

# Docker



docker  
Compose



# Alves Lobo Michael

- <https://www.linkedin.com/in/michael-alves-lobo>
- <https://github.com/kairrel/learning>
- 
- Devops, dev, réseau et administration système
- En poste chez Vade/Hornet
- Automatisation
  - ansible
  - puppet
  - terraform
  - docker



# SOMMAIRE

## Qu'est-ce que Docker ?

- **Conteneurisation** : emballer une application avec toutes ses dépendances
- **Différence VM vs Conteneurs** : léger, rapide, isolé
- **Pourquoi Docker ?**
  - Même environnement partout (dev, test, prod)
  - Installation rapide et propre
  - Isolation des applications



# SOMMAIRE

## Concepts de base

- **Image** : modèle immuable (comme un CD d'installation)
- **Conteneur** : instance d'exécution d'une image (comme un programme lancé)
- **Registry** : bibliothèque d'images (Docker Hub)

## Installation

- Installation de Docker Desktop (Windows/Mac) ou Docker (Linux)
- Vérification : `docker --version`
- Premier test : `docker run hello-world`



# SOMMAIRE

## Premiers pas avec Docker

- Commandes essentielles
- Exercice pratique 1 : Lancer un serveur web
- Exercice pratique 2 : Volumes (persistance des données)

## Créer ses propres images

- Le Dockerfile
- Construire et lancer l'image
- Exercice pratique 3 : Créer une image simple



# SOMMAIRE

## Introduction à Docker Compose

### Pourquoi Docker Compose ?

- Gérer plusieurs conteneurs facilement
- Configuration déclarative en YAML
- Réseau automatique entre conteneurs
- Parfait pour environnements de développement



# SOMMAIRE

## Introduction à Docker Compose




- Structure d'un fichier docker-compose.yml
- Commandes Docker Compose essentielles
- Bonnes pratiques
- Projet pratique complet
  - a. GLPI encore lui en 3 containers
    - i. un container mysql
    - ii. un container nginx
    - iii. un container avec php et l'application



# DOCKER

"Ça marche sur ma machine !"

3 colonnes avec icônes :

-  **Machine du dev** : Node 18, Python 3.9
-  **Serveur de test** : Node 16, Python 3.8
-  **Production** : Node 20, Python 3.10



**À dire :** "Combien d'entre vous ont déjà entendu ou dit cette phrase : 'Mais ça marche sur ma machine !' ? C'est LE problème classique du développement. Votre application fonctionne parfaitement sur votre ordinateur, mais plante en production. Pourquoi ? Parce que les environnements sont différents : versions de langages différentes, bibliothèques manquantes, configurations système différentes. Docker résout ce problème."





# DOCKER

## La solution traditionnelle : Les machines virtuelles

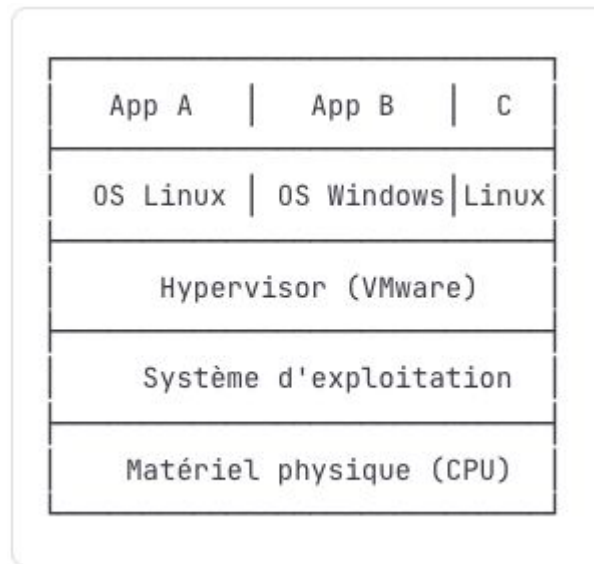
Machines Virtuelles (VMs)

Schéma en couches :

Points négatifs :

- ✗ Lourdes (plusieurs Go par VM)
- ✗ Lentes à démarrer (minutes)
- ✗ Consomment beaucoup de ressources

Avant Docker, la solution était les machines virtuelles. Chaque VM embarque un système d'exploitation complet. Si vous voulez 3 applications isolées, vous avez besoin de 3 OS complets ! C'est très lourd : une VM Windows pèse facilement 20 Go et met plusieurs minutes à démarrer. Sans compter que chaque VM consomme sa propre RAM et CPU."



# DOCKER

## La solution Docker : Les conteneurs

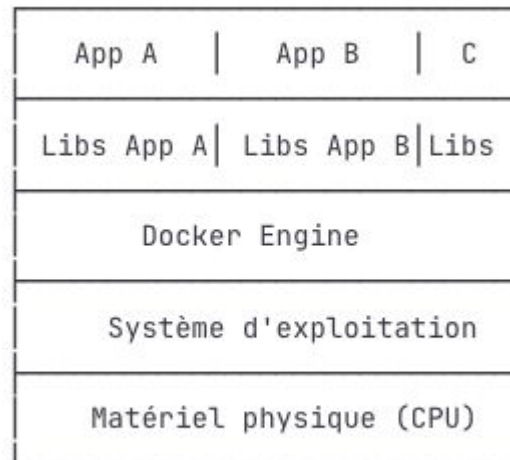
Conteneurs Docker

**Schéma en couches :**

**Points positifs :**

- ☒ Légers (quelques Mo)
- ☒ Rapides à démarrer (secondes)
- ☒ Partagent le noyau de l'OS hôte

Les conteneurs Docker, c'est différent. Au lieu d'avoir un OS complet par application, tous les conteneurs partagent le même noyau système. Chaque conteneur n'embarque que ce dont l'application a besoin : le code et ses dépendances. Résultat : un conteneur pèse quelques Mo et démarre en quelques secondes. Sur une même machine, vous pouvez lancer des dizaines de conteneurs sans problème."



# DOCKER

## Wikipédia definition:

**Docker** est un ensemble de [produits](#) de [plate-forme en tant que service](#) (PaaS) qui utilisent [la virtualisation](#) au [niveau du système d'exploitation](#) pour fournir des logiciels dans des paquets appelés [conteneurs](#).<sup>[5]</sup> Le service a à la fois des niveaux gratuits et premium. Le logiciel qui héberge les conteneurs s'appelle **Docker Engine**.<sup>[6]</sup> Il a été publié pour la première fois en 2013 et est développé par [Docker, Inc.](#)<sup>[7]</sup>




Docker est un outil qui est utilisé pour automatiser le déploiement [d'applications](#) dans des conteneurs légers afin que les applications puissent fonctionner efficacement dans différents environnements isolément.



# DOCKER

C'est quoi ?

**"Docker permet d'empaqueter une application et toutes ses dépendances dans un conteneur isolé et portable"**

-  **Empaqueter** : Code + dépendances + configuration
-  **Isoler** : Chaque conteneur est indépendant
-  **Porter** : Fonctionne partout (dev, test, prod)





"Docker, c'est simple : imaginez une boîte hermétique où vous mettez votre application avec tout ce dont elle a besoin pour fonctionner. Cette boîte, vous pouvez la déplacer n'importe où : sur votre PC, sur celui de votre collègue, sur un serveur de test, en production. Et elle fonctionnera toujours de la même manière. C'est exactement comme un conteneur maritime qui peut être transporté par bateau, camion ou train, d'où le logo de Docker."



# DOCKER

## Les avantages de Docker

Pourquoi utiliser Docker ?

1.  **Cohérence**
  - Même environnement partout
  - Fini "ça marche sur ma machine"
2.  **Rapidité**
  - Déploiement en secondes
  - Rollback instantané
3.  **Économie**
  - Meilleure utilisation des ressources
  - Moins de serveurs nécessaires
4.  **Simplicité**
  - Installation propre
  - Pas de conflits de dépendances





## Les avantages de Docker





"Les bénéfices sont immenses. Primo, cohérence : votre application fonctionne pareil partout, vous testez exactement ce qui sera en production. Secundo, rapidité : déployer une nouvelle version prend quelques secondes au lieu de minutes ou heures. Tertio, vous économisez des ressources serveur car les conteneurs sont légers. Et enfin, simplicité : besoin de PostgreSQL version 12 ET version 15 sur la même machine ? Aucun problème avec Docker, ils sont isolés. Sans Docker, c'est le cauchemar."



# DOCKER

## Cas d'usage concrets

Qui utilise Docker et pourquoi ?

-  **Développeurs**
  - Environnement de dev identique pour toute l'équipe
  - Installation rapide pour nouveaux arrivants
-  **Testeurs**
  - Tester plusieurs versions en parallèle
  - Environnement de test jetable
-  **DevOps**
  - Déploiement simplifié
  - Scaling horizontal facile
-  **Entreprises**
  - Netflix, Uber, Spotify, PayPal...
  - Gestion de milliers de microservices



# DOCKER

## Cas d'usage concrets

Docker est utilisé par tout le monde dans l'industrie. Les développeurs l'adorent : un nouveau membre rejoint l'équipe ? Au lieu de passer 2 jours à installer 15 outils et bibliothèques, il lance un conteneur et c'est prêt en 5 minutes. Les testeurs peuvent facilement tester sur différentes versions de bases de données. Et les DevOps déploient des applications en un clin d'œil. Des géants comme Netflix utilisent Docker pour gérer des milliers de microservices."





# DOCKER

## Les concepts fondamentaux (1/3)

Concept #1 : L'Image Docker

### Définition :

Une image est un **modèle immuable** qui contient tout le nécessaire pour exécuter une application

### Caractéristiques :

- 📀 Read-only (lecture seule)
- 📦 Contient : OS minimal + code + dépendances
- 🏗️ Construite en couches (layers)
- 📚 Stockée dans un registry (Docker Hub)

### Exemples :

- `nginx:latest` - serveur web
- `postgres:15` - base de données
- `node:18-alpine` - environnement Node.js



# DOCKER

## Les concepts fondamentaux (1/3)

Première notion importante : l'image. Pensez à un CD d'installation de Windows ou une recette de cuisine. C'est le modèle, le plan de construction. Une image Docker contient tout ce qu'il faut : un mini système d'exploitation, votre code, toutes les bibliothèques nécessaires. Elle est immuable, c'est-à-dire qu'on ne peut pas la modifier. Si vous voulez changer quelque chose, vous créez une nouvelle image. Les images sont stockées sur Docker Hub, un peu comme GitHub mais pour des images Docker. Il y a des milliers d'images officielles : nginx, postgres, mysql, python..."



# DOCKER

## Les concepts fondamentaux (2/3)






Concept #2 : Le Conteneur Docker

**Analogie visuelle :** Programme en cours d'exécution / Gâteau cuit à partir d'une recette

**Définition :**

Un conteneur est une **instance en cours d'exécution** d'une image

**Caractéristiques :**

-  Processus actif et isolé
-  Peut avoir son propre état modifiable
-  Créé à partir d'une image
-  Peut être démarré, arrêté, supprimé
-  Plusieurs conteneurs peuvent utiliser la même image



# DOCKER

## Les concepts fondamentaux (2/3)

"Le conteneur, c'est l'image qui tourne, qui est vivante. Si l'image c'est la recette, le conteneur c'est le gâteau que vous avez cuisiné. Si l'image c'est le CD de Windows, le conteneur c'est Windows en train de tourner sur votre PC. Un point crucial : à partir d'une seule image nginx, vous pouvez lancer 10, 50, 100 conteneurs différents. Chacun est isolé des autres. C'est comme avoir 100 serveurs web indépendants sur une seule machine."









## Les concepts fondamentaux (3/3)

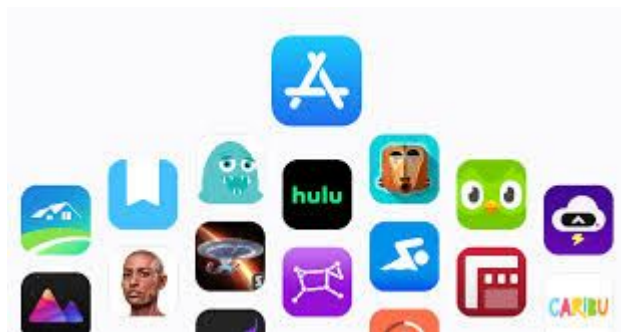
### Concept #3 : Docker Hub (Registry)

### Définition :

## Docker Hub est un **dépôt public** d'images Docker

## Chiffres clés :

-  Plus de 100 000 images publiques
-  Images officielles vérifiées
-  Images communautaires
-  Registries privés possibles





## Les concepts fondamentaux (3/3)

Docker Hub, c'est comme l'App Store ou le Play Store, mais pour des images Docker. Au lieu de télécharger chaque fois l'image depuis Internet, Docker la garde en local sur votre machine. La première fois vous faites un 'pull' pour télécharger, ensuite elle est dispo instantanément. Il y a des images officielles maintenues par Docker (nginx, ubuntu, postgres...) et des millions d'images créées par la communauté. Vous pouvez aussi créer vos propres images et les partager.



# DOCKER

## Récapitulatif des concepts

Image vs Conteneur vs Registry

Tableau comparatif :

| Concept      | Image                     | Conteneur               | Registry                 |
|--------------|---------------------------|-------------------------|--------------------------|
| C'est quoi ? | Modèle                    | Instance active         | Bibliothèque             |
| Analogie     | Recette                   | Plat cuisiné            | Livre de recettes        |
| État         | Immuable                  | Modifiable              | Stockage                 |
| Commande     | <code>docker build</code> | <code>docker run</code> | <code>docker pull</code> |

"Faisons un récap avant de passer à la pratique. L'image c'est le modèle, immuable, qu'on construit. Le conteneur c'est ce modèle qui tourne, qu'on peut démarrer et arrêter. Et le registry, c'est là où on stocke et partage les images. Retenez ces trois concepts, c'est la base de Docker. Des questions avant qu'on installe Docker ?"



# DOCKER

## Installation de Docker

Installer Docker

**3 options selon OS :**



**Windows /**



**macOS**

- Docker Desktop (interface graphique)
- Téléchargement : [docker.com](https://docker.com)



**Linux**

- Docker Engine (ligne de commande)
- Installation via gestionnaire de paquets :: C'est celle la qu'on va faire







## Installation de Docker

Vérification de l'installation :

```
docker --version
```

```
# Docker version 24.0.6
```

```
docker info
```

```
# Informations système
```





## Installation de Docker

TP1:: installer docker pour tout le monde sur du linux en ligne de commande :)



# DOCKER

## Premier test : Hello World

Notre premier conteneur !

**Commande :**

**docker run hello-world**

**Ce qui se passe :**

1. 🔍 Docker cherche l'image `hello-world` localement
2. ⬇ Ne la trouve pas → télécharge depuis Docker Hub
3. ✅ Lance un conteneur à partir de l'image
4. 📝 Le conteneur affiche un message et se termine



# DOCKER

**Sortie attendue :**

Hello from Docker!

This message shows that your installation  
appears to be working correctly.



# DOCKER

## PREMIERS PAS AVEC DOCKER

### Les commandes essentielles

#### IMAGES

- `docker pull` - Télécharger
- `docker images` - Lister
- `docker rmi` - Supprimer

#### CONTENEURS

- `docker run` - Créer et lancer
- `docker ps` - Lister (actifs)
- `docker stop` - Arrêter
- `docker start` - Démarrer
- `docker rm` - Supprimer



# DOCKER

## PREMIERS PAS AVEC DOCKER

### Les commandes essentielles



#### INSPECTION

- `docker logs` - Voir les logs
- `docker exec` - Exécuter une commande



#### NETTOYAGE

- `docker system prune` - Nettoyer

Maintenant, on va manipuler Docker. Il y a 4 types de commandes : celles pour gérer les images, celles pour les conteneurs, celles pour inspecter ce qui se passe, et celles pour nettoyer. Pas de panique, on va les voir une par une avec des exemples.



# DOCKER

## Commande : docker pull

Télécharger une image

### Syntaxe :

```
docker pull [nom_image]:[tag]
```

### Exemples :

*# Télécharger nginx (dernière version)*

```
docker pull nginx
```

*# Télécharger nginx version spécifique*

```
docker pull nginx:1.25
```

*# Télécharger Alpine Linux (OS minimal)*

```
docker pull alpine
```



# DOCKER

## 💡 Tags courants :

- `latest` - dernière version (par défaut)
- `1.25`, `15.3` - version spécifique
- `alpine` - version minimaliste

"Pour télécharger une image depuis Docker Hub, on utilise 'docker pull'. Par défaut, Docker télécharge la version 'latest', la plus récente. Mais vous pouvez spécifier une version précise avec un tag. C'est important en production d'utiliser des versions spécifiques pour éviter les surprises lors des mises à jour. Essayons : 'docker pull nginx'. Vous voyez les layers qui se téléchargent ? Les images sont construites en couches, c'est très optimisé."

## [📌] EXERCICE : Télécharger Tous NGINX





# DOCKER

## Commande : docker images

Lister les images locales

`docker images`

Colonnes importantes :

- **REPOSITORY** : nom de l'image
- **TAG** : version
- **SIZE** : taille de l'image
- **IMAGE ID** : identifiant unique

Sortie exemple :

| REPOSITORY  | TAG    | IMAGE ID     | CREATED      | SIZE   |
|-------------|--------|--------------|--------------|--------|
| nginx       | latest | 605c77e624dd | 2 weeks ago  | 141MB  |
| hello-world | latest | feb5d9fea6a5 | 6 months ago | 13.3kB |
| alpine      | latest | c1aabb73d233 | 7 weeks ago  | 7.33MB |



# DOCKER

Pour voir toutes les images que vous avez sur votre machine, utilisez 'docker images'. Vous voyez nginx qu'on vient de télécharger, elle fait 141 Mo. Comparez avec hello-world : 13 Ko ! Et Alpine Linux, un OS complet, seulement 7 Mo. C'est ça la magie des conteneurs.

**[ II ] EXERCICE : Tout le monde liste ses images**



# DOCKER

## Commande : docker run (basique)

Lancer un conteneur

### Syntaxe basique :

```
docker run [options] [image] [commande]
```

### Exemple simple :

*# Lancer nginx en arrière-plan*

```
docker run -d nginx
```

*# Lancer Alpine et exécuter une commande*

```
docker run alpine echo "Hello Docker"
```



# DOCKER

## Commande : docker run (basique)

### Options courantes :

- `-d` : mode détaché (arrière-plan)
- `--name` : donner un nom au conteneur
- `-p` : mapper un port
- `-v` : monter un volume

La commande magique : `docker run`. C'est elle qui crée et démarre un conteneur. L'option `'-d'` lance le conteneur en arrière-plan, sinon il occupe votre terminal. Sans cette option, vous verriez tous les logs défiler. Testons avec Alpine : `'docker run alpine echo Hello Docker'`. Le conteneur démarre, affiche le message, et s'arrête immédiatement. C'est normal ! Un conteneur vit tant que son processus principal tourne



# DOCKER

## Commande : docker ps

Lister les conteneurs

### Commandes :

*# Conteneurs actifs seulement*

`docker ps`

*# TOUS les conteneurs (actifs + arrêtés)*

`docker ps -a`

### Sortie exemple :

| CONTAINER ID | IMAGE | COMMAND                 | STATUS       |
|--------------|-------|-------------------------|--------------|
| a3f2bc1d34e5 | nginx | "/docker-entrypoint..." | Up 2 minutes |



# DOCKER

## Commande : docker ps

### Informations importantes :

- **CONTAINER ID** : identifiant unique
- **IMAGE** : image utilisée
- **STATUS** : état (Up / Exited)
- **PORTS** : ports exposés

Pour voir vos conteneurs actifs, utilisez 'docker ps'. Vous ne voyez que les conteneurs qui tournent actuellement. Pour voir TOUS les conteneurs, même ceux qui sont arrêtés, ajoutez '-a'. C'est important car un conteneur arrêté reste sur votre système, il prend de la place. On verra comment nettoyer ça plus tard.



# DOCKER

## Commande : docker run (avec options)

Lancer un serveur web accessible

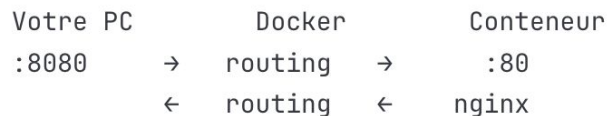
**Commande complète :**

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

**Décomposition :**

- `docker run` : lancer un conteneur
- `-d` : en arrière-plan (detached)
- `-p 8080:80` : port local:port conteneur
- `--name mon-nginx` : nom personnalisé
- `nginx` : image à utiliser

**Schéma de port mapping :**



# DOCKER

## Commande : docker run (avec options)

**Test :** Ouvrir <http://localhost:8080>

Maintenant on lance un vrai serveur web ! Cette commande est plus complexe. '-p 8080:80' est crucial : ça dit 'relie le port 8080 de mon PC au port 80 du conteneur'. Nginx écoute sur le port 80 à l'intérieur du conteneur, mais on y accède via le port 8080 de notre machine. C'est comme une redirection. Lancez la commande, puis ouvrez votre navigateur sur localhost:8080. Vous devriez voir la page d'accueil de nginx. Incroyable non ? Un serveur web en une seule ligne de commande !

[  EXERCICE PRATIQUE : Tout le monde lance nginx]





# DOCKER

## Commande : docker stop & start

Arrêter et redémarrer un conteneur

### Commandes :

*# Arrêter un conteneur*

`docker stop mon-nginx`

*# Redémarrer un conteneur arrêté*

`docker start mon-nginx`

*# Redémarrer (stop + start)*

`docker restart mon-nginx`



# DOCKER

## Commande : docker stop & start

💡 **Bon à savoir :**

- Un conteneur arrêté reste en mémoire
- Ses données sont préservées
- On peut le redémarrer à tout moment
- Utilisez le nom OU l'ID du conteneur

Pour arrêter votre conteneur, 'docker stop' suivi du nom ou de l'ID. Le conteneur s'arrête proprement, il a le temps de finir ses tâches en cours. Vérifiez : rafraîchissez localhost:8080, ça ne répond plus. Mais le conteneur existe toujours, il est juste arrêté. Pour le relancer : 'docker start mon-nginx'. Magie, le site est de retour ! C'est très pratique pour gérer vos services.

[📌] **EXERCICE : Arrêter et redémarrer nginx**



# DOCKER

## Commande : docker logs

Voir les logs d'un conteneur

### Commandes :

*# Afficher les logs*

`docker logs mon-nginx`

*# Suivre les logs en temps réel*

`docker logs -f mon-nginx`

*# 100 dernières lignes*




`docker logs --tail 100 mon-nginx`



# DOCKER

## Commande : docker logs

### Utilité :

-  Débuguer des problèmes
-  Voir les requêtes HTTP
-  Identifier les erreurs

Les logs sont essentiels pour comprendre ce qui se passe dans votre conteneur. Tapez 'docker logs mon-nginx'. Vous voyez toutes les requêtes HTTP que nginx a reçues. Avec '-f', vous suivez les logs en temps réel, comme un 'tail -f'. Rafraîchissez votre navigateur et regardez les nouvelles lignes apparaître. C'est votre meilleur ami pour déboguer !

[  DÉMONSTRATION avec refresh du navigateur ]



# DOCKER

## Commande : docker exec

Exécuter une commande dans un conteneur

### Syntaxe :

```
docker exec [options] [conteneur] [commande]
```

### Exemples :

*# Ouvrir un terminal interactif*

```
docker exec -it mon-nginx bash
```

*# Exécuter une commande simple*

```
docker exec mon-nginx ls /usr/share/nginx/html
```

*# Voir les processus*

```
docker exec mon-nginx ps aux
```



# DOCKER

## Commande : docker exec

### Options :

- `-i` : mode interactif
- `-t` : allouer un pseudo-terminal
- `-it` : combinaison (shell interactif)

Docker exec permet d'entrer dans un conteneur qui tourne. C'est comme se connecter en SSH à un serveur, mais en instantané. La commande magique : `'docker exec -it mon-nginx bash'`. Vous êtes maintenant À L'INTÉRIEUR du conteneur ! Tapez `'ls'`, `'pwd'`, explorez. Vous verrez les fichiers de nginx. Pour sortir, tapez `'exit'`. C'est super utile pour déboguer ou modifier des fichiers temporairement.

 **EXERCICE : Entrer dans le conteneur et explorer]**



# DOCKER

## Commande : docker rm & docker rmi

Supprimer conteneurs et images

### Supprimer des conteneurs :

`docker rm mon-nginx` # *Supprimer un conteneur arrêté*

`docker rm -f mon-nginx` # *Supprimer un conteneur actif (force)*

`docker container prune` # *Supprimer tous les conteneurs arrêtés*

`docker rmi nginx` # *Supprimer une image*

`docker image prune` # *Supprimer images non utilisées*



# DOCKER

## Commande : `docker rm` & `docker rmi`

⚠ **Important** : Un conteneur doit être arrêté avant suppression (ou utiliser `-f`)

Pour supprimer un conteneur, '`docker rm`'. Attention, il doit être arrêté d'abord. Si vous voulez forcer la suppression d'un conteneur actif, ajoutez '`-f`', mais c'est brutal, évitez en production. Pour supprimer une image, '`docker rmi`'. Docker refuse si un conteneur utilise encore cette image, c'est une sécurité. Et 'prune' est votre ami pour nettoyer : il supprime tout ce qui n'est pas utilisé.





# DOCKER

## Exercice pratique

**Titre :** 🎯 Exercice : Lancer un serveur web

**Mission :**

1. Télécharger l'image `nginx`
2. Lancer un conteneur nommé `web-server`
3. Le rendre accessible sur le port `8080`
4. Vérifier dans le navigateur
5. Voir les logs
6. Arrêter le conteneur
7. Le supprimer



# DOCKER COMPOSE



# DOCKER COMPOSE

Gérer plusieurs conteneurs : le cauchemar

**Scénario : Application web classique**



bash



*# 1. Lancer la base de données*

```
docker run -d --name db \  
  -e POSTGRES_PASSWORD=secret \  
  -v db-data:/var/lib/postgresql/data \  
  postgres:15
```

*# 2. Lancer le backend*

```
docker run -d --name api \  
  -p 3000:3000 \  
  -e DATABASE_URL=postgresql://db:5432 \  
  --link db \  
  mon-api:latest
```

*# 3. Lancer le frontend*

```
docker run -d --name web \  
  -p 80:80 \  
  --link api \  
  mon-frontend:latest
```

# DOCKER COMPOSE

Gérer plusieurs conteneurs : le cauchemar

## 🤖 Problèmes :

- 3 commandes longues et complexes
- Ordre de démarrage important
- Difficile à documenter/partager
- Erreur = recommencer tout

Regardez cette horreur ! Pour une simple application avec 3 services, vous devez taper 3 commandes énormes. Et attention à l'ordre : si vous lancez l'API avant la base de données, ça plante. Si un nouveau développeur rejoint le projet, il doit copier ces 3 commandes, les comprendre, les adapter... C'est ingérable. Et si vous avez 10 services ? 20 ? Imaginez le cauchemar !



# DOCKER COMPOSE

## La solution : Docker Compose

Docker Compose : l'orchestrateur de conteneurs

Même application avec Compose :

Fichier : `docker-compose.yml`



# DOCKER COMPOSE

## La solution : Docker Compose

Docker Compose : l'orchestrateur de conteneurs

Une seule commande

`docker-compose up`



yaml



```
services:
  db:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data

  api:
    build: ./api
    ports:
      - "3000:3000"
    depends_on:
      - db

  web:
    build: ./frontend
    ports:
      - "80:80"
    depends_on:
      - api

volumes:
  db-data:
```

# DOCKER COMPOSE

## La solution : Docker Compose

### ✨ Avantages :

- Configuration déclarative (YAML)
- Une seule commande
- Réseau automatique entre services
- Ordre de démarrage géré
- Facile à versionner (Git)

Voici Docker Compose ! Au lieu de commandes interminables, vous écrivez un fichier YAML qui décrit votre infrastructure. Vous y définissez tous vos services, leurs relations, leurs configurations. Et ensuite ? Une seule commande : 'docker-compose up'. C'est tout ! Docker Compose lance tout dans le bon ordre, crée le réseau entre les services, gère les volumes... C'est de la magie. Et ce fichier YAML, vous le committez dans Git, donc toute l'équipe a exactement la même configuration.



# DOCKER COMPOSE

## Qu'est-ce que Docker Compose ?

Docker Compose en détail

### Définition :

Docker Compose est un outil pour définir et exécuter des applications Docker multi-conteneurs

### Concepts clés :

#### Fichier de configuration

- `docker-compose.yml` (YAML)
- Définit tous les services
- Versionnable avec Git





# DOCKER COMPOSE

## Qu'est-ce que Docker Compose ?

Concepts clés :



### Services

- Chaque conteneur = un service
- Nomme et configure chaque partie



### Réseau automatique

- Tous les services peuvent communiquer
- Par leur nom de service



### Volumes partagés

- Persistance des données
- Partage entre services



# DOCKER COMPOSE

## Qu'est-ce que Docker Compose ?

"Docker Compose se base sur un fichier YAML où vous décrivez votre infrastructure complète. Chaque conteneur devient un 'service' avec un nom. L'énorme avantage : Docker Compose crée automatiquement un réseau privé où tous vos services peuvent communiquer entre eux en utilisant simplement leur nom. Plus besoin de --link ou d'IP complexes. L'API veut parler à la base de données ? Elle utilise simplement 'db' comme hostname. C'est génial !



# DOCKER COMPOSE

## Installation de Docker Compose

Docker Compose : déjà installé !



**Bonne nouvelle :**

**Docker Desktop (Windows/Mac) :**

- ☒ Compose inclus automatiquement
- Rien à installer !

**Docker Engine (Linux) :**

- ☒ Souvent inclus
- Sinon : `apt install docker-compose-plugin`



# DOCKER COMPOSE

## Installation de Docker Compose

Vérification :

`docker compose version`

Excellente nouvelle : si vous avez Docker Desktop, vous avez déjà Docker Compose ! Sur Linux, vérifiez avec 'docker-compose --version'. Vous verrez qu'il existe deux syntaxes : 'docker-compose' avec un tiret (l'ancienne) et 'docker compose' sans tiret (la nouvelle, c'est un plugin). Les deux fonctionnent, mais Docker recommande la nouvelle. Dans ce cours, j'utilise l'ancienne car elle est plus répandue, mais vous pouvez utiliser l'une ou l'autre.



## Structure d'un fichier

"Décortiquons la structure. En haut, la version du format Compose, mettez '3.8' ou '3.9', c'est stable. Ensuite, la section 'services' : c'est là que vous listez tous vos conteneurs. Chaque service a un nom que vous choisissez librement. Pour chaque service, vous définissez l'image, les ports, les variables d'environnement, etc. C'est exactement les mêmes options qu'avec 'docker run', mais en YAML au lieu de ligne de commande. En bas, vous pouvez déclarer des volumes et réseaux réutilisables."

[illegible]

# DOCKER COMPOSE

## Exemple simple : Nginx

Titre : Premier docker-compose.yml

Fichier : **docker-compose.yml**

version: '3.8'

services:

web:

image: nginx:alpine

ports:

- "8080:80"

volumes:

- ./html:/usr/share/nginx/html



# DOCKER COMPOSE

## Exemple simple : Nginx

**Titre :** Premier docker-compose.yml

**mon-projet/**

└─ **docker-compose.yml**

└─ **html/**

└─ **index.html**

`docker-compose up`



# DOCKER COMPOSE

**À dire :** "Commençons simple. Ce fichier compose lance un serveur nginx. Créez un dossier 'mon-projet', mettez-y ce fichier compose et créez un sous-dossier 'html' avec un index.html dedans. C'est exactement ce qu'on faisait ce matin avec 'docker run', mais maintenant c'est documenté dans un fichier. Tapez 'docker-compose up' dans le terminal. Boom ! Nginx démarre et sert vos fichiers HTML. Pour arrêter, Ctrl+C ou 'docker-compose down'."

**[📌 EXERCICE RAPIDE : Créer ce premier compose]**





# DOCKER COMPOSE

## Les sections principales (1/4)

Section : services

Définition des services :

yaml

```
services:
  mon-service:
    # Utiliser une image existante
    image: nginx:latest

    # OU construire depuis un Dockerfile
    build:
      context: ./mon-app
      dockerfile: Dockerfile

    # Nom du conteneur (optionnel)
    container_name: mon-nginx

    # Commande à exécuter
    command: npm start

    # Répertoire de travail
    working_dir: /app
```



# DOCKER COMPOSE

## Les sections principales (1/4)

Section : services

💡 **Choix important :**

- **image** : pour images Docker Hub
- **build** : pour vos propres applications

La section services, c'est le cœur. Pour chaque service, vous choisissez : soit vous utilisez une image toute faite de Docker Hub avec 'image', soit vous construisez votre propre image avec 'build' en pointant vers un dossier qui contient un Dockerfile. Les deux approches sont valides. En général : images officielles pour les bases de données, nginx, redis... Et build pour votre propre code applicatif.



# DOCKER COMPOSE

## Les sections principales (2/4)

Section : ports et environment

**Ports (exposition) :**

services:

api:

image: node:18

ports:

- "3000:3000"      # *host:container*
- "8080:80"      # *différents ports*
- "127.0.0.1:5000:5000" # *bind sur localhost uniquement*



# DOCKER COMPOSE

## Les sections principales (2/4)

Variables d'environnement :

"Les ports, même principe que 'docker run -p' : port de votre machine vers port du conteneur. Les variables d'environnement : trois façons de les définir. Directement dans le YAML, en liste avec '=', ou depuis un fichier .env. Cette dernière méthode est la meilleure pour les secrets : vous ne committez pas le .env dans Git, chacun a son propre fichier local avec ses credentials."



yaml



```
services:
  api:
    environment:
      # Format clé: valeur
      NODE_ENV: production
      DATABASE_URL: postgresql://user:pass@db:5432/mydb

      # OU format liste
    environment:
      - NODE_ENV=production
      - API_KEY=secret123

      # OU fichier externe
    env_file:
      - .env
```

# DOCKER COMPOSE

## Les sections principales (3/4)

**Titre :** Section : volumes et depends\_on

**Volumes :**

yaml

```
services:
  db:
    image: postgres:15
    volumes:
      # Volume nommé (géré par Docker)
      - postgres-data:/var/lib/postgresql/data

      # Bind mount (dossier local)
      - ./config:/etc/postgresql

      # Volume anonyme
      - /var/lib/postgresql/data

# Déclaration des volumes nommés
volumes:
  postgres-data:
```



# DOCKER COMPOSE

## Les sections principales (3/4)

**Titre :** Section : volumes et depends\_on

### Volumes :

Les volumes nommés pour les données importantes (bases de données), bind mounts pour partager du code en développement. La clé 'depends\_on' est super importante : elle dit 'ne démarre pas ce service avant que tel autre soit démarré'. Attention : 'démarré' ne veut pas dire 'prêt'. Si votre API se connecte à PostgreSQL, elle peut démarrer avant que Postgres soit vraiment prêt à accepter des connexions. Il faut gérer ça dans votre code."

yaml



```
services:
```

```
  web:
```

```
    depends_on:
```

```
      - api      # Attendre que api soit lancée
```

```
  api:
```

```
    depends_on:
```

```
      - db      # Attendre que db soit lancée
```



# DOCKER COMPOSE

## Les sections principales (4/4)

Section : networks

### Réseau par défaut :

- Docker Compose crée automatiquement un réseau
- Tous les services peuvent communiquer
- Par nom de service (DNS automatique)



# DOCKER COMPOSE

## Les sections principales (4/4)

Section : networks

yaml

```
services:
  frontend:
    networks:
      - frontend-net

  backend:
    networks:
      - frontend-net
      - backend-net

  db:
    networks:
      - backend-net      # Isolée du frontend

networks:
  frontend-net:
  backend-net:
```





# DOCKER COMPOSE

## Les sections principales (4/4)

Section : networks

💡 **Avantage** : Isolation et sécurité

Par défaut, tous les services d'un compose peuvent se parler. Mais vous pouvez créer plusieurs réseaux pour isoler vos services. Exemple : votre frontend ne devrait pas pouvoir parler directement à la base de données, seulement au backend. Vous créez deux réseaux : frontend-net pour web↔api, et backend-net pour api↔db. La base de données n'est accessible que par l'API. C'est une bonne pratique de sécurité.



# DOCKER COMPOSE

## Commandes Docker Compose (1/2)

Commandes essentielles

Démarrage et arrêt :

```
bash
```

```
# Démarrer tous les services (logs visibles)
```

```
docker-compose up
```

```
# Démarrer en arrière-plan
```

```
docker-compose up -d
```

```
# Arrêter et supprimer les conteneurs
```

```
docker-compose down
```

```
# Arrêter ET supprimer volumes
```

```
docker-compose down -v
```

### Construction :

```
bash
```

```
# Construire les images
```

```
docker-compose build
```

```
# Construire sans cache
```

```
docker-compose build --no-cache
```

```
# Up avec rebuild
```

```
docker-compose up --build
```



# DOCKER COMPOSE

## Commandes Docker Compose (1/2)

Monitoring et débogage

### Voir l'état :

```
bash

# Lister les services actifs
docker-compose ps

# Voir les logs de tous les services
docker-compose logs

# Suivre les logs en temps réel
docker-compose logs -f

# Logs d'un service spécifique
docker-compose logs -f api
```

### Exécution de commandes :

```
bash

# Exécuter une commande dans un service
docker-compose exec api bash

# Exécuter sans terminal interactif
docker-compose exec -T api npm test

# Redémarrer un service
docker-compose restart api
```



# DOCKER COMPOSE

## Commandes Docker Compose (1/2)

### Monitoring et débogage

Pour le monitoring : 'ps' liste vos services, 'logs' affiche les logs. Avec '-f', vous suivez en temps réel. Vous pouvez filtrer par service, super pratique quand vous avez 10 services et que vous voulez juste voir les logs de votre API. 'exec' pour entrer dans un conteneur, exactement comme 'docker exec'. Et petit bonus : vous pouvez scaler horizontalement avec '--scale'. Trois instances de votre API ? Pas de problème.



# DOCKER COMPOSE

## Exercice pratique : Encore Lui GLPI

🎯 Exercice : Nginx + Base de données + Application GLPI + Php-fm

**Mission : Vous avez déjà installé GLPI version serveur, ici on va faire pareil version docker**

### Objectifs :

1. Service **web** : nginx sur port 8080
2. Service **db** : postgres:15
3. Monter **./html** dans nginx
4. Variables d'env pour postgres
5. Volume pour persister les données DB

