Les commandes Docker

- Les commandes utiles
- Gestion du réseau et des ports Docker

Les commandes utiles

Si docker ne démarre pas

```
sudo dockerd --debug
```

Lister les conteneurs

```
sudo docker ps -a
```

Afficher les stats des conteneurs

```
sudo docker stats
```

Pour afficher les stats d'un seul conteneur :

```
sudo docker stats <container_name>
```

Entrer dans un conteneur

```
docker exec -it <container_name> bash
```

> A savoir : Vous ne pourrez pas rentrer dans un conteneur s'il est pas allumé.

se connecter en tant que

```
sudo docker exec -it -u <userId> <container_name> bash
```

Lancer un conteneur

```
sudo docker start <container_name>
```

Voir les logs d'un conteneur

```
sudo docker logs <container_name>
```

Renommer un conteneur

```
sudo docker rename <old-name> <new-name>
```

Lancer une commande dans un conteneur

```
sudo docker exec <container-name> <command>
```

Afficher l'interface d'un conteneur

```
root@server:/ docker exec -it <CONTAINER> ls /sys/class/net/
eth0  lo
root@server:/ docker exec -it <CONTAINER> cat /sys/class/net/eth0/iflink
38
```

Copier un fichier dans un conteneur

```
# Syntaxe:
docker cp </path/to/file/in/host> <container_name>:<path_to_volume>

# Exemple:
docker cp /var/backup/dump.sql my_postgres_1:/backups
```

Gestion du réseau et des ports Docker

> Cette page est dédier à la question suivante :
>
> Comment bien configuré ses conteneurs sur le réseau ?

Portée : Docker Compose, hosts Linux avec UFW comme pare-feu, reverse proxy type Traefik/Caddy/nginx.
Objectif : standardiser une architecture réseau 100 % maîtrisée sur toutes les stacks compose.

1. Comportement par défaut de Docker

1.1 Réseaux créés automatiquement

Par défaut, `docker compose up` crée **un réseau bridge dédié par projet** (nommé `<nom_projet>_default`), distinct du bridge `docker0` historique.
Tous les services d'un même fichier compose sont attachés à ce réseau et se résolvent entre eux **par nom de service** (DNS interne Docker), sans avoir besoin de publier le moindre port.

➡ Conséquence clé : **la communication inter-conteneurs ne nécessite jamais `ports:`**.
`ports:` ne sert qu'à exposer un service **en dehors du host Docker** (vers le LAN/WAN).

1.2 `EXPOSE` vs `ports:`

Directive	Effet réel	Portée
`EXPOSE` (Dockerfile)	Documentation uniquement, aucun binding réseau réel	N/A
`expose:` (compose)	Idem, informatif	N/A
`ports:` (compose)	Publie réellement le port sur une interface du **host** via NAT	Host + extérieur

1.3 Docker et iptables : le piège UFW

C'est le point le plus critique en environnement Ops/DevSec.

Quand un conteneur publie un port avec `ports:`, Docker insère automatiquement des règles dans les chaînes **`DOCKER`** et **`DOCKER-USER`** de la table `nat`/`filter` d'iptables — **en amont** de la chaîne `INPUT` gérée par UFW.

> **Conséquence : un port publié via `ports:` est accessible depuis l'extérieur même si UFW le bloque explicitement.**
> UFW ne voit jamais passer ce trafic car Docker route directement via NAT/FORWARD.

```
Trafic entrant → PREROUTING (Docker DNAT) → FORWARD → DOCKER-USER → DOCKER → conteneur
                                                        ↑
                                        UFW (INPUT) n'intercepte PAS ce chemin
```

---

2. Le comportement de `ports:` en détail

2.1 Syntaxes

```
ports:
  - "8080:80"            # host:container — bind 0.0.0.0 (TOUTES interfaces) par défaut, port
ouvert sur iptables
  - "127.0.0.1:8080:80"  # bind explicite localhost uniquement
  - "10.10.10.1:8080:80" # bind sur une IP précise (ex: interface WireGuard)
  - "8080:80/udp"        # protocole explicite, port ouvert sur iptables
```

2.2 Le piège du binding implicite

`"8080:80"` publie sur **`0.0.0.0:8080`**, donc sur **toutes les interfaces réseau du host** (LAN, WAN, VPN...), pas seulement en local.

> **C'est la cause n°1 des services/ports exposées par erreur sur Internet.**

---

3. Bonnes pratiques Ops / DevSec

- **Principe de moindre exposition** : ne publier (`ports:`) que ce qui doit être joignable depuis l'extérieur du host. Tout le reste communique via le réseau interne Docker (DNS de service).
- **Un réseau dédié par stack**, jamais le réseau `default` partagé pour plusieurs projets sans raison. Nommer explicitement :

```
networks:
    proxy-web:
      name: proxy-public # Destiné à exposé les services sur les ports 80/443
    backend:
      name: <stack>-internal # Destiné a rester en local, jamais exposé publiquement mais
reste accessbles pour les autres services internes
      internal: true
```

Type d’exposition	Type de service
`internal: true`	DB / cache / workers / Clamav
`proxy-public`	Traefik/ Caddy/ nginx
`network_mode: host`	VPN

Éviter **`network_mode: host`** sauf nécessité technique réelle (ex: WireGuard qui a besoin d'accéder aux interfaces réseau du host). Cas contraire, `host` désactive toute l'isolation réseau Docker.

Convention de nommage des réseaux cohérente sur toutes les stacks (`<stack>-internal`, `proxy-public`, `vpn-admin`...) pour faciliter l'audit avec `docker network ls`.

---

4. Commandes utiles pour l'audit

4.1 Depuis Docker

```
# Ports publiés par tous les conteneurs actifs
docker ps --format "table {{.Names}}\t{{.Ports}}"

# Ports publiés par un conteneur précis
docker port <container>
```

```
# Lister les réseaux et repérer les réseaux "trop partagés"
docker network ls

# Détail d'un réseau : conteneurs attachés, subnet, internal ou non
docker network inspect <network>

# Vue globale via l'API compose
docker compose ps
```

4.2 Depuis le host (vérité terrain, indépendante de Docker)

```
# Tous les sockets en écoute, avec le PID/process
ss -tulpn

# Équivalent netstat
sudo netstat -tulpn

# Règles NAT injectées par Docker (bindings réels)
sudo iptables -t nat -L DOCKER -n --line-numbers

# Règles de filtrage custom (celles qui priment sur UFW)
sudo iptables -L DOCKER-USER -n --line-numbers

# Vérification externe depuis une autre machine (la vraie preuve)
nmap -Pn -p- <ip_du_host>
```

> `iptables` sur le host est la seule vérité fiable : la lecture des `docker-compose.yml` seule ne garantit pas ce qui est réellement exposé (héritage de configs, overrides, etc.).

Explication de la sortie affichée par la commande `sudo iptables -t nat -L DOCKER -n -v --line-numbers` :
Lecture des colonnes (avec `-v`, qui ajoute compteurs et interfaces — utile pour vérifier qu'une règle est réellement traversée) :

Colonne	Signification
`num`	Numéro de la règle dans la chaîne (utile pour cibler une règle précise avec `-D DOCKER-USER <num>`)

Colonne	Signification
`pkts`	Nombre de paquets ayant matché cette règle depuis le dernier reset des compteurs — un `0` persistant indique une règle jamais déclenchée (donc potentiellement inutile ou mal placée)
`bytes`	Volume total de données ayant matché la règle (utile pour repérer un trafic anormal sur un port)
`target`	Action appliquée : `DNAT` (redirection vers l'IP/port du conteneur), `ACCEPT`, `DROP`, `RETURN`, ou le nom d'une sous-chaîne (ex: un nom de réseau Docker généré)
`prot`	Protocole concerné : `tcp`, `udp`, ou `all`
`opt`	Options additionnelles de la règle (généralement `--` si aucune)
`in`	Interface réseau d'entrée du paquet (ex: `eth0`, `wg0`, `br-xxxxx`) — `*` = toutes interfaces
`out`	Interface réseau de sortie — `*` = toutes interfaces
`source`	Adresse IP/subnet source du paquet — `0.0.0.0/0` = n'importe quelle source
`destination`	Adresse IP/subnet de destination — `0.0.0.0/0` = n'importe quelle destination
`to:` (en fin de ligne, table `nat`)	Présent uniquement sur les règles `DNAT` : indique l'IP:port réel du conteneur vers lequel le trafic est redirigé (ex: `to:172.17.0.3:80`) — c'est cette colonne qui révèle le binding effectif d'un `ports:`

Point de vigilance particulier : sur la chaîne `DOCKER` (table `nat`), une règle avec `source 0.0.0.0/0` et une colonne `to:` renseignée confirme qu'un port est bien publié en NAT vers un conteneur — indépendamment de ce qu'affiche UFW, qui n'intercepte pas ce chemin.

---

5 Exemples par cas d'usage

5.1 Postgres non exposé

```
services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/pg_password
    networks:
      - bdd-internal
```

```
    # AUCUN "ports:" — accessible uniquement via le réseau interne

  app:
    image: mon-app
    networks:
      - bdd-internal
    depends_on:
      - postgres
    # se connecte à "postgres:5432" via le DNS interne Docker

networks:
  bdd-internal:
    internal: true   # aucune route sortante/entrante possible
```

5.2 Reverse proxy avec réseau dédié

```
services:
  traefik:
    image: traefik:v3
    ports:
      - "80:80"
      - "443:443"        # seul conteneur exposé sur 0.0.0.0
    networks:
      - proxy-public
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro

  webapp:
    image: mon-webapp
    networks:
      - proxy-public   # visible par Traefik
      - bdd-internal   # accès DB
    labels:
      - traefik.enable=true
      - traefik.http.routers.webapp.rule=Host(`app.example.com`)
    # AUCUN "ports:" sur webapp — tout passe par Traefik

  postgres:
    image: postgres:16
    networks:
```

```
      - bdd-internal   # jamais sur proxy-public


networks:
  proxy-public:
    name: proxy-public
    external: true   # réseau partagé, créé une fois, rejoint par chaque stack
  bdd-internal:
    internal: true
```

Le service `webapp` n'a pas de `ports:` du tout : il n'est joignable que via Traefik, lui-même seul point d'entrée public du host.

Cas particulier : le reverse proxy tourne **directement sur le host** (pas en conteneur). Il ne peut pas "rejoindre" un network Docker comme le ferait un conteneur — deux approches valables selon le niveau de rigueur voulu.

Solution : Chaque service publie son port uniquement sur `127.0.0.1`, Apache proxy vers localhost :

```
services:
  webapp:
    image: mon-webapp
    ports:
      - "127.0.0.1:8081:80"   # jamais 0.0.0.0
    networks:
      - bdd-internal
```

```
<VirtualHost *:443>
    ServerName app.example.com
    ProxyPass / http://127.0.0.1:8081/
    ProxyPassReverse / http://127.0.0.1:8081/
</VirtualHost>
```

Avantage : simple, aucune règle iptables custom nécessaire (un socket bindé en loopback n'est routable par personne d'autre que le host lui-même).

5.3 VPN WireGuard : accès admin sans passer par le reverse proxy

> Cas où un service d'administration (Portainer, dashboard interne, Home Assistant...) ne doit être joignable **que via le tunnel WireGuard**, jamais via Internet public, même pas via le reverse proxy.

```
services:
  wireguard:
    image: linuxserver/wireguard
    network_mode: host   # nécessaire pour le routing VPN natif
    cap_add:
      - NET_ADMIN
      - SYS_MODULE
    volumes:
      - ./wg-config:/config

  admin-panel:
    image: mon-admin-panel
    ports:
      - "10.10.10.1:9000:9000"   # IP wireguard + bind UNIQUEMENT sur l'IP de l'interface wg0
    networks:
      - backend
```

Complément côté firewall (le bind seul limite déjà l'exposition, mais on verrouille aussi explicitement) :

```
# Autoriser uniquement depuis le subnet WireGuard
sudo iptables -I DOCKER-USER -i wg0 -s 10.10.10.0/24 -j ACCEPT
sudo iptables -I DOCKER-USER -p tcp --dport 9000 -j DROP
```

5.4 Debug local uniquement

Pattern générique pour tout service qu'on veut inspecter temporairement sans l'exposer :

```
    ports:
      - "127.0.0.1:PORT:PORT"
```

> Accessible uniquement en `curl localhost:PORT` depuis le host lui-même — jamais depuis le LAN, le WAN ou le VPN.