etna CO-LABS

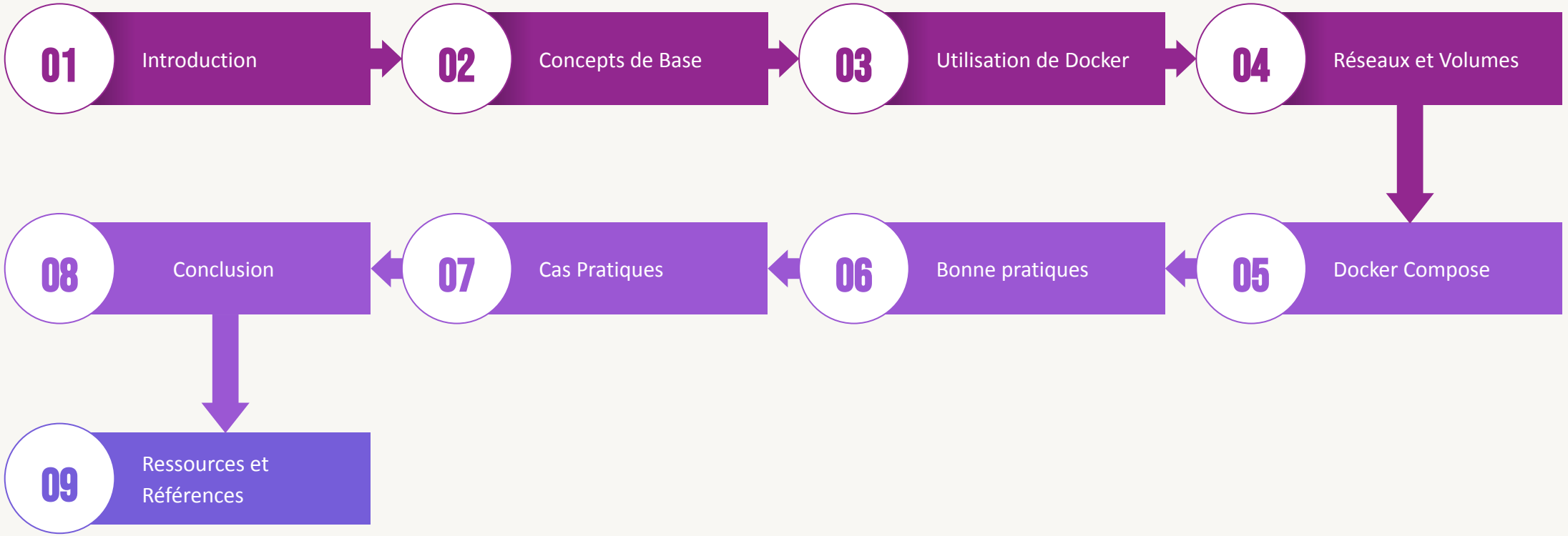
DOCKER INTRODUCTION

COLABS - ETNA – IONIS EDUCATION GROUP

Kévin ANSARD

Ivry-sur-seine, 11 juin 2024

SOMMAIRE



INTRODUCTION

C'EST QUOI DOCKER ?

Un conteneur est une unité standardisée de logiciel qui empaquette le code et toutes ses dépendances pour que l'application fonctionne rapidement et de manière fiable d'un environnement informatique à un autre. Les conteneurs permettent l'isolation des processus et la gestion efficace des ressources, en partageant le noyau du système d'exploitation de l'hôte mais en isolant les applications au niveau des processus et des ressources.

C'EST QUOI UN CONTENEUR ?

Docker est une plateforme open-source qui utilise la technologie de conteneurisation pour automatiser le déploiement, la mise à l'échelle et l'exécution des applications dans des conteneurs, garantissant ainsi portabilité, efficacité et cohérence entre différents environnements.

DIFFÉRENCE ENTRE UNE VM ET UN CONTENEUR

CONTENEUR

Partagent le noyau du système d'exploitation de l'hôte, mais fonctionnent comme des processus isolés

Offrent une isolation au niveau des processus et des ressources, mais partagent le même noyau de l'hôte.

Sont plus légers et démarrent en quelques secondes.

Sont extrêmement portables entre différents environnements (développement, test, production) sans changement de configuration.

MACHINE VIRTUELLE

Incluent un système d'exploitation complet, elles possèdent leur propre noyau.

Fournissent une isolation complète de l'OS.

Prennent plus de temps à démarrer car elles doivent initialiser un OS complet.

Sont également portables mais nécessitent souvent plus de configurations spécifiques à l'environnement.

AVANTAGES DE DOCKER

PORTABILITÉ	EFFICACITÉ	RAPIDITÉ
ISOLATION ET SÉCURITÉ	GESTION DES DÉPENDANCES	SCALABILITÉ

CONCEPTS DE BASES

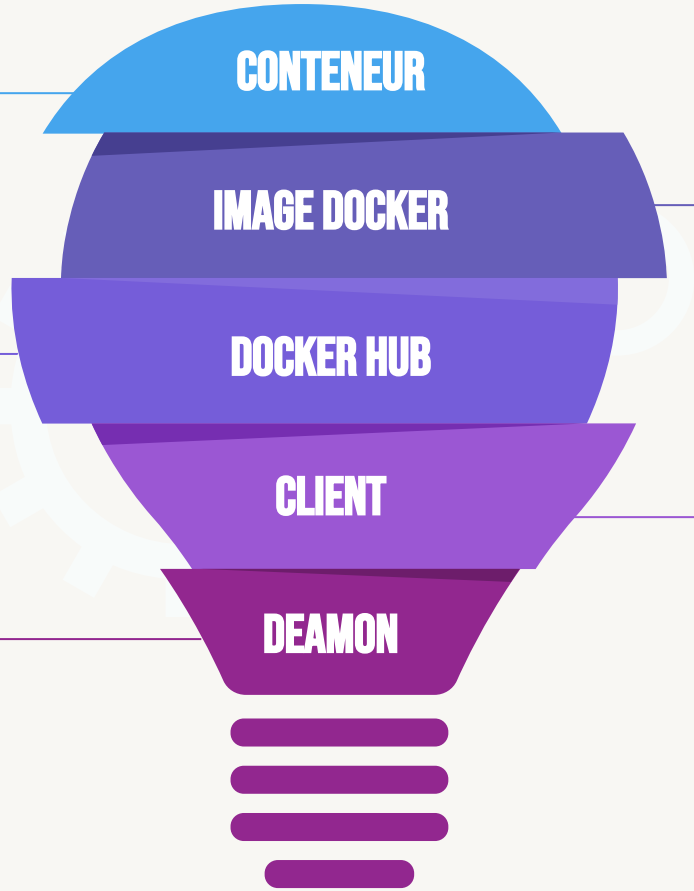
TOUT SAVOIR SUR DOCKER

ARCHITECTURE DE DOCKER

Les Docker Containers sont des **packages logiciels légers**, **autonomes** et **exécutables** qui comprennent tout ce qui est nécessaire pour **exécuter une application** : le code, le runtime, les outils système, les bibliothèques et les paramètres.

Docker Hub est un **service de registre** basé sur le cloud où les utilisateurs de Docker peuvent **créer**, **stocker** et **distribuer** des **images** Docker.

Le Docker Daemon (`dockerd`) est un service d'arrière-plan qui s'exécute sur le système d'exploitation hôte. Il est **responsable** de la **gestion des images** Docker, des **conteneurs**, des **réseaux** et des **volumes** de **stockage**.



Les Docker Images sont des modèles en lecture seule **utilisés pour créer** des **conteneurs** Docker. Elles **contiennent l'application** et ses **dépendances**, ainsi que les **paramètres de configuration**.

Le Docker Client (`docker`) est une interface en **ligne de commande** (CLI) qui permet aux utilisateurs **d'interagir** avec le **Docker Daemon**.

DIFFÉRENCE ENTRE UNE IMAGE ET UN CONTENEUR

IMAGE

Modèle statique et immuable.

Utilisé pour créer des conteneurs.

Construit à partir de Dockerfile.

Contient toutes les dépendances nécessaires à l'application.

CONTENEUR

Instance en cours d'exécution et mutable.

Créé à partir d'une image.

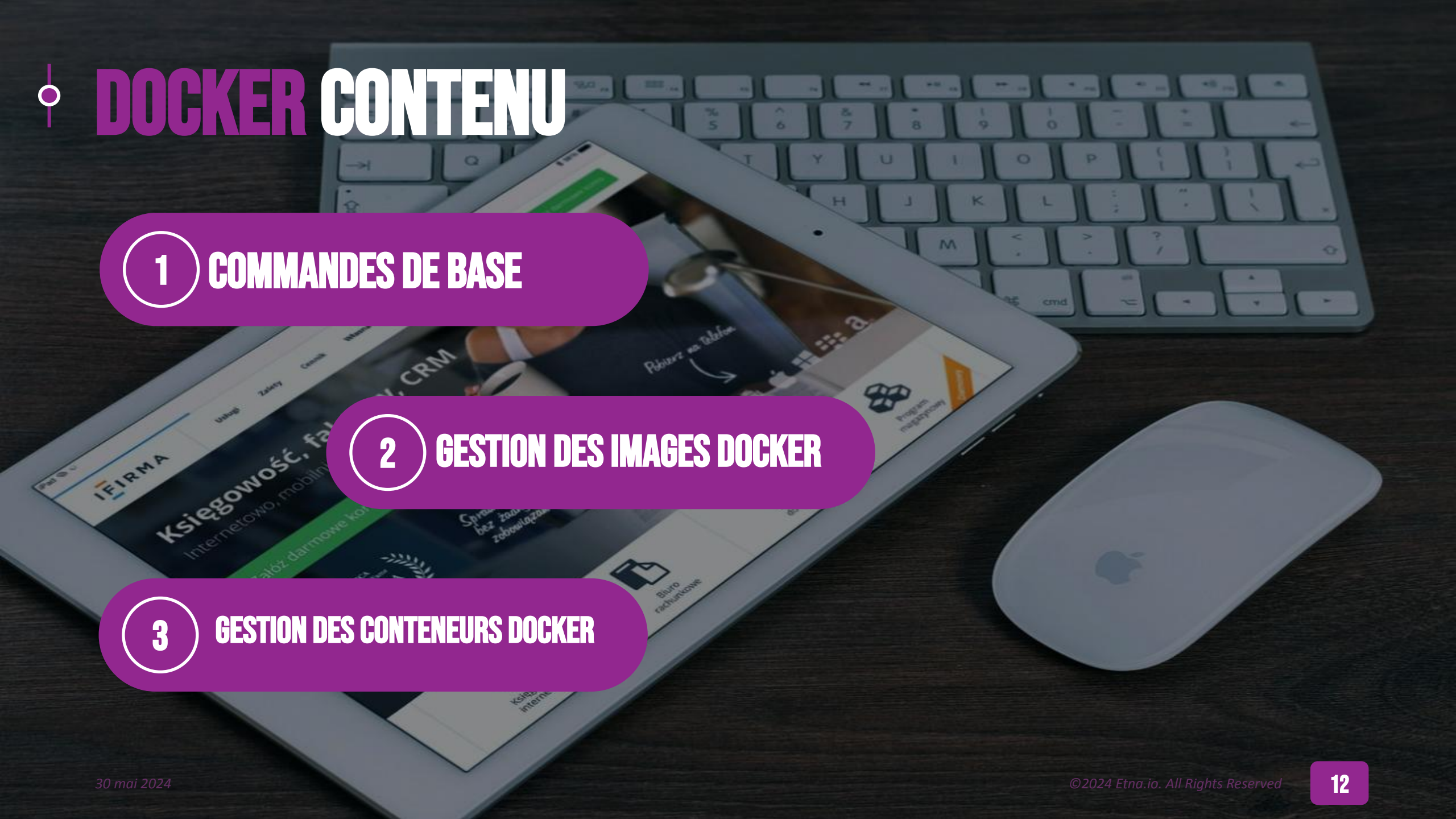
Peut avoir son propre état et modifications pendant l'exécution.

Utilisé pour exécuter des applications dans un environnement isolé.

UTILISATION DE DOCKER

LE GLOSSAIRE

DOCKER CONTENU



1. COMMANDES DE BASE
2. GESTION DES IMAGES DOCKER
3. GESTION DES CONTENEURS DOCKER

OVERVIEW DES COMMANDES DOCKER

DOCKER RUN

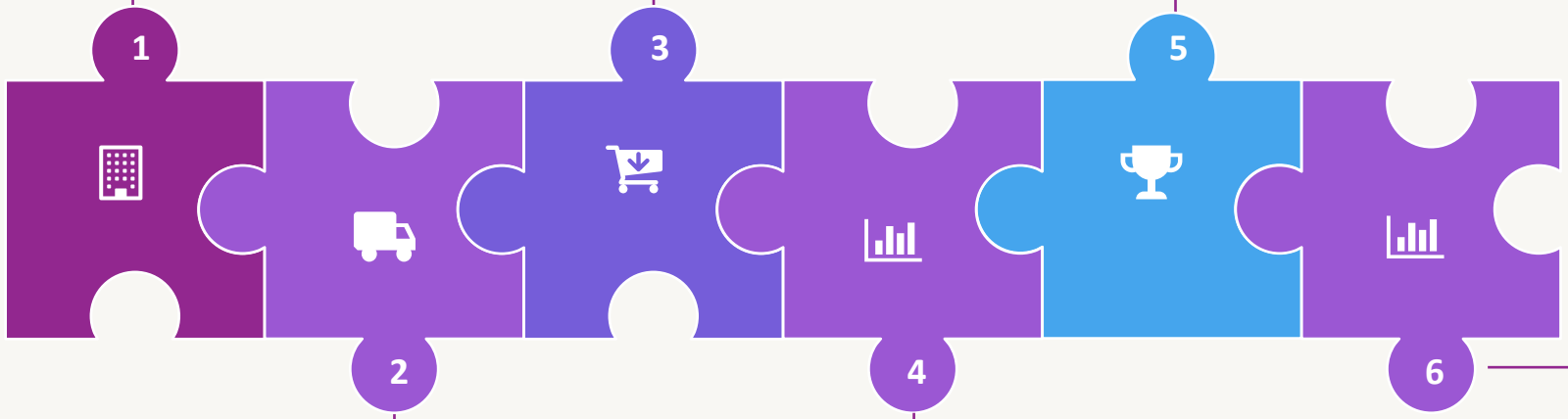
Créer et démarrer un nouveau conteneur à partir d'une image spécifiée

DOCKER STOP

arrête un conteneur en cours d'exécution.

DOCKER IMAGES

liste toutes les images Docker sur l'hôte.



DOCKER RMI

Cette commande supprime une ou plusieurs images Docker

DOCKER PS

iste les conteneurs actuellement en cours d'exécution.

DOCKER RM

supprime un ou plusieurs conteneurs arrêtés.

1 - DOCKER RUN

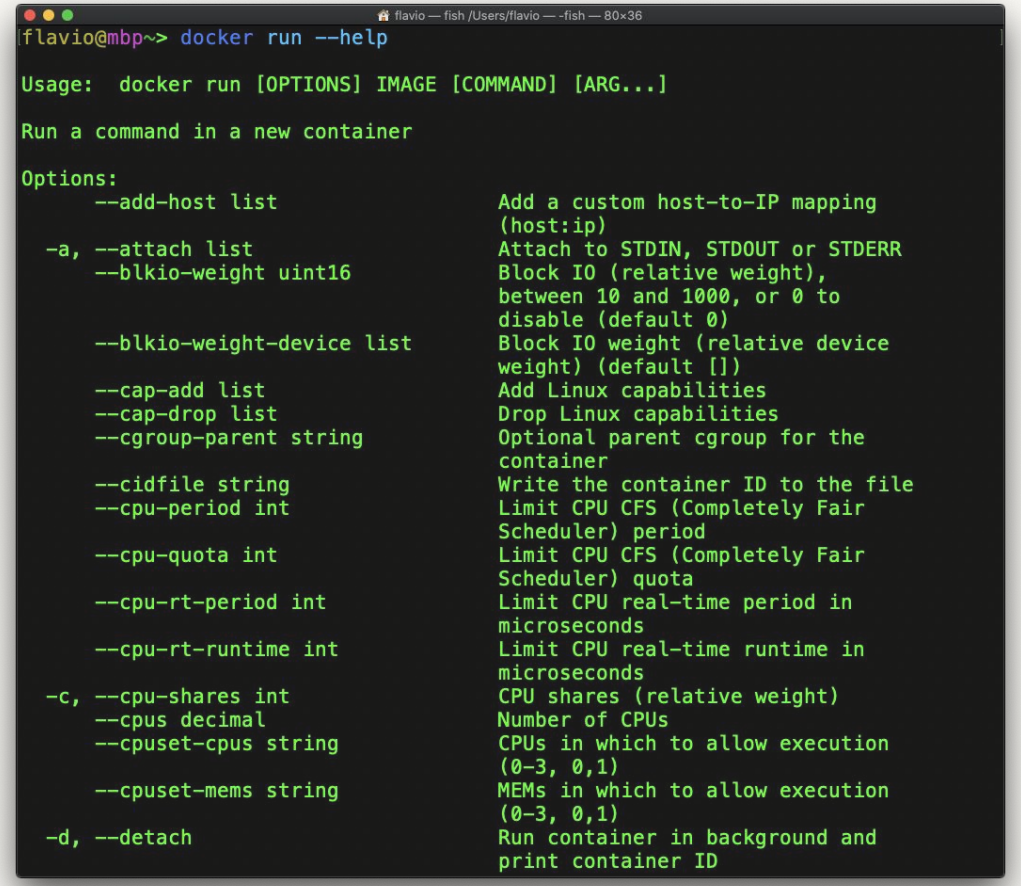
INFORMATIONS :

Description :

Cette commande est utilisée pour créer et démarrer un nouveau conteneur à partir d'une image spécifiée.

Utilisation :
`docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`

Exemple :
`docker run -d -p 80:80 nginx` (Cela exécute un serveur web Nginx en mode détaché et mappe le port 80 de l'hôte au port 80 du conteneur.)



2 - DOCKER PS

INFORMATIONS :

Description :

Cette commande liste les conteneurs actuellement en cours d'exécution.

Utilisation :
`docker ps [OPTIONS]`

Exemple :
`docker ps` (Cela liste tous les conteneurs en cours d'exécution. Pour lister tous les conteneurs, y compris ceux arrêtés, utilisez `docker ps -a`.)

```
> docker ps --help

Usage:  docker ps [OPTIONS]

List containers

Aliases:
  docker container ls, docker container list, docker container ps, docker ps

Options:
  -a, --all             Show all containers (default shows just running)
  -f, --filter filter   Filter output based on conditions provided
      --format string   Format output using a custom template:
                        'table':            Print output in table format
                        with column headers (default)
                        'table TEMPLATE':   Print output in table format
                        using the given Go template
                        'json':             Print in JSON format
                        'TEMPLATE':         Print output using the given
                        Go template.
                        Refer to https://docs.docker.com/go/formatting/
                        for more information about formatting output with
                        templates
  -n, --last int        Show n last created containers (includes all
                        states) (default -1)
  -l, --latest          Show the latest created container (includes all
                        states)
      --no-trunc        Don't truncate output
  -q, --quiet           Only display container IDs
  -s, --size            Display total file sizes
```

3 - DOCKER STOP

INFORMATIONS :

Description :
Cette commande arrête un conteneur en cours d'exécution.

Utilisation :
`docker stop [OPTIONS] CONTAINER [CONTAINER..]`

Exemple :
`docker stop mon_conteneur` (Cela arrête un conteneur avec le nom `mon_conteneur`.)

```
> docker stop --help

Usage:  docker stop [OPTIONS] CONTAINER [CONTAINER...]

Stop one or more running containers

Aliases:
  docker container stop, docker stop

Options:
  -s, --signal string   Signal to send to the container
  -t, --time int        Seconds to wait before killing the container
```

4 - DOCKER RM

INFORMATIONS :

Description :
Cette commande supprime un ou plusieurs conteneurs arrêtés.

Utilisation :
`docker rm [OPTIONS] CONTAINER [CONTAINER...]`

Exemple :
`docker rm mon_conteneur` (Cela supprime un conteneur avec le nom `mon_conteneur`. Pour supprimer tous les conteneurs arrêtés, utilisez `docker rm $(docker ps -a -q)`.)

```
> docker rm --help

Usage:  docker rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

Aliases:
  docker container rm, docker container remove, docker rm

Options:
  -f, --force     Force the removal of a running container (uses SIGKILL)
  -l, --link      Remove the specified link
  -v, --volumes   Remove anonymous volumes associated with the container
```

5 - DOCKER IMAGES

INFORMATIONS :

Description :
Cette commande liste toutes les images Docker sur l'hôte.

Utilisation :
`docker images [OPTIONS] [REPOSITORY[:TAG]]`

Exemple :
`docker images` (Cela liste toutes les images. Pour lister les images d'un dépôt spécifique, utilisez `docker images [REPOSITORY]`.)

```
> docker images --help

Usage:  docker images [OPTIONS] [REPOSITORY[:TAG]]

List images

Aliases:
  docker image ls, docker image list, docker images

Options:
  -a, --all             Show all images (default hides intermediate images)
      --digests         Show digests
  -f, --filter filter   Filter output based on conditions provided
      --format string   Format output using a custom template:
                        'table':            Print output in table format with
                        column headers (default)
                        'table TEMPLATE':   Print output in table format using the
                        given Go template
                        'json':             Print in JSON format
                        'TEMPLATE':         Print output using the given Go
                        template.
                        Refer to https://docs.docker.com/go/formatting/ for more
                        information about formatting output with templates
      --no-trunc        Don't truncate output
  -q, --quiet           Only show image IDs
```

6 - DOCKER RMI

INFORMATIONS :

Description :

Cette commande supprime une ou plusieurs images Docker.

Utilisation :

`docker rmi [OPTIONS] IMAGE [IMAGE...]`

Exemple :

`docker rmi mon_image` (Cela supprime une image avec le nom `mon_image`. Pour forcer la suppression, utilisez `docker rmi -f mon_image`.)

```
> docker rmi --help

Usage:  docker rmi [OPTIONS] IMAGE [IMAGE...]

Remove one or more images

Aliases:
  docker image rm, docker image remove, docker rmi

Options:
  -f, --force      Force removal of the image
      --no-prune   Do not delete untagged parents
```

COMMANDES ANNEXES DOCKER

DOCKER BUILD

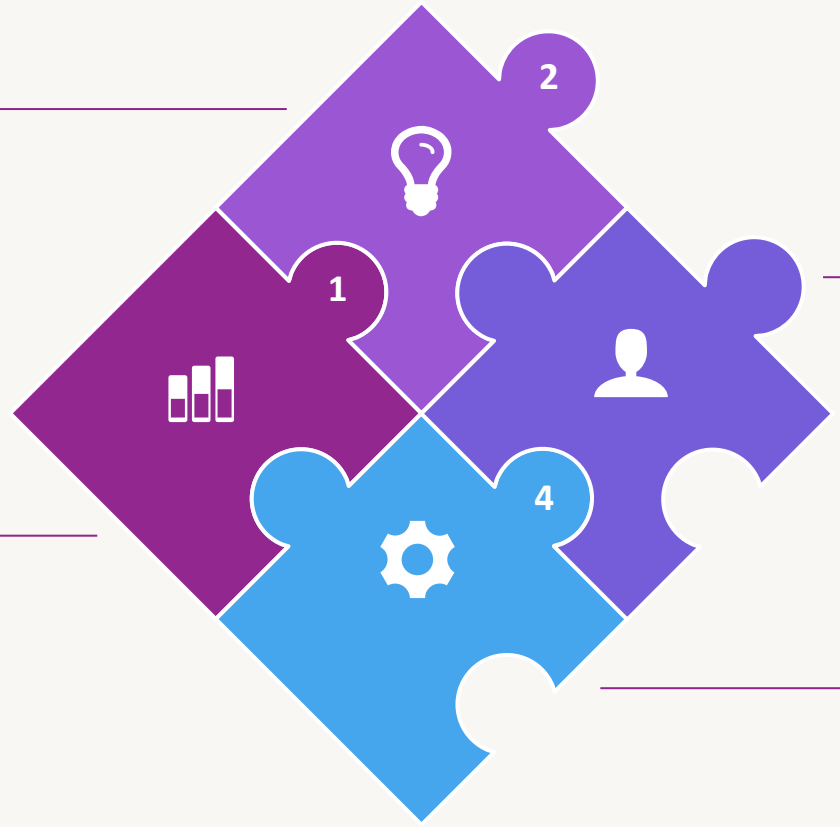
Construction d'une Image Docker

docker build -t <name>:<tag> <path>

DOCKER PULL

Télécharger une image

docker pull <name>:<tag>



DOCKER EXEC

Pour exécuter des commandes dans un conteneur en cours d'exécution

docker exec -it <conteneur> <commande>
(docker exec -it mon_nginx /bin/bash)

DOCKER SEARCH

Rechercher des images

docker search <name>

UTILISATION DE DOCKER

CRÉER VOS PROPRES IMAGES

LES DOCKERFILE

DEFINITION :

Un Dockerfile est un fichier texte contenant les instructions nécessaires pour construire une image Docker.

Voici un exemple de Dockerfile et une explication des directives de base

LISTE DES COMMANDES :

1. docker build -t <name>:<tag> <path>
2. docker run -d -p <port machine>:<port_conteneur> <name>:<tag>

Le conteneur est exécuté en arrière-plan grâce à l'option `-d`.
Le port 5000 sur la machine hôte est mappé au port 5000 dans le conteneur, ce qui permet d'accéder à l'application en cours d'exécution à l'intérieur du conteneur via `http://localhost:5000`.

```
# Utiliser une image de base
FROM ubuntu:20.04

# Définir le mainteneur de l'image
LABEL maintainer="votre_email@example.com"

# Mettre à jour le système et installer des paquets
RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip

# Copier les fichiers locaux dans l'image
COPY . /app

# Définir le répertoire de travail
WORKDIR /app

# Installer les dépendances Python
RUN pip3 install -r requirements.txt

# Exposer le port utilisé par l'application
EXPOSE 5000

# Commande pour lancer l'application
CMD ["python3", "app.py"]
```

RÉSEAUX ET VOLUMES

RÉSEAUX

LES RÉSEAUX SOUS DOCKER

Les réseaux Docker permettent aux conteneurs de communiquer entre eux et avec d'autres services extérieurs. Chaque conteneur peut être connecté à un ou plusieurs réseaux. Docker gère automatiquement les réseaux par défaut, mais il est également possible de créer et de gérer des réseaux personnalisés.

LES TYPES DE RÉSEAUX

01

BRIDGE

C'est le type de réseau par défaut lorsque vous créez un conteneur.

Les conteneurs connectés à ce réseau peuvent communiquer entre eux, mais ils ne sont pas accessibles directement depuis l'extérieur du réseau host (sauf si des ports sont explicitement mappés).

02

HOST

Utilise le réseau de l'hôte directement, ce qui peut améliorer les performances pour certaines applications.

Les conteneurs n'ont pas d'isolation réseau de l'hôte.

03

NONE

Désactive le réseautage pour le conteneur.

Utilisé lorsque vous voulez que le conteneur soit complètement isolé au niveau réseau.

04

OVERLAY

Utilisé pour créer des réseaux qui s'étendent sur plusieurs hôtes Docker.

Nécessite un orchestrateur comme Docker Swarm ou Kubernetes.

RÉSEAUX ET VOLUMES

VOLUMES

LES VOLUMES SOUS DOCKER

Les volumes Docker permettent de persister les données générées et utilisées par les conteneurs. Les volumes sont stockés sur le système de fichiers de l'hôte et sont entièrement gérés par Docker.

Les volumes peuvent être montés dans un conteneur au moment de sa création. Cela permet au conteneur d'accéder et de stocker des données sur le volume

SAUVEGARDE ET RESTAURATION DES VOLUMES

Pour sauvegarder et restaurer les données d'un volume, vous pouvez utiliser des conteneurs temporaires pour copier les données vers ou depuis le volume.

Sauvegarde des données d'un volume :

```
docker run --rm -v my_volume:/volume -v $(pwd):/backup busybox tar cvf /backup/backup.tar /volume
```

Restauration des données vers un volume :

```
docker run --rm -v my_volume:/volume -v $(pwd):/backup busybox tar xvf /backup/backup.tar -C /volume
```

DOCKER COMPOSE

UN OUTILS POUR LES GOUVERNER TOUS

DOCKER COMPOSE CONTENUS

DOCKER COMPOSE

INTRODUCTION

C'EST QUOI ?

Docker Compose est un outil de Docker qui permet de définir et de gérer des applications multi-conteneurs. Grâce à un fichier de configuration YAML (`docker-compose.yml`), vous pouvez spécifier les services, les réseaux et les volumes nécessaires pour votre application, puis utiliser des commandes simples pour gérer l'ensemble du cycle de vie de votre application.

Avantages de l'utilisation de Docker Compose

- **Simplicité** : Un seul fichier YAML pour définir toute l'architecture de l'application.
- **Portabilité** : Facilite le partage et la versioning de la configuration de l'application.
- **Orchestration** : Simplifie le démarrage, l'arrêt et la mise à jour de l'ensemble des services.
- **Isolation** : Chaque service s'exécute dans son propre conteneur, ce qui réduit les conflits entre les dépendances.
- **Évolutivité** : Facile à adapter pour des environnements de développement, de test et de production.

EXEMPLE

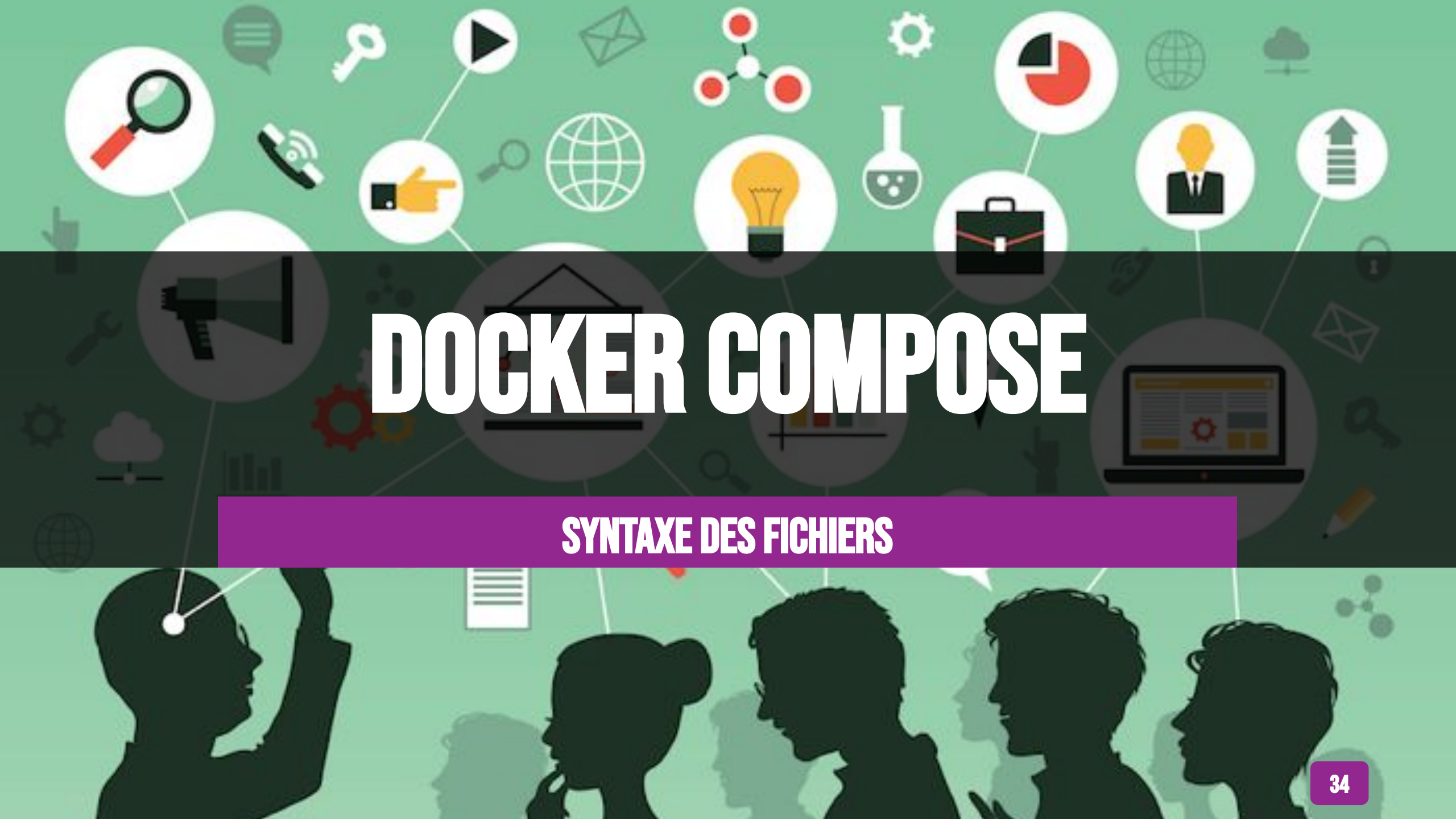
```yaml
version: '3.8'

services:
  backend:
    image: python:3.9-slim
    working_dir: /app
    volumes:
      - ./backend:/app
    command: ["flask", "run", "--host=0.0.0.0"]
    environment:
      FLASK_APP: app.py
      FLASK_ENV: development
      DATABASE_URL: postgres://user:password@db:5432/mydatabase
    networks:
      - webnet
    depends_on:
      - db
    restart: always
    resources:
      limits:
        cpus: "0.5"
        memory: "512M"
```

```yaml
  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
    volumes:
      - dbdata:/var/lib/postgresql/data
    networks:
      - webnet
    restart: unless-stopped

networks:
  webnet:
    driver: bridge

volumes:
  dbdata:
```



DOCKER COMPOSE

SYNTAXE DES FICHIERS

STRUCTURE DE BASE

```yaml
version: '3.8'
services:
  # Définition des services ici
networks:
  # Définition des réseaux ici
volumes:
  # Définition des volumes ici
```

LES SERVICES

```yaml
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    networks:
      - webnet
    volumes:
      - webdata:/usr/share/nginx/html
    environment:
      - NGINX_HOST=nginx
      - NGINX_PORT=80
    depends_on:
      - db
    restart: always
    command: ["nginx", "-g", "daemon off;"]
    entrypoint: ["sh", "-c"]
    resources:
      limits:
        cpus: "0.5"
        memory: "512M"
```

```yaml
  db:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
      - dbdata:/var/lib/mysql
    networks:
      - webnet
    restart: unless-stopped
```

LES RÉSEAUX

INFORMATIONS :

Les réseaux permettent aux services de communiquer entre eux.

```
networks:
  webnet:
    driver: bridge
```

LES VOLUMES

INFORMATIONS :

Les volumes permettent de persister les données générées par les conteneurs.

```
volumes:
  webdata:
  dbdata:
```

LES VOLUMES ET LES SCRIPTS

INFORMATIONS :

Les volumes permettent de persister les données générées par les conteneurs, de plus vous pouvez vous en servir pour initialiser une base de données

```
volumes:
  - ./db/data:/var/lib/mysql
  - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql
```

LES VARIABLES D'ENVIRONNEMENT

INFORMATIONS :

Les variables d'environnement sont utilisées pour configurer les conteneurs.

```
environment:
  - ENV_VAR_NAME=value
```

LES PORTS

INFORMATIONS :

Les ports permettent de mapper les ports de l'hôte aux conteneurs.

```
ports:
  - "host_port:container_port"
```

LES DÉPENDANCES ENTRE SERVICES

INFORMATIONS :

La directive depends_on définit les services qui doivent être démarrés avant le service courant.

```
depends_on:
  - service_name
```

LES HEALTHCHECK

INFORMATIONS :

Cela permettra à Docker Compose de s'assurer que les services sont prêts avant de démarrer les services dépendants.

```
healthcheck:
  test: ["CMD-SHELL", "pg_isready -U user -d mydatabase"]
  interval: 30s
  timeout: 10s
  retries: 5
```

LES DEPEND_ON LIÉ AU HEALTHCHECK

INFORMATIONS :

Cela permettra à Docker Compose de s'assurer que les services sont prêts avant de démarrer les services dépendants.

```
depends_on:
  db:
    condition: service_healthy
```

ENTRYPOINT ET COMMAND

INFORMATIONS :

Vous pouvez personnaliser la commande et le point d'entrée des conteneurs.
Cela en plus de ce que contient votre dockerfile

```
command: ["command", "arg1", "arg2"]
entrypoint: ["entrypoint.sh", "arg1", "arg2"]
```

OPTION DE REDÉMARRAGE

INFORMATIONS :

Les options de redémarrage contrôlent le comportement de redémarrage des conteneurs.

```
restart: "no" | "always" | "on-failure" | "unless-stopped"
```

LES TYPES DE RESTART

01

“NO”

Aucun redémarrage automatique du conteneur ne sera tenté. Cette option est utile lorsque vous voulez que le conteneur s'exécute une seule fois et ne se relance pas en cas de panne.

02

“ALWAYS”

Le conteneur sera toujours redémarré, quel que soit le mode de sortie (qu'il soit arrêté manuellement ou suite à une défaillance). C'est utile pour les services qui doivent être disponibles en permanence.

03

“ON-FAILURE”

Le conteneur sera redémarré uniquement s'il se termine avec un code d'erreur non nul (ce qui indique une défaillance). Il ne redémarrera pas si le conteneur est arrêté manuellement ou s'il se termine avec succès (code d'erreur zéro).

04

UNLESS-STOPPED

Le conteneur sera redémarré sauf s'il a été explicitement arrêté. Cela signifie que les conteneurs redémarreront toujours sauf si vous les arrêtez manuellement. C'est similaire à always, mais avec une exception pour les arrêts manuels.

LES LIMITES SUR LES RESSOURCES

INFORMATIONS :

Vous pouvez définir des limites pour l'utilisation des ressources par les conteneurs.

```
resources:
  limits:
    cpus: "0.5"        # Nombre de CPUs limité à 0.5
    memory: "512M"     # Mémoire limitée à 512MB
  reservations:
    cpus: "0.25"       # CPU réservé de 0.25
    memory: "256M"     # Mémoire réservée de 256MB
```

DOCKER COMPOSE

LES WATCHERS

LES LIMITES SUR LES RESSOURCES

INFORMATIONS :

Les watchers surveillent les modifications de fichiers et rechargent automatiquement les services.

```yaml
services:
  web:
    build: .
    command: npm start
    develop:
      watch:
        - action: sync
          path: ./web
          target: /src/web
          ignore:
            - node_modules/
        - action: rebuild
          path: package.json
```

DOCKER COMPOSE

LES COMMANDES

LES COMMANDES DOCKER COMPOSE

01 docker compose up

02 docker compose down

03 docker compose start

04 docker compose stop

05 docker compose logs

06 docker compose exec <name> <cmd>

07 docker compose config

08 docker compose build

LES BONNES PRATIQUES

DOCKER ET COMMENCE BIEN S'EN SERVIR

BONNES PRATIQUES ET SÉCURITÉ

ISOLATION DES PROCESSUS

Utilisez les namespaces et les cgroups pour isoler les processus au sein des conteneurs.

MISES À JOUR RÉGULIÈRES

Gardez vos images Docker à jour avec les derniers correctifs de sécurité.

UTILISER DES UTILISATEURS NON-ROOT

Configurez vos conteneurs pour qu'ils s'exécutent avec des utilisateurs non-root.

UTILISER DES BASES D'IMAGES MINIMALES

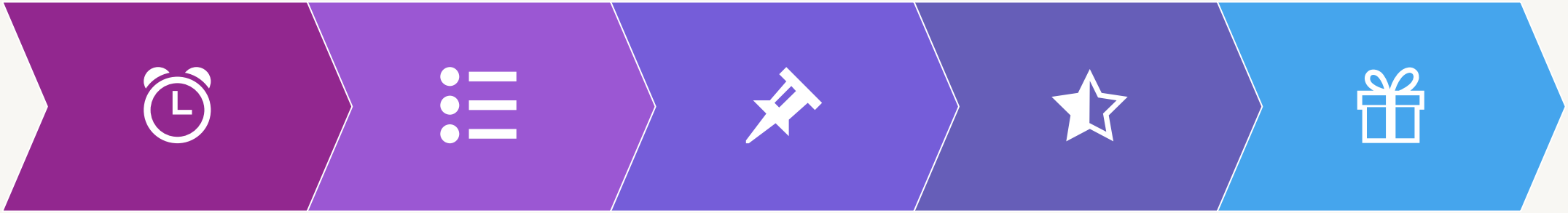
Privilégiez les bases d'images légères comme Alpine pour réduire la surface d'attaque.

SCANS DE VULNÉRABILITÉS

Intégrez des outils de scan de vulnérabilités comme Clair ou Trivy dans votre pipeline CI/CD pour vérifier les images Docker.

BONNES PRATIQUES

BONNE PRATIQUE



UTILISATION DES VARIABLES D'ENVIRONNEMENT

FICHIERS .ENV

NETTOYAGE DES CONTENEURS ET DES IMAGES INUTILISÉES

UTILISATION DE LABELS ET DE TAGS POUR L'ORGANISATION

CONFIGURATION DES VOLUMES POUR ÉVITER LES PROBLÈMES DE SYNCHRONISATION

MISE EN PRATIQUE

DE NOS JOURS, ON PEUT SURVIVRE À TOUT, SAUF À LA MORT.

DES QUESTIONS ?

DOCKER ET CI/CD

COMMENT ON DÉPLOYER AVEC DOCKER

DOCKER ET GITLAB

GITLAB :

```yaml
image: docker:latest

services:
  - docker:dind

stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - docker build -t my-app:$CI_COMMIT_SHA .

test:
  stage: test
  script:
    - docker run --rm my-app:$CI_COMMIT_SHA make test

deploy:
  stage: deploy
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker push my-app:$CI_COMMIT_SHA
```

DOCKER ET JENKINS

JENKINS :

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                script {
                    dockerImage = docker.build("my-app:${env.BUILD_ID}")
                }
            }
        }
        stage('Test') {
            steps {
                script {
                    dockerImage.inside {
                        sh 'make test'
                    }
                }
            }
        }
        stage('Deploy') {
            steps {
                script {
                    dockerImage.push()
                }
            }
        }
    }
}
```

DOCKER ET GITHUB

GITHUB :

```yaml
name: CI/CD Pipeline

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1

      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build Docker image
        run: docker build -t my-app:${{ github.sha }} .

      - name: Push Docker image
        run: docker push my-app:${{ github.sha }}

  test:
    runs-on: ubuntu-latest

    needs: build

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Pull Docker image
        run: docker pull my-app:${{ github.sha }}

      - name: Run tests
        run: docker run --rm my-app:${{ github.sha }} make test
```

etna CO-LABS

https://www.youtube.com/@ETNA_io

MERCI À VOUS !