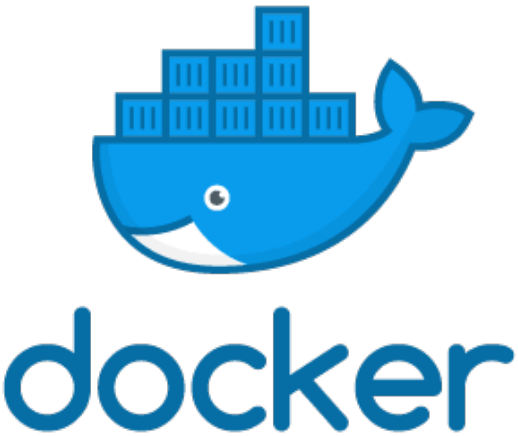
Mastering Docker

Guide Complet et Professionnel

Containers, Networking, Storage, Registry & Security



Rédigé par : CHERIF Maryem

Ingénieure en Cybersécurité | DevSecOps

9 novembre 2025

Documentation Avancée

Table des matières

Introduction

Docker est devenu l'outil de référence pour la conteneurisation d'applications, révolutionnant les pratiques DevOps, CI/CD et le déploiement cloud. Cette documentation approfondie couvre l'ensemble de l'écosystème Docker, des concepts fondamentaux aux techniques avancées de production.

Objectifs de cette documentation

— Maîtriser l'architecture et les mécanismes internes de Docker
— Optimiser les images et les performances des conteneurs
— Implémenter des stratégies de networking et storage en production
— Sécuriser l'infrastructure Docker selon les standards de l'industrie
— Automatiser le déploiement avec Docker Compose et orchestration

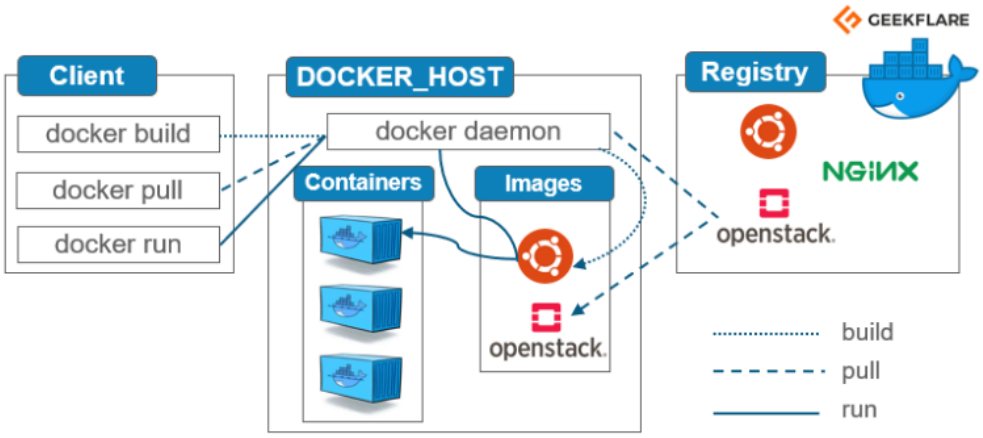


FIGURE 1 – Architecture complète de l'écosystème Docker

Fondamentaux des Conteneurs

Concept de Conteneur

Un conteneur est une unité d'exécution légère qui encapsule une application et ses dépendances dans un environnement isolé. Contrairement aux machines virtuelles qui virtualisent le matériel, les conteneurs partagent le noyau du système d'exploitation hôte, offrant ainsi une meilleure performance et une empreinte mémoire réduite.

2.1.1 Conteneurs vs Machines Virtuelles

Conteneurs	Machines Virtuelles
Partagent le noyau de l'hôte	Possèdent leur propre noyau
Démarrage en secondes	Démarrage en minutes
Légers (Mo)	Lourds (Go)
Isolation au niveau processus	Isolation complète
Moins de ressources	Plus de ressources

TABLE 1 – Comparaison Conteneurs vs VMs

Docker Engine et Daemon

Le **Docker Engine** constitue le cœur du système de conteneurisation et comprend plusieurs composants critiques :

— **dockerd (daemon)** : service principal qui gère le cycle de vie des conteneurs, images, volumes et réseaux

— **containerd** : runtime de haut niveau qui gère l'exécution des conteneurs

— **runc** : runtime de bas niveau conforme à l'OCI (Open Container Initiative)

— **Docker CLI** : interface en ligne de commande pour interagir avec le daemon

— **Docker API REST** : permet l'automatisation et l'intégration programmatique

Commandes Docker Daemon Essentielles

```
# V rifier le statut du daemon
sudo systemctl status docker
# D marrer/Arr ter le daemon
sudo systemctl start docker
sudo systemctl stop docker
# Activer au d marrage
sudo systemctl enable docker
# Visualiser les logs du daemon
sudo journalctl -u docker -f
# Informations d taill es du syst me
docker info
docker version
```

Namespaces et Cgroups

Docker utilise les fonctionnalités du noyau Linux pour assurer l'isolation :

2.3.1 Namespaces

Fournissent l'isolation des ressources système :

— **PID** : isolation des processus

— **NET** : pile réseau isolée

— **MNT** : système de fichiers isolé
— **UTS** : hostname et domaine isolés
— **IPC** : communication inter-processus isolée
— **USER** : mappage des utilisateurs

2.3.2 Control Groups (cgroups)

Limitent et contrôlent l'utilisation des ressources :

```
# Limiter la m moire
docker run -m 512m nginx
# Limiter le CPU
docker run --cpus="1.5" nginx
# Limiter les I/O disque
docker run --device-write-bps /dev/sda:1mb nginx
```

Images et Dockerfile Avancé

Architecture des Images

Les images Docker suivent une architecture en couches (layers) basée sur le système de fichiers UnionFS. Chaque instruction du Dockerfile crée une nouvelle couche immuable.

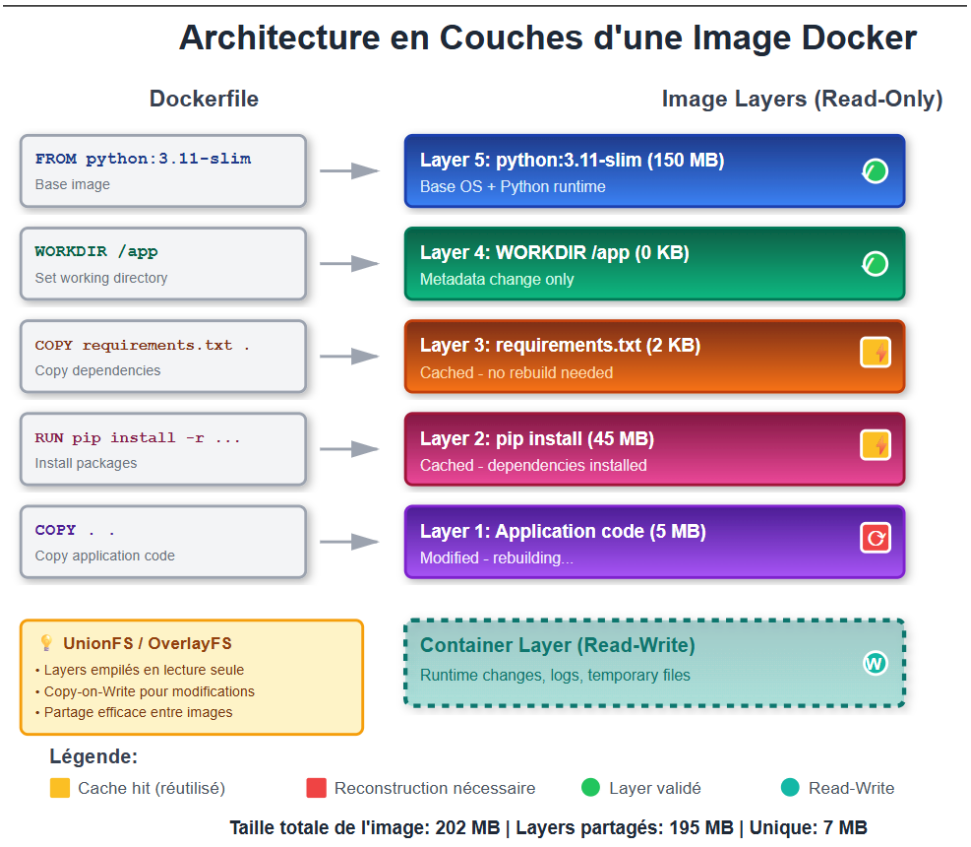


FIGURE 2 – Architecture en couches d'une image Docker avec cache

Dockerfile Optimisé - Exemple Complet

Listing 1 – Dockerfile multi-stage optimisé pour Python

```
# Stage 1: Builder
FROM python:3.11-slim AS builder
# Variables d'environnement
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PIP_NO_CACHE_DIR=1 \
    PIP_DISABLE_PIP_VERSION_CHECK=1
WORKDIR /build
# Installation des d pendances syst me
RUN apt-get update && apt-get install -y --no-install-recommends \
    gcc \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*
# Installation des d pendances Python
COPY requirements.txt .
RUN pip wheel --no-cache-dir --wheel-dir /build/wheels \
    -r requirements.txt
# Stage 2: Runtime
FROM python:3.11-slim
# Cr ation d'un utilisateur non-root
RUN groupadd -r appuser && useradd -r -g appuser appuser
WORKDIR /app
# Copie des wheels du stage builder
COPY --from=builder /build/wheels /wheels
RUN pip install --no-cache /wheels/*
# Copie du code applicatif
COPY --chown=appuser:appuser . .
# Changement vers utilisateur non-root
USER appuser
# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s \
  CMD python -c "import␣requests;␣requests.get('http://localhost
    ↪ :8000/health')"
# Exposition du port
EXPOSE 8000
# Point d'entr e
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Bonnes Pratiques Dockerfile - Guide Complet

1. **Multi-stage builds** : Séparer la compilation de l'exécution pour réduire la taille
2. **Images de base minimales** : Préférer alpine, distroless ou scratch
3. **Ordre des layers** : Placer les instructions les moins changeantes en premier
4. **Combiner les RUN** : Réduire le nombre de couches
5. **.dockerignore** : Exclure les fichiers inutiles du contexte de build

6. **Utilisateur non-root** : Ne jamais exécuter en tant que root
7. **Labels et métadonnées** : Documenter l'image avec des labels
8. **Scanning de sécurité** : Intégrer dans le pipeline CI/CD

Exemple .dockerignore

```
# Fichiers Git
.git
.gitignore
# Environnements virtuels
venv/
env/
*.pyc
__pycache__/
# Documentation
*.md
docs/
# Tests
tests/
*.test
# Fichiers de configuration locale
.env
.env.local
docker-compose*.yml
```

Gestion Avancée des Images

```
# Build avec cache personnalis
docker build --cache-from myapp:latest -t myapp:v2.0 .
# Build multi-plateforme (ARM + AMD64)
docker buildx build --platform linux/amd64,linux/arm64 \
  -t myapp:multiarch --push .
# Inspection d'une image
docker image inspect nginx:latest
docker history nginx:latest
# Nettoyage des images inutilis es
docker image prune -a --filter "until=168h"
# Export/Import d'images
docker save -o myapp.tar myapp:latest
docker load -i myapp.tar
```

Docker Networking Avancé

Types de Réseaux Docker

Docker propose plusieurs types de réseaux pour isoler et gérer la communication entre conteneurs. Chaque type a ses usages et ses particularités.

Bridge

Bridge : Réseau privé par défaut pour les conteneurs sur un même hôte. Les conteneurs communiquent entre eux via ce réseau isolé.

Host

Host : Le conteneur partage la pile réseau de l'hôte. Pas d'isolation réseau, ce qui peut améliorer les performances réseau.

Overlay

Overlay : Permet la communication entre conteneurs sur différents hôtes dans un cluster Docker Swarm. Idéal pour les services distribués.

Macvlan

Macvlan : Attribue une adresse MAC unique à chaque conteneur, permettant au conteneur d'apparaître comme un périphérique réseau physique sur le réseau local.

None

None : Aucun réseau. Le conteneur n'a pas d'accès réseau externe et fonctionne isolé.

Driver	**Usage**	**Scope**
bridge	Réseau privé par défaut	local
host	Partage la stack réseau de l'hôte	local
overlay	Communication multi-hôtes (Swarm)	swarm
macvlan	Attribution d'adresses MAC	local
none	Aucun réseau	local

TABLE 2 – Résumé des drivers réseau Docker

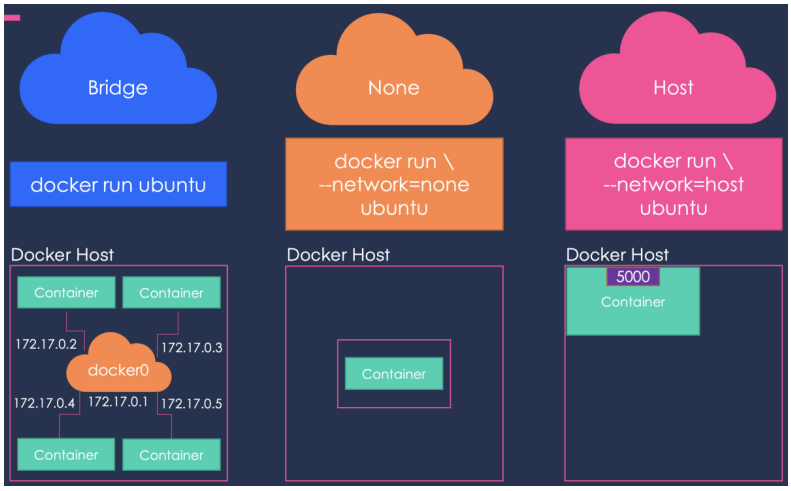


FIGURE 3 – Topologie des différents types de réseaux Docker

Configuration Réseau Avancée

```
# Cr er un r seau bridge avec subnet personnalis
docker network create --driver bridge \
  --subnet=172.20.0.0/16 \
  --ip-range=172.20.240.0/20 \
  --gateway=172.20.0.1 \
  --opt "com.docker.network.bridge.name"="br-custom" \
  my_custom_network
# R seau overlay pour Docker Swarm
docker network create --driver overlay \
  --attachable \
  --subnet=10.0.9.0/24 \
  my_overlay_network
# R seau macvlan (acc s direct au r seau physique)
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 \
  macvlan_net
# Connecter un conteneur    plusieurs r seaux
docker network connect frontend web_server
docker network connect backend web_server
```

DNS et Service Discovery

Docker fournit un DNS embarqué pour la résolution de noms entre conteneurs :

```
# Les conteneurs peuvent se r soudre par leur nom
docker run -d --name db --network app_net postgres
docker run -d --name api --network app_net myapi
# L'API peut acc der    : postgresql://db:5432
# Alias DNS personnalis s
docker run -d --name web --network app_net \
  --network-alias webserver \
  --network-alias www nginx
```

Inspection et Débogage Réseau

```
# Inspecter un r seau
docker network inspect my_network
# Voir les conteneurs connect s
docker network inspect my_network \
  --format='{{range␣.Containers}}{{.Name}}␣{{end}}'
# Tester la connectivit
docker exec web ping -c 3 api
# Capturer le trafic r seau
docker run --rm --net container:web \
  nicolaka/netshoot tcpdump -i eth0
```

Docker Storage - Gestion Avancée

Types de Stockage - Comparaison

Type	Avantages	Inconvénients
Volumes	Gérés par Docker, performants, sauvegardables	Moins de contrôle direct
Bind Mounts	Accès direct au système hôte	Dépendance au chemin hôte
tmpfs	Très rapide (RAM)	Données volatiles

TABLE 3 – Comparaison des types de stockage Docker

Volumes - Manipulation Avancée

```
# Cr er un volume avec driver sp cifique
docker volume create --driver local \
  --opt type=nfs \
  --opt o=addr=192.168.1.100,rw \
  --opt device=:/path/to/share \
  nfs_volume
# Cr er un volume avec labels
docker volume create --label env=production \
  --label backup=daily \
  prod_data
# Lister avec filtres
docker volume ls --filter label=env=production
# Utiliser un volume avec permissions
docker run -v myvolume:/data:ro nginx # Read-only
docker run -v myvolume:/data:rw nginx # Read-write
# Partager un volume entre conteneurs
docker run -d --name app1 -v shared_data:/data app1
docker run -d --name app2 -v shared_data:/data app2
```

Backup et Restauration - Stratégies Professionnelles

```
# Backup complet d'un volume avec compression
docker run --rm \
  -v my_volume:/source:ro \
  -v $(pwd):/backup \
  alpine \
  tar czf /backup/backup-$(date +%Y%m%d-%H%M%S).tar.gz -C /source
    ↪ .
# Restauration d'un volume
docker run --rm \
  -v my_volume:/target \
  -v $(pwd):/backup \
  alpine \
  tar xzf /backup/backup-20250108-120000.tar.gz -C /target
```

```
# Cloner un volume
docker volume create new_volume
docker run --rm \
  -v old_volume:/source:ro \
  -v new_volume:/target \
  alpine \
  sh -c "cp␣-av␣/source/.␣/target/"
# Backup vers S3 (AWS)
docker run --rm \
  -v my_volume:/data:ro \
  -e AWS_ACCESS_KEY_ID \
  -e AWS_SECRET_ACCESS_KEY \
  amazon/aws-cli \
  s3 sync /data s3://my-bucket/backups/$(date +%Y%m%d)/
```

Drivers de Volume

Docker supporte plusieurs drivers de volume pour différents backends :

- **local** : stockage local (par défaut)
- **nfs** : Network File System
- **cifs/smb** : Windows file shares
- **rexray** : stockage cloud (AWS EBS, Azure Disk)
- **convoy** : snapshots et backups
- **flocker** : migration de données entre hôtes

Docker Registry - Infrastructure Privée

Types de Registries

- **Docker Hub** : registre public officiel
- **Harbor** : registre d'entreprise avec scanning de sécurité
- **Quay.io** : registre de Red Hat avec fonctionnalités avancées
- **GitHub Container Registry** : intégré à GitHub
- **AWS ECR / Azure ACR / GCP GCR** : registres cloud natifs
- **Docker Registry (OSS)** : registre open-source simple

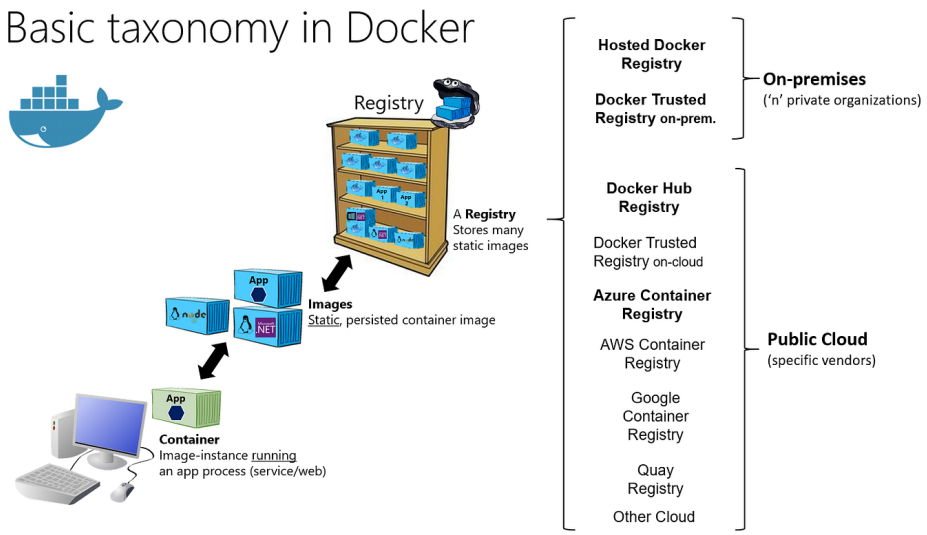


FIGURE 4 – Architecture d'un Docker Registry privé avec proxy inverse

Déploiement d'un Registry Privé Sécurisé

Listing 2 – Docker Compose pour Registry avec Nginx

```
# docker-compose.yml
version: '3.8'
services:
  registry:
    image: registry:2
    restart: always
    environment:
      REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
      REGISTRY_AUTH: htpasswd
      REGISTRY_AUTH_HTPASSWD_REALM: Registry
      REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
      REGISTRY_STORAGE_DELETE_ENABLED: 'true'
    volumes:
      - registry_data:/data
      - ./auth:/auth
    networks:
      - registry_net
  nginx:
    image: nginx:alpine
    restart: always
    ports:
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./ssl:/etc/nginx/ssl:ro
    depends_on:
      - registry
    networks:
      - registry_net
```

```
volumes:
  registry_data:
networks:
  registry_net:
```

Configuration Nginx pour Registry

Listing 3 – Configuration Nginx avec SSL

```
# nginx.conf
events {
    worker_connections 1024;
}
http {
    upstream docker-registry {
        server registry:5000;
    }
    server {
        listen 443 ssl http2;
        server_name registry.example.com;
        ssl_certificate /etc/nginx/ssl/cert.pem;
        ssl_certificate_key /etc/nginx/ssl/key.pem;
        ssl_protocols TLSv1.2 TLSv1.3;
        ssl_ciphers HIGH:!aNULL:!MD5;
        client_max_body_size 0;
        chunked_transfer_encoding on;
        location / {
            proxy_pass http://docker-registry;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For
               ↪ $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_read_timeout 900;
        }
    }
}
```

Authentification et Gestion des Accès

```
# Cr er un fichier htpasswd
docker run --rm --entrypoint htpasswd \
  httpd:2 -Bbn admin password > auth/htpasswd
# Login au registry
docker login registry.example.com
# Tag et push d'une image
docker tag myapp:latest registry.example.com/myapp:v1.0
docker push registry.example.com/myapp:v1.0
# Pull depuis le registry
```

```
docker pull registry.example.com/myapp:v1.0
# Lister les images dans le registry
curl -X GET https://registry.example.com/v2/_catalog \
  -u admin:password
```

Docker Compose - Orchestration Multi-Conteneurs

Exemple Complet - Application 3-Tiers

Listing 4 – docker-compose.yml pour stack complète

```
version: '3.8'
services:
  # Base de donn es PostgreSQL
  database:
    image: postgres:15-alpine
    restart: unless-stopped
    environment:
      POSTGRES_DB: ${DB_NAME:-myapp}
      POSTGRES_USER: ${DB_USER:-postgres}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql:ro
    networks:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "pg_isready␣-U␣postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
  # Cache Redis
  cache:
    image: redis:7-alpine
    restart: unless-stopped
    command: redis-server --appendonly yes --requirepass ${
      ↪ REDIS_PASSWORD}
    volumes:
      - redis_data:/data
    networks:
      - backend
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 3s
      retries: 5
  # API Backend
  api:
    build:
      context: ./backend
```

```
      dockerfile: Dockerfile
      target: production
    restart: unless-stopped
    environment:
      DATABASE_URL: postgresql://${DB_USER}:${DB_PASSWORD}
        ↪ @database:5432/${DB_NAME}
      REDIS_URL: redis://:${REDIS_PASSWORD}@cache:6379/0
      JWT_SECRET: ${JWT_SECRET}
    depends_on:
      database:
        condition: service_healthy
      cache:
        condition: service_healthy
    networks:
      - backend
      - frontend
    deploy:
      replicas: 2
      resources:
        limits:
          cpus: '1'
          memory: 512M
  # Frontend Web
  web:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    restart: unless-stopped
    environment:
      API_URL: http://api:8000
    depends_on:
      - api
    networks:
      - frontend
  # Reverse Proxy Nginx
  nginx:
    image: nginx:alpine
    restart: unless-stopped
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./nginx/ssl:/etc/nginx/ssl:ro
      - ./nginx/logs:/var/log/nginx
    depends_on:
      - web
      - api
    networks:
      - frontend
volumes:
```

```
  postgres_data:
    driver: local
  redis_data:
    driver: local
networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
    internal: true
```

Commandes Docker Compose Avancées

```
# D marrage avec logs
docker-compose up -d && docker-compose logs -f
# Rebuild et red marrage d'un service sp cifique
docker-compose up -d --build --force-recreate api
# Scaling d'un service
docker-compose up -d --scale api=5
# Ex cuter des commandes dans un service
docker-compose exec api python manage.py migrate
docker-compose exec database psql -U postgres
# Variables d'environnement depuis multiples fichiers
docker-compose --env-file .env.prod \
  -f docker-compose.yml \
  -f docker-compose.prod.yml up -d
# Voir la configuration fusionn e
docker-compose config
# Voir les ressources utilis es
docker-compose top
```

Sécurité Docker - Guide Complet

Principes de Sécurité

8.1.1 1. Sécurité des Images

```
# Scanner avec Trivy
trivy image --severity HIGH,CRITICAL nginx:latest
# Scanner avec Docker Scout
docker scout cves nginx:latest
# Scanner avec Snyk
snyk container test nginx:latest
# Signer les images avec Docker Content Trust
export DOCKER_CONTENT_TRUST=1
docker push registry.example.com/myapp:signed
```

8.1.2 2. Isolation et Capabilities

```
# Limiter les capabilities
docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx
# Utiliser le mode read-only
docker run --read-only --tmpfs /tmp nginx
# AppArmor / SELinux
docker run --security-opt apparmor=docker-default nginx
docker run --security-opt label=level:s0:c100,c200 nginx
# D sactiver les nouveaux privil ges
docker run --security-opt=no-new-privileges nginx
```

8.1.3 3. User Namespaces

```
# Configuration du daemon pour user namespaces
# /etc/docker/daemon.json
{
  "userns-remap": "default"
}
# Red marrer Docker
sudo systemctl restart docker
```

Checklist de Sécurité Docker

Checklist Sécurité

- ✓ Toujours utiliser des images officielles ou vérifiées
- ✓ Scanner les images avant déploiement
- ✓ Utiliser des tags spécifiques, jamais `latest`
- ✓ Exécuter les conteneurs en tant qu'utilisateur non-root
- ✓ Limiter les ressources (CPU, mémoire, I/O)
- ✓ Utiliser des réseaux privés
- ✓ Activer Docker Content Trust
- ✓ Mettre à jour régulièrement Docker Engine
- ✓ Auditer avec Docker Bench Security
- ✓ Chiffrer les données sensibles
- ✓ Implémenter la rotation des secrets
- ✓ Monitorer les logs et métriques

Docker Secrets (Swarm)

```
# Cr er un secret
echo "mysecretpassword" | docker secret create db_password -
# Utiliser dans un service
```

```
docker service create --name myapp \
  --secret db_password \
  myimage
# Acc der au secret dans le conteneur
# Secret disponible dans /run/secrets/db_password
```

Monitoring et Logging

Monitoring avec Prometheus et Grafana

Le monitoring permet de surveiller les performances et la disponibilité de vos conteneurs Docker et de votre infrastructure. Dans cette configuration, nous utilisons :

- **Prometheus :** Collecte et stocke les métriques (CPU, mémoire, etc.) depuis des exporteurs.
- **Grafana :** Visualise les métriques sous forme de dashboards interactifs.
- **cAdvisor :** Exporteur de métriques Docker (utilisation CPU, mémoire, réseau des conteneurs).
- **Node Exporter :** Exporteur de métriques système (CPU, mémoire, disque, réseau).

Listing 5 – Stack de monitoring complète avec Docker Compose

```
version: '3.8'
services:
  # --- Prometheus ---
  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml #
        ↪ configuration Prometheus
      - prometheus_data:/prometheus                    # stockage
        ↪ persistant
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.retention.time=30d'           #
        ↪ conservation des m triques sur 30 jours
    ports:
      - "9090:9090"
    networks:
      - monitoring

  # --- Grafana ---
  grafana:
    image: grafana/grafana:latest
    environment:
      GF_SECURITY_ADMIN_PASSWORD: ${GRAFANA_PASSWORD}  # mot de
        ↪ passe admin
      GF_INSTALL_PLUGINS: grafana-piechart-panel       # plugin
        ↪ suppl mentaire
```

```
    volumes:
      - grafana_data:/var/lib/grafana                    # stockage
        ↪ persistant
      - ./grafana/dashboards:/etc/grafana/provisioning/dashboards
    ports:
      - "3000:3000"
    depends_on:
      - prometheus
    networks:
      - monitoring

  # --- cAdvisor ---
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    privileged: true
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
      - /dev/disk/:/dev/disk:ro
    ports:
      - "8080:8080"
    networks:
      - monitoring

  # --- Node Exporter ---
  node_exporter:
    image: prom/node-exporter:latest
    command:
      - '--path.rootfs=/host'
    volumes:
      - /:/host:ro,rslave
    ports:
      - "9100:9100"
    networks:
      - monitoring

# --- Volumes persistants ---
volumes:
  prometheus_data:
  grafana_data:

# --- R seau  d  di   Monitoring ---
networks:
  monitoring:
    driver: bridge
```

Logging Centralisé avec ELK Stack

Le stack ELK (Elasticsearch, Logstash, Kibana) permet de centraliser et visualiser les logs de vos conteneurs Docker.

— **Elasticsearch :** Base de données pour stocker et indexer les logs.

— **Logstash :** Collecte, transforme et envoie les logs vers Elasticsearch.

— **Kibana :** Interface graphique pour visualiser et analyser les logs.

Listing 6 – Docker Compose pour ELK Stack avec explications

```
version: '3.8'
services:
  # --- Elasticsearch ---
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.11.0
    environment:
      - discovery.type=single-node       # mode single-node pour
        ↪ simplifier
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m" # limite m moire JVM
      - xpack.security.enabled=false     # d sactive la
        ↪ s curit  pour test
    volumes:
      - elasticsearch_data:/usr/share/elasticsearch/data
    ports:
      - "9200:9200"                      # acc s HTTP
    networks:
      - elk

  # --- Logstash ---
  logstash:
    image: docker.elastic.co/logstash/logstash:8.11.0
    volumes:
      - ./logstash/pipeline:/usr/share/logstash/pipeline #
        ↪ configuration des pipelines
    ports:
      - "5000:5000/tcp"                  # entr e TCP pour les
        ↪ logs
      - "5000:5000/udp"                  # entr e UDP pour les
        ↪ logs
      - "9600:9600"                      # API monitoring
    depends_on:
      - elasticsearch
    networks:
      - elk

  # --- Kibana ---
  kibana:
    image: docker.elastic.co/kibana/kibana:8.11.0
    environment:
      ELASTICSEARCH_HOSTS: http://elasticsearch:9200
    ports:
```

```
      - "5601:5601"                    # acc s web Kibana
    depends_on:
      - elasticsearch
    networks:
      - elk

# --- Volumes persistants ---
volumes:
  elasticsearch_data:

# --- R seau d di  ELK ---
networks:
  elk:
    driver: bridge
```

Configuration des Drivers de Logging Docker

Docker permet de rediriger les logs de conteneurs vers différents systèmes de stockage ou de traitement. Voici les principaux exemples :

- **Syslog :** Envoie les logs vers un serveur Syslog externe.
- **Fluentd :** Envoie les logs vers un collecteur Fluentd pour traitement et analyse.
- **JSON-file :** Stocke les logs localement au format JSON avec rotation.
- **Configuration globale :** Permet de définir le driver et les options par défaut pour tous les conteneurs.

Listing 7 – Exemples de drivers de logging Docker

```
# Logging vers syslog
docker run --log-driver=syslog \
  --log-opt syslog-address=udp://logserver:514 \
  --log-opt tag="{{.Name}}" \
  nginx

# Logging vers Fluentd
docker run --log-driver=fluentd \
  --log-opt fluentd-address=localhost:24224 \
  --log-opt tag="docker.{{.Name}}" \
  nginx

# Logging vers JSON avec rotation
docker run --log-driver=json-file \
  --log-opt max-size=10m \
  --log-opt max-file=3 \
  nginx

# Configuration globale dans daemon.json
# /etc/docker/daemon.json
{
  "log-driver": "json-file",
```

```
  "log-opts": {
    "max-size": "10m",
    "max-file": "5",
    "labels": "production",
    "env": "os,customer"
  }
}
```

Performance et Optimisation

Optimisation des Images

Technique	Description	Gain
Multi-stage build	Séparer build et runtime	50-80%
Images Alpine	Base minimale	60-90%
Distroless	Sans OS complet	70-85%
Layer caching	Réutiliser les couches	Temps build
.dockerignore	Réduire le contexte	Temps build

TABLE 4 – Techniques d'optimisation des images

Benchmarking des Conteneurs

```
# Mesurer les performances CPU
docker run --rm -it progrium/stress --cpu 2 --timeout 60s
# Tester les performances r seau
docker run --rm -it networkstatic/iperf3 -c server_ip
# Tester les I/O disque
docker run --rm -v /data:/data \
  ubuntu:latest dd if=/dev/zero of=/data/test bs=1M count=1000
# Profiling avec cAdvisor
curl http://localhost:8080/api/v2.0/stats?type=docker&count=1
# Statistiques en temps r el
docker stats --format "table␣{{.Name}}\t{{.CPUPerc}}\t{{.MemUsage
  ↪ }}"
```

Tuning du Daemon Docker

Le fichier `/etc/docker/daemon.json` permet d'optimiser le comportement du moteur Docker sur votre serveur.

- **log-driver / log-opts :** Configure le type et la rotation des logs pour éviter de saturer le disque.
- **storage-driver / storage-opts :** Définit le driver de stockage pour les conteneurs et les volumes. `overlay2` est recommandé pour les performances.

— **max-concurrent-downloads / uploads :** Limite le nombre de téléchargements et uploads parallèles pour éviter de saturer la bande passante.

— **default-ulimits :** Définition des limites systèmes pour tous les conteneurs (ici `nofile`).

— **live-restore :** Permet à Docker de maintenir les conteneurs en fonctionnement lors d'un redémarrage du daemon.

— **userland-proxy :** Désactivé pour des performances réseau optimales.

— **icc :** Communication inter-conteneurs désactivée pour plus de sécurité.

— **default-address-pools :** Définit les plages IP par défaut pour les réseaux Docker.

Listing 8 – Fichier daemon.json optimisé

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",   # taille max d'un fichier log
    "max-file": "3"      # nombre max de fichiers log
  },
  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ],
  "max-concurrent-downloads": 10,
  "max-concurrent-uploads": 10,
  "default-ulimits": {
    "nofile": {
      "Name": "nofile",
      "Hard": 64000,
      "Soft": 64000
    }
  },
  "live-restore": true,
  "userland-proxy": false,
  "icc": false,
  "default-address-pools": [
    {
      "base": "172.80.0.0/16",
      "size": 24
    }
  ]
}
```

—

CI/CD avec Docker

La mise en place d'un pipeline CI/CD permet d'automatiser : la construction, les tests, la sécurité et le déploiement des images Docker. Nous présentons ici deux exemples : GitLab CI et GitHub Actions.

—

Pipeline GitLab CI

- **stages :** Définissent les étapes : build, test, scan (sécurité), deploy.
- **variables :** Variables globales pour Docker, tags, et registre.
- **before_script :** Commandes exécutées avant chaque job, ici login au registre Docker.
- **build :** Construction de l'image Docker et push vers le registre.
- **test :** Exécution des tests unitaires et couverture de code.
- **security_scan :** Analyse de vulnérabilités avec Trivy, bloque en cas de fail critique.
- **deploy_staging :** Déploiement sur l'environnement de test.
- **deploy_production :** Déploiement manuel sur production via Docker Stack.

Listing 9 – Pipeline GitLab CI complet

```
stages:
  - build
  - test
  - scan
  - deploy

variables:
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: "/certs"
  IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA

before_script:
  - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
    ↪ $CI_REGISTRY

build:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG
  only:
    - main
    - develop

test:
  stage: test
  image: $IMAGE_TAG
  script:
    - pytest tests/
    - coverage report
  coverage: '/TOTAL.*\s+(\d+%)$/'

security_scan:
```

```
  stage: scan
  image: aquasec/trivy:latest
  script:
    - trivy image --exit-code 1 --severity CRITICAL $IMAGE_TAG
  allow_failure: false

deploy_staging:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --no-cache openssh-client
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | ssh-add -
  script:
    - ssh user@staging-server "docker␣pull␣$IMAGE_TAG"
    - ssh user@staging-server "docker-compose␣up␣-d"
  environment:
    name: staging
    url: https://staging.example.com
  only:
    - develop

deploy_production:
  stage: deploy
  image: alpine:latest
  script:
    - ssh user@prod-server "docker␣pull␣$IMAGE_TAG"
    - ssh user@prod-server "docker␣stack␣deploy␣-c␣docker-compose.
      ↪ yml␣app"
  environment:
    name: production
    url: https://example.com
  only:
    - main
  when: manual
```

—

Pipeline GitHub Actions

GitHub Actions offre une alternative pour CI/CD : - Build et push de l'image Docker vers GHCR. - Scan de sécurité automatique avec Trivy. - Intégration des résultats dans GitHub Security pour suivi.

Listing 10 – Pipeline GitHub Actions pour Docker CI/CD

```
name: Docker CI/CD
on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]
```

```yaml

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to Container Registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Extract metadata
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
            type=ref,event=branch
            type=sha,prefix={{branch}}-
            type=semver,pattern={{version}}

      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max

      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{
            ↪  github.sha }}
          format: 'sarif'
```

```
          output: 'trivy-results.sarif'

      - name: Upload Trivy results to GitHub Security
        uses: github/codeql-action/upload-sarif@v2
        with:
          sarif_file: 'trivy-results.sarif'
```

Docker Swarm - Orchestration Native

Docker Swarm est le moteur d'orchestration natif de Docker. Il permet de gérer des clusters de plusieurs hôtes Docker et d'automatiser le déploiement, la mise à l'échelle et la haute disponibilité des services.

—

Initialisation d'un Cluster Swarm

Pour créer un cluster Swarm, il faut définir un nœud manager et ajouter des workers.

```
# --- Initialiser le manager ---
docker swarm init --advertise-addr 192.168.1.100
# Le manager g re l' tat  du cluster et coordonne les services.

# --- Ajouter des workers ---
docker swarm join --token SWMTKN-... 192.168.1.100:2377
# Chaque worker ex cute les conteneurs mais ne prend pas les
   ↪ d cisions.

# --- Lister les n uds du cluster ---
docker node ls
# Permet de v rifier l' tat  des n uds et leur r le (Manager ou
   ↪  Worker).

# --- Promouvoir un worker en manager ---
docker node promote worker-node-1
# Utile pour la haute disponibilit  ou le red marrage  d un
   ↪ manager.

# --- Labelliser les n uds pour placement de services ---
docker node update --label-add environment=production node-1
docker node update --label-add type=database node-2
# Les labels permettent de restreindre certains services
   ↪ certains n uds.
```

—

Déploiement de Services

Docker Swarm déploie des services (groupes de conteneurs) avec différentes stratégies de placement et de mise à jour.

```
# --- Cr er un service simple ---
docker service create --name web \
  --replicas 3 \
  --publish 80:80 \
  nginx:alpine
# 3 r plicas de Nginx expos s sur le port 80.

# --- Service avec contraintes de placement et volume ---
docker service create --name db \
  --constraint 'node.labels.type==database' \
  --mount type=volume,source=db_data,target=/var/lib/postgresql/
    ↪ data \
  postgres:15
# Le service PostgreSQL ne sera lanc  que sur les n uds
  ↪ labellis s "database".

# --- Service avec Rolling Update ---
docker service create --name api \
  --replicas 5 \
  --update-parallelism 2 \
  --update-delay 10s \
  --update-failure-action rollback \
  myapi:latest
# Permet de mettre    jour progressivement les conteneurs sans
  ↪ interruption de service.

# --- Mise    l' chelle  d'un service ---
docker service scale web=10
# Augmente le nombre de r plicas    10 pour g rer plus de trafic
  ↪ .

# --- Mise    jour d'un service ---
docker service update --image nginx:latest web
# Permet de mettre    jour l'image  d un  service sans le
  ↪ supprimer.

# --- Rollback en cas de probl me ---
docker service rollback web
# Restaure la version pr c dente du service si  l update
  ↪  choue .
```

—

Stack Swarm Complète

Une **stack** regroupe plusieurs services, réseaux, volumes, secrets et configurations.

Listing 11 – docker-stack.yml pour production

```
version: '3.8'
services:
  # --- Service web ---
```

```yaml
  web:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
        max_attempts: 3
      placement:
        constraints:
          - node.role == worker
          - node.labels.environment == production
    networks:
      - frontend
    configs:
      - source: nginx_config
        target: /etc/nginx/nginx.conf
    secrets:
      - ssl_cert
      - ssl_key

  # --- Service API ---
  api:
    image: myregistry.com/api:v2.0
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
        reservations:
          cpus: '0.25'
          memory: 256M
      update_config:
        parallelism: 2
        delay: 10s
        failure_action: rollback
      rollback_config:
        parallelism: 2
        delay: 5s
      restart_policy:
        condition: any
        delay: 5s
        max_attempts: 3
    networks:
      - frontend
```

```
      - backend
    secrets:
      - db_password
      - jwt_secret
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s

  # --- Service Database ---
  database:
    image: postgres:15-alpine
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.labels.type == database
      restart_policy:
        condition: on-failure
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - backend
    secrets:
      - db_password
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password

# --- Configs et secrets ---
configs:
  nginx_config:
    file: ./nginx.conf
secrets:
  db_password:
    external: true
  jwt_secret:
    external: true
  ssl_cert:
    external: true
  ssl_key:
    external: true

# --- Volumes ---
volumes:
  db_data:
    driver: local

# --- R seaux ---
networks:
```

```
  frontend:
    driver: overlay
  backend:
    driver: overlay
    internal: true
```

—

Gestion de la Stack

```
# D ployer la stack
docker stack deploy -c docker-stack.yml myapp

# Lister les stacks d ploy es
docker stack ls

# Lister les services d'une stack
docker stack services myapp

# Suivre les logs d'un service
docker service logs -f myapp_api

# Supprimer la stack
docker stack rm myapp
```

—

Résumé professionnel : - Swarm facilite l'orchestration native sans Kubernetes. - Les services peuvent être configurés avec réplicas, placement, healthchecks, rolling updates et rollback. - Les stacks permettent de gérer l'ensemble d'une application (frontend, backend, base de données) avec secrets, configs et réseaux dédiés. - Swarm assure haute disponibilité et scalabilité avec simplicité pour les environnements de production.

Troubleshooting et Débogage

Commandes de Diagnostic

```
# Inspecter un conteneur
docker inspect container_name
docker inspect --format='{{.State.Health.Status}}' container_name
# Voir les processus dans un conteneur
docker top container_name
# Statistiques en temps r el
docker stats --no-stream
#    vnements    Docker
docker events --since '30m' --filter 'type=container'
# Examiner les logs avec timestamps
docker logs --timestamps --since 30m container_name
# Suivre les logs en temps r el
docker logs -f --tail 100 container_name
# Ex cuter un shell dans un conteneur
```

```
docker exec -it container_name /bin/sh
# Copier des fichiers depuis/vers un conteneur
docker cp container_name:/app/logs/app.log ./
docker cp ./config.yml container_name:/app/config/
# Comparer les changements du syst me de fichiers
docker diff container_name
# Exporter le syst me de fichiers d'un conteneur
docker export container_name > container_fs.tar
```

Problèmes Courants et Solutions

Résolution de Problèmes

1. Conteneur qui redémarre en boucle

```
# Voir les derniers logs
docker logs --tail 50 container_name
# D sactiver le restart pour d boguer
docker update --restart=no container_name
# V rifier le healthcheck
docker inspect --format='{{json .State.Health}}'
    ↪ container_name
```

2. Problème de réseau

```
# Tester la connectivit
docker exec container1 ping container2
# Voir la configuration r seau
docker network inspect network_name
# Nettoyer les r seaux orphelins
docker network prune
```

3. Manque d'espace disque

```
# Voir l'utilisation disque
docker system df
# Nettoyage complet
docker system prune -a --volumes
# Nettoyer s lectivement
docker image prune -a --filter "until=168h"
docker volume prune --filter "label!=keep"
```

4. Performances lentes

```
# Profiler un conteneur
docker stats container_name
# Limiter les ressources
docker update --cpus="1.5" --memory="1g" container_name
# V rifier les I/O
docker exec container_name iostat -x 1
```

Migration vers Docker

Stratégie de Containerisation

1. **Évaluation** : Identifier les applications candidates
2. **Stateless d'abord** : Commencer par les apps sans état
3. **Dépendances** : Mapper toutes les dépendances externes
4. **Données** : Planifier la stratégie de persistance
5. **Configuration** : Externaliser les configs (12-factor app)
6. **Testing** : Valider en environnement de staging
7. **Monitoring** : Mettre en place l'observabilité
8. **Rollout progressif** : Déployer par phases

Checklist Pre-Migration

✓ Architecture applicative documentée
✓ Dépendances identifiées et versionnées
✓ Stratégie de gestion des données définie
✓ Plan de rollback préparé
✓ Métriques de performance baseline établies
✓ Tests de charge planifiés
✓ Documentation d'exploitation créée
✓ Formation des équipes effectuée

Ressources et Outils

Outils Essentiels

Outil	Usage	URL
Docker Desktop	Dev local (Mac/Windows)	docker.com
Portainer	GUI de gestion	portainer.io
Dive	Analyse d'images	github.com/wagoodman/dive
Hadolint	Lint Dockerfile	hadolint.github.io
Docker Bench	Audit sécurité	github.com/docker/docker-bench-security
Trivy	Scan vulnérabilités	aquasecurity.github.io/trivy
Lazydocker	TUI pour Docker	github.com/jesseduffield/lazydocker
Watchtower	Auto-update conteneurs	containrrr.dev/watchtower

TABLE 5 – Outils recommandés pour l'écosystème Docker

Commandes de Maintenance

```
# Audit de s curit  automatique
docker run --rm -it \
  --net host \
  --pid host \
  --userns host \
  --cap-add audit_control \
  -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
  -v /var/lib:/var/lib \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v /etc:/etc \
  --label docker_bench_security \
  docker/docker-bench-security
# Analyse d'une image avec Dive
dive nginx:alpine
# Nettoyage programm  (cron)
# 0 2 * * * docker system prune -af --filter "until=168h"
# Backup automatique des volumes
#!/bin/bash
for volume in $(docker volume ls -q); do
  docker run --rm \
    -v $volume:/source:ro \
    -v /backup:/backup \
    alpine \
    tar czf /backup/$volume-$(date +\%Y\%m\%d).tar.gz -C /source .
done
```

Définitions Essentielles

Conteneur Unité logicielle légère et portable qui encapsule une application et ses dépendances.

Image Docker Fichier immuable contenant le code, les bibliothèques et les configurations nécessaires pour exécuter une application.

Layer Chaque instruction du Dockerfile crée une couche immuable dans l'image (principe UnionFS).

Volume Mécanisme de persistance des données géré par Docker, indépendant du cycle de vie du conteneur.

Bind Mount Montage direct d'un répertoire hôte dans un conteneur.

Registry Dépôt centralisé pour stocker et distribuer des images Docker.

Daemon (dockerd) Service système qui gère les conteneurs, images, volumes et réseaux.

Namespace Fonctionnalité du noyau Linux permettant l'isolation des ressources (PID, NET, MNT, etc.).

Cgroup Mécanisme Linux pour limiter et monitorer l'usage des ressources (CPU, RAM, I/O).

Overlay Network
: Réseau virtuel multi-hôtes utilisé dans Docker Swarm pour la communication entre nœuds.

Healthcheck : Instruction vérifiant périodiquement l'état de santé d'un conteneur.

Multi-stage Build
: Technique permettant de séparer la phase de compilation de l'exécution pour réduire la taille de l'image.

Docker Content Trust
: Système de signature cryptographique garantissant l'intégrité des images.

User Namespace
: Isolation des UID/GID entre hôte et conteneur pour renforcer la sécurité.

Swarm Mode : Mode d'orchestration native de Docker pour gérer des clusters de conteneurs.

Pourquoi chaque étape ?

Multi-stage build
: Réduit la taille finale de l'image en éliminant les outils de compilation inutiles en production.

Utilisateur non-root
: Évite l'exécution avec des privilèges élevés, limitant les risques en cas de compromission.

Scanning de sécurité
: Détecte les vulnérabilités avant déploiement, respecte le principe *Shift-Left Security*.

Volumes nommés
: Garantit la persistance des données et facilite les sauvegardes/rotations.

Réseaux privés (internal : true)
: Isoler les services backend du monde extérieur, principe de moindre privilège.

Healthcheck : Permet à l'orchestrateur de redémarrer automatiquement un conteneur défaillant.

Tags spécifiques (v1.2.3)
: Assure la reproductibilité et évite les comportements imprévisibles avec `latest`.

Docker Content Trust
: Vérifie l'intégrité et l'authenticité des images, évitant les attaques de type *supply chain*.

Limitation des ressources (cgroups)
: Empêche un conteneur de monopoliser les ressources de l'hôte.

Monitoring (Prometheus + Grafana)
: Fournit une observabilité complète pour détecter les anomalies en temps réel.

Pipeline CI/CD automatisé
Garantit que chaque changement est testé, scanné et déployé de manière reproductible.

Rolling updates (Swarm)
Permet des mises à jour sans interruption de service (zero-downtime).

Glossaire

Container	Unité d'exécution isolée contenant une application et ses dépendances
Image	Template immuable utilisé pour créer des conteneurs
Dockerfile	Script décrivant la construction d'une image
Registry	Système de stockage et distribution d'images
Volume	Mécanisme de persistance des données
Network	Infrastructure virtuelle connectant les conteneurs
Compose	Outil de définition d'applications multi-conteneurs
Swarm	Solution d'orchestration native de Docker
Layer	Couche d'une image représentant une instruction du Dockerfile
Daemon	Service système gérant les conteneurs (dockerd)
OCI	Open Container Initiative - standard de conteneurs

Overlay Network
Réseau virtuel multi-hôtes

Health Check
Test vérifiant l'état de santé d'un conteneur

Conclusion et Perspectives

Docker a transformé la manière dont nous développons, déployons et opérons les applications modernes. Cette documentation complète couvre l'ensemble des aspects nécessaires pour maîtriser Docker en environnement professionnel.

Points Clés à Retenir

— La containerisation améliore la portabilité et la cohérence des déploiements
— La sécurité doit être intégrée dès la phase de construction des images
— L'optimisation des images réduit les coûts et améliore les performances
— Le monitoring et le logging sont essentiels en production
— L'orchestration (Swarm/Kubernetes) est nécessaire à grande échelle
— Les bonnes pratiques DevOps s'appliquent naturellement avec Docker

Évolutions Futures

L'écosystème Docker continue d'évoluer avec :

— **WebAssembly** : Support natif de Wasm dans les conteneurs
— **Rootless mode** : Exécution sans privilèges par défaut
— **BuildKit** : Builder de nouvelle génération plus performant
— **Docker Extensions** : Écosystème de plugins pour Docker Desktop
— **Supply Chain Security** : Traçabilité complète des artefacts

Prochaines Étapes

Pour approfondir vos connaissances :

1. Pratiquer avec des projets réels
2. Explorer Kubernetes pour l'orchestration avancée
3. Implémenter des pipelines CI/CD complets
4. Contribuer à l'écosystème open source
5. Obtenir des certifications (Docker Certified Associate)

Documentation rédigée par CHERIF Maryem
Ingénieure en Cybersécurité
- 9 novembre 2025

Pour toute question ou suggestion : cherif.maryem24@gmail.com