sur hub.docker.com.

```
docker search nginx
docker search mysql
```

On regarde les étoiles, si l'image est marquée "Official" et on note la version stable.

Étape 2 — Tester rapidement avec `docker run`

On lance un conteneur temporaire pour vérifier que ça fonctionne.

```
docker run --rm -d -p 8080:80 --name test-nginx nginx:1.25
# --rm = suppression automatique à l'arrêt
```

Étape 3 — Écrire le `docker-compose.yml`

Une fois que ça fonctionne, on formalise dans un fichier Compose. Plus besoin de retaper des longues commandes.

```
services:
  web:
    image: nginx:1.25
    container_name: mon-site
    restart: unless-stopped
    ports:
      - "80:80"
    volumes:
      - ./html:/usr/share/nginx/html
```

Étape 4 — Créer le `.env` et le `.gitignore`

```
# .env
APP_PORT=80
DB_PASSWORD=secret
```

```
# .gitignore
.env
```

Étape 5 — Versionner le projet

On commit le `docker-compose.yml`, le `Dockerfile` (si on en a un), le `.gitignore`, **mais pas le `.env`**. On fournit un `.env.example` à la place :

```
# .env.example — à copier en .env et à remplir
MYSQL_ROOT_PASSWORD=
MYSQL_DATABASE=
MYSQL_USER=
MYSQL_PASSWORD=
```

Étape 6 — Partager avec l'équipe

N'importe qui peut cloner le projet et démarrer l'environnement complet avec :

```
git clone https://github.com/mon-org/mon-projet
cd mon-projet
cp .env.example .env
# (remplir le .env)
docker compose up -d
```

C'est ça, la vraie puissance de Docker : **l'environnement est reproductible en quelques secondes sur n'importe quelle machine**.

Récapitulatif des commandes de démarrage rapide

Situation	Commande recommandée
Tester une image rapidement	`docker run --rm -p 8080:80 nginx:1.25`
Lancer un service permanent	`docker run -d --name X --restart unless-stopped ...`
Projet multi-services	`docker compose up -d`
Voir ce qui tourne	`docker ps`
Nettoyer	`docker system prune`
Voir l'espace utilisé	`docker system df`