Cours DevOps - Partie 4 : Orchestration avec Docker Swarm

Vue d'ensemble - Points abordés dans cette Partie 4

Aperçu rapide de tout ce qui est vu aujourd'hui

📚 Théories

- **Introduction à l'orchestration**
  - Limites de docker-compose en production
  - Problématiques de scalabilité et haute disponibilité
  - Pourquoi l'orchestration est nécessaire
- **Docker Swarm vs docker-compose vs Kubernetes**
  - Comparaison détaillée des trois outils
  - Cas d'usage appropriés pour chacun
  - Quand choisir Swarm plutôt qu'autre chose
- **Architecture Docker Swarm**
  - Concepts de cluster et nodes (manager vs worker)
  - Services vs conteneurs classiques
  - Tasks et leur cycle de vie
  - Raft consensus algorithm
  - Ingress routing mesh
- **Overlay Networks**
  - Communication inter-nodes
  - Différence avec bridge networks
  - Isolation et sécurité
- **Services Swarm**
  - Modes de déploiement (replicated vs global)
  - Scaling horizontal
  - Load balancing automatique
  - Rolling updates et rollbacks
- **Stacks Swarm**
  - Format docker-compose.yml pour Swarm
  - Différences avec docker-compose classique
  - Gestion de stacks complètes
- **Secrets et Configs Swarm**
  - Gestion sécurisée des secrets

    - Distribution des configurations
    - Différence avec variables d'environnement
- **Placement et Constraints**
    - Contrôler où les services s'exécutent
    - Labels et contraintes
    - Affinités et anti-affinités

🛠️ Mises en pratique

- **Initialisation d'un Swarm**
    - Premier swarm en mode single-node
    - Ajout de nodes (simulation)
    - Inspection du cluster
- **Premiers services**
    - Créer un service nginx simple
    - Scaling manuel avec replicas
    - Inspection des tasks
    - Accès via ingress routing mesh
- **Stack multi-services**
    - Déploiement d'une stack avec docker stack deploy
    - Services interconnectés (API + database)
    - Networks overlay
    - Volumes pour persistance
- **Rolling updates**
    - Mise à jour progressive d'un service
    - Configuration de l'update policy
    - Rollback en cas d'erreur
- **Secrets Swarm**
    - Création de secrets
    - Utilisation dans les services
    - Rotation de secrets
- **Placement constraints**
    - Labels sur nodes
    - Contraintes de déploiement
    - Services globaux vs replicated
- **Projet d'après-midi : API Flask complète**
    - Backend Flask avec endpoints REST
    - PostgreSQL avec volume persistant

- Redis pour cache
- Frontend nginx statique
- Déploiement complet en Swarm
- Scaling et résilience
- Secrets pour credentials
- Rolling updates

---

1 - Introduction : Pourquoi l'Orchestration ?

L'essentiel

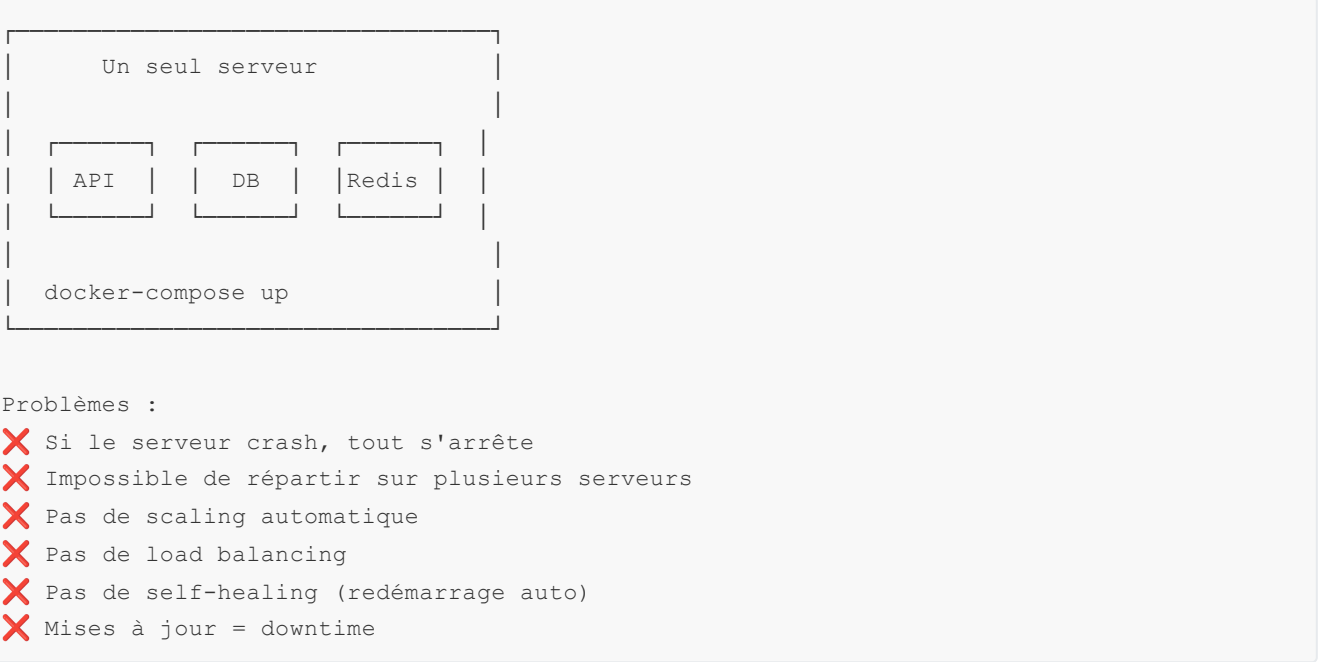
Le problème avec docker-compose en production

Vous avez appris à utiliser `docker-compose` pour orchestrer plusieurs conteneurs sur une seule machine. C'est parfait pour le développement et les petits projets.

Mais qu'arrive-t-il en production quand :

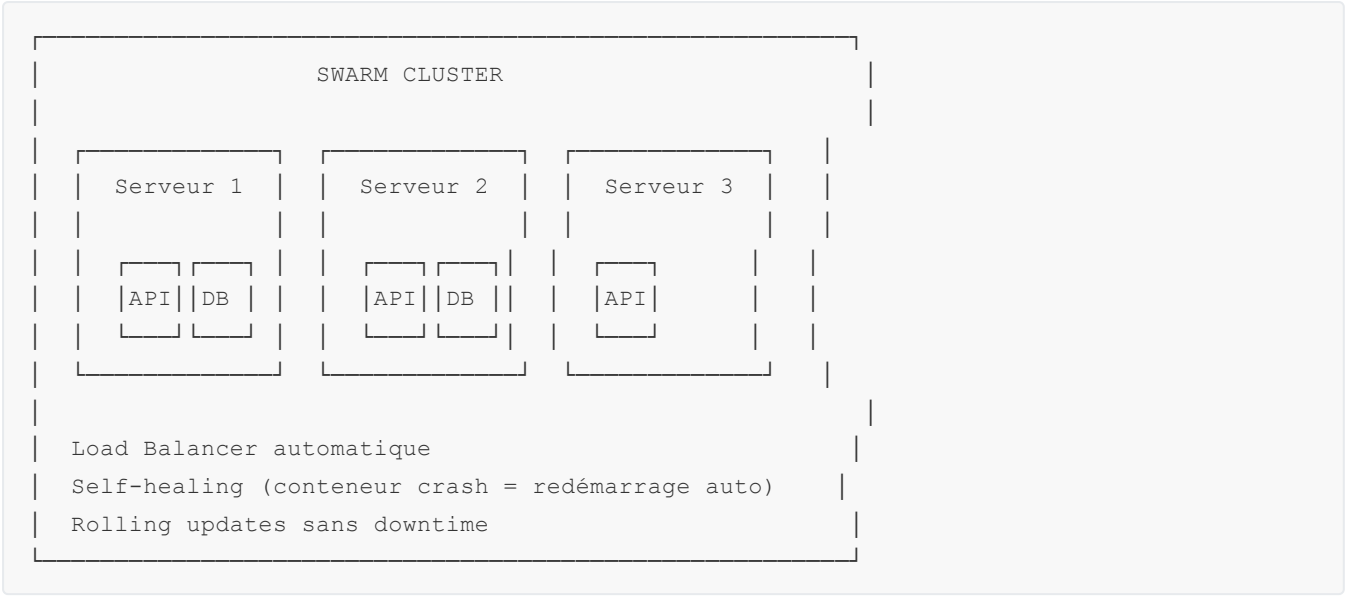
- Votre application reçoit 10,000 requêtes par seconde ?
- Votre serveur tombe en panne ?
- Vous devez faire une mise à jour sans downtime ?
- Vous avez besoin de plus de puissance (scale horizontal) ?

Limites de docker-compose :



Ce que l'orchestration apporte

L'orchestration résout ces problèmes en gérant automatiquement vos conteneurs à travers **plusieurs serveurs**.



Avantages :

✅ **Haute disponibilité** : Si un serveur tombe, les autres prennent le relais
✅ **Scalabilité horizontale** : Ajouter des serveurs pour plus de puissance
✅ **Load balancing** : Répartition automatique du trafic
✅ **Self-healing** : Redémarrage automatique des conteneurs qui crashent
✅ **Rolling updates** : Mises à jour sans interruption de service
✅ **Déclaratif** : Vous déclarez l'état souhaité, l'orchestrateur s'en occupe

Cas d'usage concrets

Scénario 1 : E-commerce en période de soldes

```
Trafic normal : 100 req/s
└─> 3 réplicas de l'API suffisent

Black Friday : 10,000 req/s
└─> Swarm scale automatiquement à 50 réplicas
└─> Répartis sur 10 serveurs
└─> Retour à la normale après les soldes
```

Scénario 2 : Mise à jour sans downtime

```
Version 1.0 de l'API en production (10 réplicas)

Déploiement version 1.1 avec rolling update :
1. Démarre 2 réplicas v1.1
2. Vérifie healthcheck OK
3. Stop 2 réplicas v1.0
4. Répète jusqu'à 100% v1.1
5. Aucune interruption de service
```

Scénario 3 : Panne serveur

```
Serveur 2 tombe en panne
└─> Les conteneurs qui y tournaient sont automatiquement
    redémarrés sur Serveur 1 et Serveur 3
└─> Aucune intervention manuelle
└─> Service continue de fonctionner
```

Philosophie déclarative

Avec l'orchestration, vous ne dites plus **"comment faire"**, mais **"ce que vous voulez"**.

Sans orchestration (impératif) :

```bash
# Vous devez tout faire manuellement
docker run -d --name api1 mon-api
docker run -d --name api2 mon-api
docker run -d --name api3 mon-api

# Si un conteneur crash
docker ps  # vérifier
docker start api2  # redémarrer manuellement

# Pour une mise à jour
docker stop api1
docker rm api1
docker run -d --name api1 mon-api:v2
# ... répéter pour tous les conteneurs
```

Avec orchestration (déclaratif) :

```
# Vous déclarez l'état souhaité
docker service create --name api --replicas 3 mon-api

# Swarm s'occupe de TOUT :
# - Créer 3 conteneurs
# - Les répartir sur le cluster
# - Redémarrer automatiquement si crash
# - Load balancer le trafic

# Mise à jour en une commande
docker service update --image mon-api:v2 api
# Swarm gère le rolling update automatiquement
```

---

2 - Comparaison : docker-compose vs Swarm vs Kubernetes

L'essentiel

Tableau comparatif

Critère	docker-compose	Docker Swarm	Kubernetes
Complexité	⭐ Très simple	⭐⭐ Moyenne	⭐⭐⭐⭐⭐ Complexe
Courbe d'apprentissage	Quelques heures	1-2 jours	Plusieurs semaines
Multi-serveurs	❌ Non (single host)	✅ Oui	✅ Oui
Scaling automatique	❌ Non	⚠️ Manuel	✅ Automatique (HPA)
Load balancing	❌ Non	✅ Intégré (ingress mesh)	✅ Intégré
Self-healing	⚠️ restart policies	✅ Oui	✅ Oui
Rolling updates	❌ Non	✅ Oui	✅ Oui
Secrets management	⚠️ Fichiers + .gitignore	✅ Secrets intégrés	✅ Secrets intégrés
Écosystème	Limité	Docker ecosystem	Énorme (Helm, operators, etc)
Monitoring	Extensions tierces	Basique	Avancé (Prometheus, etc)
Cas d'usage	Dev, petits projets	Prod PME, équipes Docker	Prod grande échelle
Installation	Inclus avec Docker	Inclus avec Docker	Installation séparée
Compatibilité Docker	Natif	Natif (API Docker)	Via CRI (abstraction)

Quand utiliser quoi ?

Utilisez docker-compose quand :

```
✅ Environnement de développement local
✅ Petits projets (< 10 services)
✅ Déploiement sur un seul serveur
✅ Prototypes et démos
✅ Tests d'intégration en CI/CD

Exemple : Projet personnel, startup MVP, environnement de dev
```

Utilisez Docker Swarm quand :

```
✅ Besoin d'orchestration simple et rapide
✅ Équipe déjà familière avec Docker
✅ Production PME (jusqu'à ~100 nodes)
✅ Pas besoin de features Kubernetes avancées
✅ Vous voulez garder la simplicité Docker

Exemple : Application d'entreprise moyenne, SaaS B2B, API interne
```

Utilisez Kubernetes quand :

```
✅ Production à grande échelle (>100 nodes)
✅ Multi-cloud ou cloud provider (AWS EKS, GCP GKE, Azure AKS)
✅ Besoin de l'écosystème (Istio, Helm, operators)
✅ Auto-scaling avancé nécessaire
✅ Équipe DevOps dédiée

Exemple : Google, Netflix, Spotify, grandes entreprises
```

Exemple concret : même stack dans les 3 outils

API + PostgreSQL + Redis

docker-compose.yml :

```yaml
version: '3.8'

services:
  api:
    image: mon-api:latest
    ports:
      - "3000:3000"
    depends_on:
      - db
      - redis

  db:
    image: postgres:16-alpine
    environment:
```

```
      POSTGRES_PASSWORD: secret
    volumes:
      - db_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine

volumes:
  db_data:
```

Docker Swarm (stack.yml) :

```
version: '3.8'

services:
  api:
    image: mon-api:latest
    deploy:
      replicas: 3                    # 3 instances pour HA
      update_config:
        parallelism: 1               # Rolling update
        delay: 10s
    ports:
      - "3000:3000"
    depends_on:
      - db
      - redis
    secrets:
      - db_password

  db:
    image: postgres:16-alpine
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager      # DB sur manager node
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password
    volumes:
      - db_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine
    deploy:
      replicas: 2

secrets:
  db_password:
    external: true
```

```
volumes:
  db_data:
```

Kubernetes (équivalent) :

Nécessite plusieurs fichiers YAML (Deployment, Service, ConfigMap, Secret, PersistentVolume, etc.)

```
# api-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: api
        image: mon-api:latest
        ports:
        - containerPort: 3000
---
# api-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  type: LoadBalancer
  ports:
  - port: 3000
    targetPort: 3000
  selector:
    app: api
# ... et encore 5-6 fichiers pour db, redis, secrets, etc.
```

Observation :

- docker-compose : Le plus simple, mais limité
- Swarm : Juste un peu plus verbose que compose, mais beaucoup plus puissant
- Kubernetes : Beaucoup plus de fichiers et complexité, mais très flexible

Migration progressive

Un parcours typique pour une entreprise :

```
Phase 1 : Développement
└─> docker-compose sur machines des devs

Phase 2 : Premiers déploiements
└─> docker-compose sur un VPS
└─> Limites atteintes (traffic, disponibilité)

Phase 3 : Croissance
└─> Migration vers Docker Swarm
└─> Plusieurs serveurs, HA, scaling

Phase 4 : Grande échelle (optionnel)
└─> Migration vers Kubernetes
└─> Multi-cloud, auto-scaling, écosystème avancé
```

Bonne nouvelle : Le fichier docker-compose.yml est compatible à ~90% avec Swarm. La migration est facile !

Pourquoi apprendre Swarm ET Kubernetes ?

Swarm :

- Plus simple à comprendre (bonne introduction à l'orchestration)
- Concepts transférables à Kubernetes
- Utilisé dans beaucoup de PME
- Parfait pour comprendre les principes d'orchestration

Kubernetes (jour suivant) :

- Standard de l'industrie
- Demandé par beaucoup d'entreprises
- Écosystème riche

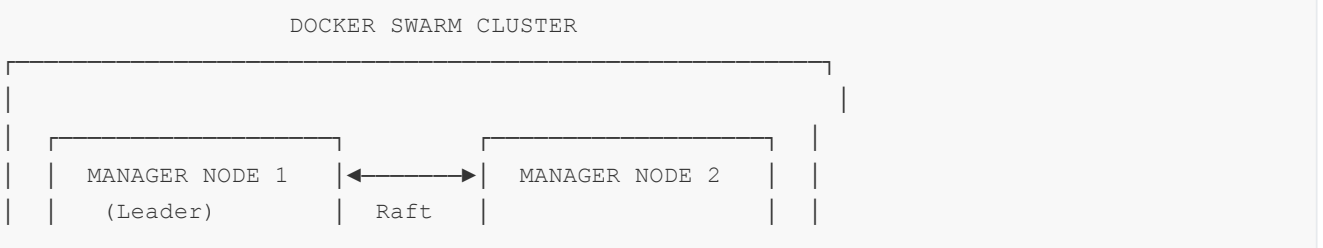
Apprendre Swarm d'abord rend Kubernetes beaucoup plus facile à comprendre ensuite.

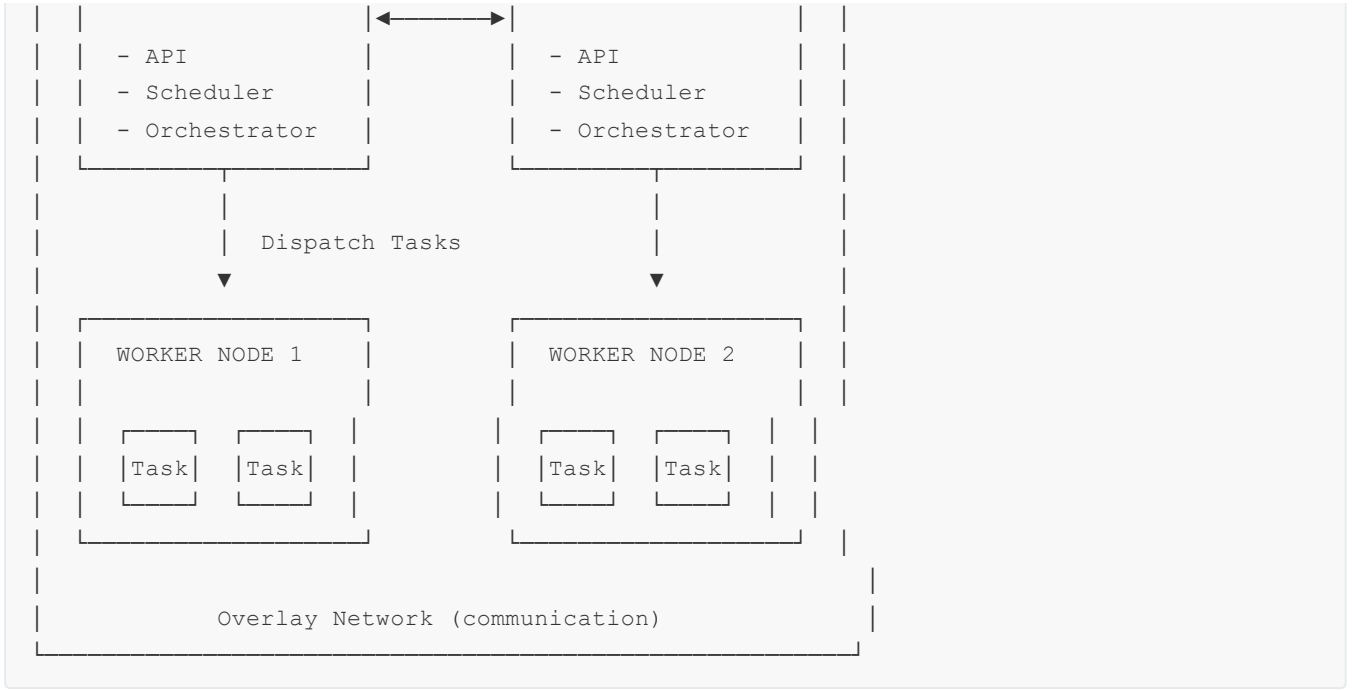
---

3 - Architecture Docker Swarm

L'essentiel

Vue d'ensemble d'un cluster Swarm





Nodes : Manager vs Worker

Un **node** est un serveur (physique ou virtuel) qui fait partie du cluster Swarm.

Manager Nodes :

```
Rôles :
✅ Gèrent l'état du cluster
✅ Prennent les décisions d'orchestration
✅ Acceptent les commandes docker service/stack
✅ Maintiennent le consensus (Raft)
✅ Peuvent aussi exécuter des tasks (par défaut)

Recommandations :
- Nombre impair (1, 3, 5, 7)
- Minimum 3 en production (tolérance de panne)
- Maximum 7 (overhead du consensus)
```

Worker Nodes :

```
Rôles :
✅ Exécutent les tasks (conteneurs)
✅ Rapportent leur état au manager
✅ Ne participent pas aux décisions

Recommandations :
- Autant que nécessaire (scalabilité horizontale)
- Peuvent être ajoutés/retirés dynamiquement
```

Exemple de topologie production :

```
3 Manager Nodes  → Haute disponibilité (1 peut tomber)
10 Worker Nodes  → Exécution des charges de travail
```

Services : le concept central

Un **service** est une abstraction qui définit comment vos conteneurs doivent tourner dans le cluster.

Différence service vs conteneur :

```
Conteneur classique (docker run) :
└─> Tourne sur UNE machine
└─> Vous gérez manuellement le cycle de vie
└─> Pas de scaling automatique
└─> Pas de répartition de charge

Service Swarm (docker service create) :
└─> Distribué sur PLUSIEURS machines
└─> Swarm gère automatiquement le cycle de vie
└─> Scaling déclaratif (replicas)
└─> Load balancing automatique
```

Deux modes de services :

1. Replicated (par défaut) :

```yaml
services:
  api:
    image: mon-api
    deploy:
      replicas: 5  # Exactement 5 instances dans le cluster
```

```
Manager décide où placer les 5 réplicas :
Node 1: [API] [API]
Node 2: [API] [API]
Node 3: [API]
```

2. Global (un par node) :

```yaml
services:
  monitoring:
    image: node-exporter
    deploy:
      mode: global  # Une instance par node
```

```
Chaque node a exactement 1 instance :
Node 1: [Monitoring]
Node 2: [Monitoring]
Node 3: [Monitoring]
```

Tasks : les unités d'exécution

Une **task** est l'unité atomique d'un service. C'est un conteneur + sa configuration.

```
Service "api" avec 3 replicas
├─> Task 1 → Conteneur sur Node 1
├─> Task 2 → Conteneur sur Node 2
└─> Task 3 → Conteneur sur Node 3
```

Cycle de vie d'une task :

```
NEW → PENDING → ASSIGNED → PREPARING → STARTING → RUNNING
                                                    │
                                                    ├─> COMPLETE (succès)
                                                    ├─> FAILED (échec)
                                                    └─> SHUTDOWN (arrêt)
```

Si une task échoue, Swarm en démarre automatiquement une nouvelle ailleurs (self-healing).

Raft Consensus Algorithm

Les managers utilisent **Raft** pour rester synchronisés et élire un leader.

```
3 Managers : [M1-Leader] [M2-Follower] [M3-Follower]

Écriture (docker service create) :
1. Commande arrive sur n'importe quel manager
2. Forwarded au leader
3. Leader propose aux followers
4. Majorité doit accepter (quorum)
5. Changement appliqué

Si le leader tombe :
1. Followers détectent l'absence
2. Élection d'un nouveau leader (<2s)
3. Cluster continue de fonctionner
```

Quorum : Majorité nécessaire pour les décisions

```
1 manager  → quorum = 1  (pas de tolérance de panne)
3 managers → quorum = 2  (tolère 1 panne)
5 managers → quorum = 3  (tolère 2 pannes)
7 managers → quorum = 4  (tolère 3 pannes)
```

Plus de managers = plus de tolérance, mais plus d'overhead.

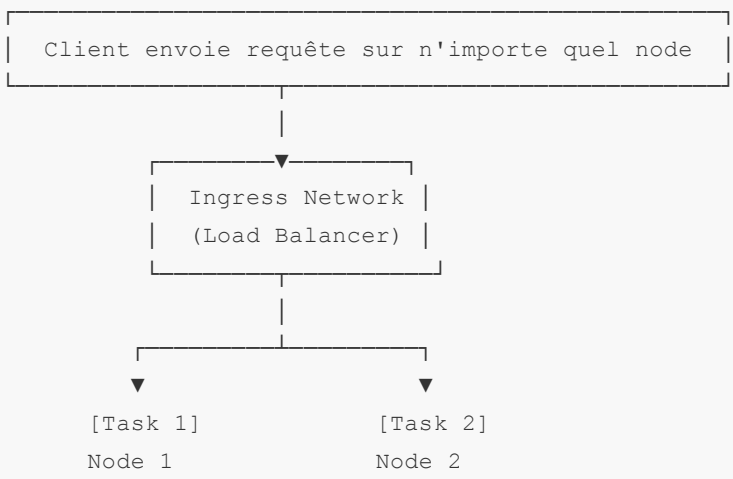
Ingress Routing Mesh

Le **routing mesh** permet d'accéder à un service sur **n'importe quel node**, même si le conteneur ne tourne pas dessus.

```
Service "api" exposé sur port 8080 :
Node 1: [API]
Node 2: [API]
Node 3: []        ← Pas d'instance ici

Client se connecte à Node 3:8080
└─> Routing mesh redirige automatiquement vers Node 1 ou 2
└─> Transparent pour le client
```

Fonctionnement :



Tous les nodes écoutent sur le port publié, et le routing mesh balance la charge.

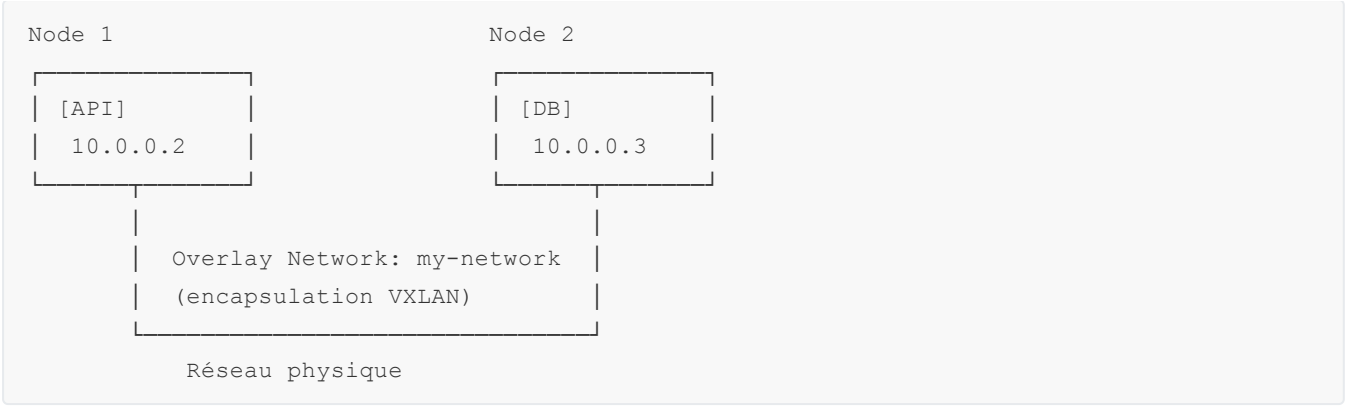
Overlay Networks

Les **overlay networks** permettent aux conteneurs sur différents nodes de communiquer comme s'ils étaient sur la même machine.

```
Sans overlay (bridge network classique) :
Node 1: [API] ─────X──────> [DB sur Node 2]  ❌ Impossible

Avec overlay network :
Node 1: [API] ──overlay──> [DB sur Node 2]  ✅ Fonctionne
```

Fonctionnement :



Swarm gère automatiquement l'encapsulation et le routage.

---

4 - Initialiser votre Premier Swarm

L'essentiel

Mode single-node pour commencer

On va commencer simple : un Swarm avec un seul node (votre machine).

C'est parfait pour :

- Apprendre les concepts
- Tester localement
- Développer et valider des stacks

Plus tard, on verra comment ajouter des nodes.

Initialiser le Swarm

```
docker swarm init
```

Sortie :

```
Swarm initialized: current node (abc123xyz) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-xxx... 192.168.1.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

Ce qui vient de se passer :

✅ Votre machine est maintenant un Swarm manager
✅ Un cluster à 1 node est créé
✅ Vous pouvez créer des services
✅ Un token d'adhésion est généré (pour ajouter d'autres nodes)

Vérifier l'état du Swarm

```
# Lister les nodes
docker node ls
```

Sortie :

```
ID              HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS
abc123xyz *     laptop     Ready     Active         Leader
```

Colonnes importantes :

- `*` : Node actuel (celui sur lequel vous êtes)
- `STATUS` : Ready = node opérationnel
- `AVAILABILITY` : Active = accepte des tasks
- `MANAGER STATUS` : Leader = manager principal

Inspecter le node

```
docker node inspect self --pretty
```

Affiche des infos détaillées : OS, CPU, RAM, labels, etc.

EXERCICE GUIDÉ 1 : Premier Swarm et exploration

Objectif : Initialiser un Swarm et comprendre sa structure

Étape 1 : Vérifier que Swarm n'est pas actif

```
docker info | grep Swarm
```

Devrait afficher :

```
Swarm: inactive
```

Étape 2 : Initialiser le Swarm

```
docker swarm init
```

Si vous avez plusieurs interfaces réseau, Docker pourrait demander :

```
docker swarm init --advertise-addr <votre_ip>
```

Étape 3 : Vérifier que Swarm est actif

```
docker info | grep Swarm
```

Devrait afficher :

```
Swarm: active
```

Étape 4 : Lister les nodes

```
docker node ls
```

Vous devriez voir votre machine comme seul node, avec le statut Leader.

Étape 5 : Inspecter le node

```
# Pretty format
docker node inspect self --pretty

# Format JSON (plus de détails)
docker node inspect self
```

Observer les informations :

- Hostname
- Rôle (Manager)
- État (Ready)
- Ressources disponibles

Étape 6 : Obtenir le token pour ajouter un worker

```
docker swarm join-token worker
```

Affiche la commande pour rejoindre le Swarm en tant que worker.

Étape 7 : Obtenir le token pour ajouter un manager

```
docker swarm join-token manager
```

Affiche la commande pour rejoindre le Swarm en tant que manager.

Note : On n'ajoute pas de nodes maintenant, c'est juste pour voir comment ça fonctionne.

Questions checkpoint

- Quelle est la différence entre un node manager et un node worker ?
- Pourquoi avoir un nombre impair de managers ?
- Que se passe-t-il si le seul manager tombe ?
- Peut-on utiliser Swarm avec un seul node ?

---

5 - Créer et Gérer des Services

L'essentiel

Créer un service simple

La commande de base :

```
docker service create [OPTIONS] IMAGE [COMMAND]
```

Exemple : Service nginx

```
docker service create \
  --name web \
  --replicas 3 \
  --publish 8080:80 \
  nginx:alpine
```

Décortiquons :

- `--name web` : Nom du service
- `--replicas 3` : 3 instances (tasks)
- `--publish 8080:80` : Port 8080 sur l'hôte → 80 dans les conteneurs
- `nginx:alpine` : Image à utiliser

Swarm va :

1. Télécharger l'image nginx:alpine si nécessaire
2. Créer 3 tasks
3. Les distribuer sur les nodes disponibles
4. Configurer le routing mesh pour le port 8080

Lister les services

```
docker service ls
```

Sortie :

```
ID            NAME    MODE          REPLICAS    IMAGE
abc123        web     replicated    3/3         nginx:alpine
```

Colonnes :

- `REPLICAS` : `3/3` = 3 tasks en cours / 3 attendues
- `MODE` : replicated (vs global)

Inspecter un service

```
# Pretty format
docker service inspect web --pretty

# Format JSON
docker service inspect web
```

Affiche toute la configuration du service.

Voir les tasks d'un service

```
docker service ps web
```

Sortie :

```
ID        NAME    IMAGE          NODE     DESIRED STATE   CURRENT STATE
abc1      web.1   nginx:alpine   laptop   Running         Running 2 minutes ago
abc2      web.2   nginx:alpine   laptop   Running         Running 2 minutes ago
abc3      web.3   nginx:alpine   laptop   Running         Running 2 minutes ago
```

Chaque ligne = une task (conteneur) du service.

Colonnes importantes :

- `NAME` : web.1, web.2, web.3 (numéros de réplicas)
- `NODE` : Sur quel node la task tourne
- `DESIRED STATE` : Ce que Swarm veut
- `CURRENT STATE` : État actuel

Scaler un service

Augmenter ou diminuer le nombre de réplicas :

```
# Scaler à 5 réplicas
docker service scale web=5

# Ou avec update
docker service update --replicas 5 web
```

Swarm va créer 2 tasks supplémentaires instantanément.

```
docker service ps web
```

Maintenant 5 tasks en cours.

Scaler à la baisse :

```
docker service scale web=2
```

Swarm va arrêter 3 tasks.

Logs d'un service

```
# Tous les logs du service (toutes les tasks)
docker service logs web

# Suivre les logs en temps réel
docker service logs -f web

# Dernières 50 lignes
docker service logs --tail 50 web
```

Les logs de **toutes les tasks** sont agrégés.

Supprimer un service

```
docker service rm web
```

Swarm arrête toutes les tasks et supprime le service.

EXERCICE GUIDÉ 2 : Créer et manipuler un service

Objectif : Créer un service nginx, le scaler, et observer le routing mesh

Étape 1 : Créer le service

```
docker service create \
  --name webapp \
  --replicas 2 \
  --publish 8080:80 \
  nginx:alpine
```

Attendre que le service soit prêt.

Étape 2 : Vérifier le service

```
docker service ls
```

Devrait afficher `2/2` pour REPLICAS.

```
docker service ps webapp
```

Devrait afficher 2 tasks en état Running.

Étape 3 : Tester le routing mesh

Ouvrir le navigateur : http://localhost:8080

Vous devriez voir la page d'accueil nginx.

Étape 4 : Voir les tasks (conteneurs) créés

```
docker ps
```

Deux conteneurs nginx devraient être listés.

Étape 5 : Scaler à 5 réplicas

```
docker service scale webapp=5
```

Attendre quelques secondes.

```
docker service ps webapp
```

Maintenant 5 tasks.

```
docker ps
```

5 conteneurs nginx.

Étape 6 : Refresh le navigateur plusieurs fois

http://localhost:8080

Le routing mesh balance la charge entre les 5 réplicas. Vous ne verrez pas de différence visuelle (c'est du nginx statique), mais le load balancing fonctionne.

Étape 7 : Voir les logs

```
docker service logs webapp
```

Vous devriez voir les requêtes HTTP dans les logs des différentes tasks.

Étape 8 : Scaler à 1

```
docker service scale webapp=1
```

Swarm va arrêter 4 tasks.

```
docker service ps webapp
```

Vous verrez l'historique : 4 tasks en état Shutdown, 1 en Running.

Étape 9 : Supprimer le service

```
docker service rm webapp
```

Vérifier :

```
docker service ls
```

Plus de services.

```
docker ps
```

Plus de conteneurs nginx.

Questions checkpoint

- Quelle est la différence entre `docker run` et `docker service create` ?
- Comment Swarm sait-il combien de tasks créer ?
- Que se passe-t-il si vous tuez manuellement un conteneur d'un service ?
- Comment le routing mesh permet d'accéder au service ?

---

6 - Update et Rollback : Mises à Jour sans Downtime

L'essentiel

Le problème des mises à jour classiques

Sans orchestration :

```
# Arrêter l'ancienne version
docker stop api
docker rm api

# Démarrer la nouvelle version
docker run -d --name api mon-api:v2
```

Problème : Downtime pendant l'arrêt et le redémarrage !

```
┌──────────────────────────────────────────┐
│ Temps                                    │
├──────────────────────────────────────────┤
│ [API v1 running] [DOWNTIME] [API v2]  │
│                  ^^^^^^^^              │
│              Service indisponible !    │
└──────────────────────────────────────────┘
```

Rolling Update avec Swarm

Swarm peut faire des **mises à jour progressives** sans downtime.

Stratégie :

```
Service avec 4 réplicas v1.0 :
[v1] [v1] [v1] [v1]

Rolling update vers v2.0 (parallelism=2, delay=10s) :
Étape 1: [v2] [v2] [v1] [v1]  ← Démarre 2 v2, arrête 2 v1
Attente 10s...
Étape 2: [v2] [v2] [v2] [v2]  ← Démarre 2 v2, arrête 2 v1

Résultat : Toujours au moins 2 réplicas actifs pendant l'update
          → Pas de downtime !
```

Configuration d'une update policy

Lors de la création d'un service :

```
docker service create \
  --name api \
  --replicas 4 \
  --update-parallelism 2 \
  --update-delay 10s \
  --update-failure-action rollback \
  mon-api:v1
```

Options :

- `--update-parallelism 2` : Mettre à jour 2 tasks à la fois
- `--update-delay 10s` : Attendre 10s entre chaque batch
- `--update-failure-action rollback` : Rollback automatique si échec

Ou dans un fichier stack :

```
services:
  api:
    image: mon-api:v1
    deploy:
      replicas: 4
      update_config:
        parallelism: 2
        delay: 10s
        failure_action: rollback
        monitor: 30s        # Observer 30s avant de continuer
        max_failure_ratio: 0.3  # Max 30% de tasks en échec
```

Déclencher une mise à jour

```
# Mettre à jour l'image
docker service update --image mon-api:v2 api
```

Swarm va :

1. Pull l'image v2

2. Arrêter 2 tasks v1
3. Démarrer 2 tasks v2
4. Attendre 10s + vérifier healthcheck
5. Répéter jusqu'à 100% v2

Suivre la progression :

```
# Voir les tasks en cours
docker service ps api

# Voir l'état de l'update
watch docker service ps api
```

Vous verrez les anciennes tasks passer en Shutdown et les nouvelles en Running.

Rollback : revenir en arrière

Si la nouvelle version a un problème, rollback :

```
docker service rollback api
```

Swarm revient à la version précédente en utilisant la même stratégie progressive.

Rollback automatique :

Si vous avez configuré `failure_action: rollback`, Swarm rollback automatiquement si :

- Trop de tasks échouent (> max_failure_ratio)
- Les healthchecks échouent

EXERCICE GUIDÉ 3 : Rolling Update et Rollback

Objectif : Mettre à jour un service nginx sans downtime et tester le rollback

Étape 1 : Créer un service nginx avec update config

```
docker service create \
  --name web \
  --replicas 4 \
  --publish 8080:80 \
  --update-parallelism 2 \
  --update-delay 10s \
  nginx:1.24-alpine
```

Attendre que les 4 réplicas soient Running.

```
docker service ps web
```

Étape 2 : Tester l'accès

```
curl http://localhost:8080
```

Devrait afficher la page nginx.

Étape 3 : Mettre à jour vers nginx 1.25

```
docker service update --image nginx:1.25-alpine web
```

Étape 4 : Observer l'update en temps réel

Dans un terminal séparé :

```
watch -n 1 'docker service ps web'
```

Vous allez voir :

- 2 nouvelles tasks démarrer (nginx:1.25)
- 2 anciennes tasks s'arrêter (nginx:1.24)
- Attente de 10s
- 2 autres nouvelles tasks démarrer
- 2 autres anciennes tasks s'arrêter

Étape 5 : Pendant l'update, tester le service

Dans un autre terminal :

```
# Boucle pour tester en continu
while true; do curl -s http://localhost:8080 -I | grep "200 OK"; sleep 1; done
```

Le service devrait rester accessible pendant toute l'update (pas de downtime).

Étape 6 : Vérifier que l'update est terminée

```
docker service ps web
```

Toutes les tasks devraient être en version 1.25.

Étape 7 : Simuler une mauvaise mise à jour

```
# Mettre à jour vers une image qui n'existe pas
docker service update --image nginx:mauvaise-version web
```

Swarm va essayer de pull l'image, échouer, et les tasks vont crasher.

Étape 8 : Observer l'échec

```
docker service ps web
```

Vous verrez des tasks en état Failed.

Étape 9 : Rollback manuel

```
docker service rollback web
```

Swarm revient à nginx:1.25-alpine.

```
docker service ps web
```

Les tasks devraient revenir à l'état Running.

Étape 10 : Vérifier le service

```
curl http://localhost:8080
```

Service fonctionnel.

Étape 11 : Cleanup

```
docker service rm web
```

Questions checkpoint

- Quelle est la différence entre `--update-parallelism 1` et `--update-parallelism 10` ?
- Que se passe-t-il si on fait `--update-delay 0s` ?
- Pourquoi configurer un healthcheck est important pour les rolling updates ?
- Peut-on annuler un update en cours ?

---

7 - Stacks Swarm : Orchestrer des Applications Multi-Services

L'essentiel

Qu'est-ce qu'une stack ?

Une **stack** est un ensemble de services déployés et gérés ensemble, définis dans un fichier `docker-compose.yml` (ou stack.yml).

Différence avec docker-compose :

```
docker-compose :
└─> Pour dev local, single host
└─> docker-compose up/down
└─> Pas de scaling automatique
└─> Pas de rolling updates

docker stack (Swarm) :
└─> Pour production, multi-host
└─> docker stack deploy
└─> Scaling, HA, load balancing
└─> Rolling updates intégrés
```

Avantage : Déployer toute votre application (API + DB + Redis + Frontend) en une seule commande.

Format du fichier stack

Le fichier est presque identique à `docker-compose.yml`, avec quelques ajouts spécifiques à Swarm.

Exemple : stack.yml

```yaml
version: '3.8'

services:
  api:
    image: mon-api:latest
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
    networks:
      - backend
    secrets:
      - db_password

  db:
    image: postgres:16-alpine
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - backend
```

```yaml
  redis:
    image: redis:7-alpine
    deploy:
      replicas: 2
    networks:
      - backend

networks:
  backend:
    driver: overlay

volumes:
  db_data:

secrets:
  db_password:
    external: true
```

Différences clés avec docker-compose.yml

1. Section `deploy:` (ignorée par docker-compose)

```yaml
services:
  api:
    deploy:
      replicas: 3           # Nombre de réplicas
      update_config:        # Configuration rolling update
        parallelism: 2
        delay: 10s
      restart_policy:       # Politique de redémarrage
        condition: on-failure
        max_attempts: 3
      placement:            # Contraintes de placement
        constraints:
          - node.role == worker
      resources:            # Limites de ressources
        limits:
          cpus: '0.5'
          memory: 512M
```

2. Networks overlay (par défaut)

```yaml
networks:
  backend:
    driver: overlay  # Nécessaire pour multi-host
```

3. Secrets Swarm

```yaml
secrets:
  db_password:
    external: true  # Secret créé avec docker secret create
```

4. Configs Swarm

```
configs:
  nginx_config:
    file: ./nginx.conf
```

Déployer une stack

```
# Créer le secret d'abord (si nécessaire)
echo "mon_mot_de_passe" | docker secret create db_password -

# Déployer la stack
docker stack deploy -c stack.yml myapp
```

Swarm va :

1. Créer les networks overlay
2. Créer les volumes
3. Démarrer tous les services
4. Distribuer les tasks sur les nodes
5. Configurer le routing mesh

Gérer une stack

```
# Lister les stacks
docker stack ls

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

# Lister toutes les tasks d'une stack
docker stack ps myapp

# Voir les logs (tous les services)
docker service logs myapp_api

# Supprimer la stack
docker stack rm myapp
```

Important : `docker stack rm` supprime les services et networks, mais **PAS les volumes** (sécurité des données).

Mettre à jour une stack

Pour mettre à jour, il suffit de modifier le fichier stack.yml et de redéployer :

```
# Modifier stack.yml (ex: changer l'image de l'API)
vim stack.yml

# Redéployer
docker stack deploy -c stack.yml myapp
```

Swarm va :

- Détecter les changements
- Faire un rolling update des services modifiés
- Ne pas toucher aux services inchangés

EXERCICE GUIDÉ 4 : Déployer une stack complète

Objectif : Déployer une stack nginx + redis multi-réplicas

Étape 1 : Créer le fichier stack.yml

```
version: '3.8'

services:
  web:
    image: nginx:alpine
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
        delay: 5s
    ports:
      - "8080:80"
    networks:
      - webnet

  redis:
    image: redis:7-alpine
    deploy:
      replicas: 2
    networks:
      - webnet

networks:
  webnet:
    driver: overlay
```

Étape 2 : Déployer la stack

```
docker stack deploy -c stack.yml webapp
```

Étape 3 : Vérifier la stack

```
# Voir la stack
docker stack ls
```

Sortie :

```
NAME      SERVICES
webapp    2
```

```
# Voir les services de la stack
docker stack services webapp
```

Sortie :

```
ID          NAME          MODE        REPLICAS   IMAGE
abc123      webapp_web    replicated  3/3        nginx:alpine
def456      webapp_redis  replicated  2/2        redis:7-alpine
```

Étape 4 : Voir toutes les tasks

```
docker stack ps webapp
```

Vous devriez voir 5 tasks : 3 nginx + 2 redis.

Étape 5 : Tester le service web

```
curl http://localhost:8080
```

Page nginx par défaut.

Étape 6 : Voir les logs d'un service

```
docker service logs webapp_web
```

Étape 7 : Mettre à jour la stack

Modifier stack.yml : changer `replicas: 3` à `replicas: 5` pour web.

```
docker stack deploy -c stack.yml webapp
```

Observer l'update :

```
watch docker stack ps webapp
```

2 nouvelles tasks web vont apparaître.

Étape 8 : Supprimer la stack

```
docker stack rm webapp
```

Vérifier :

```
docker stack ls
```

Plus de stacks.

```
docker service ls
```

Plus de services.

Questions checkpoint

- Quelle est la différence entre `docker-compose up` et `docker stack deploy` ?
- Pourquoi utiliser un network overlay ?
- Que se passe-t-il aux volumes quand on fait `docker stack rm` ?
- Peut-on déployer plusieurs stacks en même temps ?

---

8 - Secrets et Configs Swarm

L'essentiel

Secrets Swarm : gestion sécurisée

Les **secrets** permettent de stocker des données sensibles (mots de passe, clés API, certificats) de manière sécurisée dans Swarm.

Différence avec variables d'environnement :

```
Variables d'environnement :
❌ Visibles dans docker inspect
❌ Peuvent fuiter dans les logs
❌ Stockées en clair

Secrets Swarm :
✅ Chiffrés au repos (encrypted at rest)
✅ Transmis via TLS aux conteneurs
✅ Montés en tmpfs (mémoire, pas disque)
✅ Accès restreint aux services autorisés
```

Créer un secret

Depuis stdin :

```
echo "mon_mot_de_passe_secret" | docker secret create db_password -
```

Depuis un fichier :

```
docker secret create db_password ./password.txt
```

Vérifier :

```
docker secret ls
```

Sortie :

```
ID              NAME           CREATED
abc123          db_password    10 seconds ago
```

Important : On ne peut **jamais** lire le contenu d'un secret après l'avoir créé (sécurité).

```
docker secret inspect db_password
```

Affiche les métadonnées, mais PAS le contenu.

Utiliser un secret dans un service

```
docker service create \
  --name api \
  --secret db_password \
  mon-api:latest
```

Dans le conteneur, le secret est disponible à `/run/secrets/db_password`.

Dans votre application :

```
# Python example
with open('/run/secrets/db_password', 'r') as f:
    db_password = f.read().strip()
```

Secrets dans une stack

```
version: '3.8'

services:
  api:
    image: mon-api:latest
    secrets:
      - db_password
      - api_key

  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password

secrets:
  db_password:
```

```
    external: true  # Créé avec docker secret create
  api_key:
    file: ./api_key.txt  # Lu depuis un fichier au deploy
```

Avant de déployer :

```
# Créer le secret externe
echo "secret_password" | docker secret create db_password -

# Déployer
docker stack deploy -c stack.yml myapp
```

Configs Swarm : configurations non-sensibles

Les **configs** sont similaires aux secrets, mais pour des données **non-sensibles** (fichiers de config, scripts).

Différence avec secrets :

```
Secrets : Données sensibles (chiffrées)
Configs : Données publiques (non chiffrées)
```

Exemple : config nginx

```
# Créer un config
docker config create nginx_config nginx.conf

# Utiliser dans un service
docker service create \
  --name web \
  --config source=nginx_config,target=/etc/nginx/nginx.conf \
  nginx:alpine
```

Dans une stack :

```
services:
  nginx:
    image: nginx:alpine
    configs:
      - source: nginx_config
        target: /etc/nginx/nginx.conf

configs:
  nginx_config:
    file: ./nginx.conf
```

Rotation de secrets

Pour changer un secret sans downtime :

```
# 1. Créer le nouveau secret
echo "nouveau_password" | docker secret create db_password_v2 -

# 2. Mettre à jour le service
docker service update \
  --secret-rm db_password \
  --secret-add db_password_v2 \
  api

# 3. Mettre à jour votre app pour lire db_password_v2
# 4. Supprimer l'ancien secret
docker secret rm db_password
```

EXERCICE GUIDÉ 5 : Utiliser les secrets

Objectif : Créer un service PostgreSQL avec mot de passe dans un secret

Étape 1 : Créer le secret

```
echo "super_secret_password" | docker secret create postgres_password -
```

Vérifier :

```
docker secret ls
```

Étape 2 : Créer le service PostgreSQL

```
docker service create \
  --name database \
  --secret postgres_password \
  --env POSTGRES_PASSWORD_FILE=/run/secrets/postgres_password \
  postgres:16-alpine
```

Étape 3 : Vérifier que le service tourne

```
docker service ps database
```

Étape 4 : Se connecter au conteneur et vérifier le secret

```
# Trouver le conteneur
docker ps

# Se connecter
docker exec -it <container_id> sh

# Dans le conteneur
ls /run/secrets/
cat /run/secrets/postgres_password
```

Vous devriez voir `super_secret_password`.

Étape 5 : Tester la connexion PostgreSQL

```
# Dans le conteneur
psql -U postgres
# Entrer le mot de passe quand demandé
```

Étape 6 : Essayer d'inspecter le secret (échec attendu)

```
docker secret inspect postgres_password
```

Le contenu n'est PAS affiché (sécurité).

Étape 7 : Cleanup

```
docker service rm database
docker secret rm postgres_password
```

Questions checkpoint

- Pourquoi utiliser un secret plutôt qu'une variable d'environnement ?
- Où sont stockés les secrets dans le conteneur ?
- Peut-on modifier un secret existant ?
- Quelle est la différence entre un secret et un config ?

---

9 - Placement Constraints : Contrôler où les Services s'Exécutent

L'essentiel

Pourquoi des contraintes de placement ?

Par défaut, Swarm distribue les tasks uniformément sur tous les nodes disponibles. Mais parfois, vous voulez plus de contrôle :

Cas d'usage :

- Base de données sur nodes avec SSD
- Services gourmands en CPU sur nodes puissants
- Services stateful (DB) sur managers (plus stable)
- Services par région géographique

Labels sur les nodes

Vous pouvez ajouter des **labels** aux nodes pour les catégoriser.

```
# Ajouter un label
docker node update --label-add type=ssd node1
docker node update --label-add region=europe node2
docker node update --label-add env=production node3
```

Voir les labels :

```
docker node inspect node1 --pretty
```

Contraintes de placement

1. Par rôle de node :

```
services:
  db:
    image: postgres:16-alpine
    deploy:
      placement:
        constraints:
          - node.role == manager  # Seulement sur managers
```

```
services:
  worker:
    image: mon-worker
    deploy:
      placement:
        constraints:
          - node.role == worker  # Seulement sur workers
```

2. Par label :

```
services:
  db:
    image: postgres:16-alpine
    deploy:
      placement:
        constraints:
          - node.labels.type == ssd  # Seulement nodes avec SSD
```

3. Par hostname :

```
services:
  monitoring:
    image: prometheus
    deploy:
      placement:
        constraints:
          - node.hostname == server-1  # Seulement sur server-1
```

4. Combinaisons :

```yaml
services:
  api:
    image: mon-api
    deploy:
      placement:
        constraints:
          - node.role == worker
          - node.labels.env == production
          - node.labels.region == europe
```

Toutes les contraintes doivent être satisfaites (ET logique).

Préférences de placement

Au lieu de contraintes strictes, vous pouvez exprimer des **préférences** :

```yaml
services:
  api:
    image: mon-api
    deploy:
      replicas: 6
      placement:
        preferences:
          - spread: node.labels.zone  # Répartir sur différentes zones
```

Swarm essaie de répartir équitablement entre les zones (zone-a, zone-b, zone-c).

Mode global : un par node

Le mode **global** démarre exactement une task par node (pas de replicas).

```yaml
services:
  monitoring:
    image: node-exporter
    deploy:
      mode: global  # Une instance par node
```

Utile pour :

- Agents de monitoring
- Log collectors
- Reverse proxies locaux

Avec contraintes :

```yaml
services:
  monitoring:
    image: node-exporter
    deploy:
      mode: global
      placement:
        constraints:
          - node.role == worker  # Seulement workers
```

Une instance par worker node.

EXERCICE GUIDÉ 6 : Labels et contraintes

Objectif : Configurer des labels et déployer des services avec contraintes

Étape 1 : Ajouter des labels au node actuel

```
# Voir l'ID de votre node
docker node ls

# Ajouter des labels (remplacer NODE_ID)
docker node update --label-add env=dev NODE_ID
docker node update --label-add type=local NODE_ID
```

Étape 2 : Vérifier les labels

```
docker node inspect self --pretty | grep -A 5 Labels
```

Vous devriez voir vos labels.

Étape 3 : Créer un service avec contrainte

```
docker service create \
  --name api \
  --replicas 2 \
  --constraint 'node.labels.env == dev' \
  nginx:alpine
```

Étape 4 : Vérifier le placement

```
docker service ps api
```

Les 2 tasks devraient être sur votre node (seul node avec label env=dev).

Étape 5 : Essayer une contrainte impossible

```
docker service create \
  --name impossible \
  --constraint 'node.labels.env == production' \
  nginx:alpine
```

Étape 6 : Vérifier l'échec

```
docker service ps impossible
```

La task sera en état "Pending" ou "No suitable node" car aucun node ne matche.

Étape 7 : Service en mode global

```
docker service create \
  --name global-service \
  --mode global \
  alpine ping 8.8.8.8
```

Étape 8 : Vérifier

```
docker service ps global-service
```

Exactement 1 task (vous n'avez qu'un node).

Étape 9 : Cleanup

```
docker service rm api impossible global-service
```

Questions checkpoint

- Quelle est la différence entre une contrainte et une préférence ?
- Quand utiliser le mode global plutôt que replicated ?
- Que se passe-t-il si aucun node ne satisfait les contraintes ?
- Comment voir les labels d'un node ?

---

10 - Projet d'Après-Midi : Stack Flask Complète en Swarm

Objectif

Déployer une application web complète en Docker Swarm avec :

- **Backend** : API Flask (Python) avec endpoints REST
- **Database** : PostgreSQL avec volume persistant
- **Cache** : Redis
- **Frontend** : Nginx servant du HTML/JS statique

Compétences mises en pratique :

✅ Création de Dockerfiles optimisés
✅ Déploiement via docker stack
✅ Scaling des services
✅ Secrets pour credentials
✅ Networks overlay

✅ Volumes persistants
✅ Rolling updates
✅ Healthchecks

Structure du projet

```
flask-swarm-project/
├── backend/
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── app.py
│   └── models.py
├── frontend/
│   ├── Dockerfile
│   ├── index.html
│   └── app.js
├── stack.yml
├── secrets/
│   └── db_password.txt
└── README.md
```

Phase 1 : Setup initial et Backend Flask

Étape 1 : Créer la structure

```
mkdir -p flask-swarm-project/{backend,frontend,secrets}
cd flask-swarm-project
```

Étape 2 : Backend - requirements.txt

```
flask==3.0.0
flask-cors==4.0.0
psycopg2-binary==2.9.9
redis==5.0.1
```

Étape 3 : Backend - app.py

```
from flask import Flask, jsonify, request
from flask_cors import CORS
import psycopg2
import redis
import os

app = Flask(__name__)
CORS(app)

# Lire le secret
with open('/run/secrets/db_password', 'r') as f:
    DB_PASSWORD = f.read().strip()
```

```
# Configuration
DB_CONFIG = {
    'host': os.getenv('DB_HOST', 'db'),
    'database': os.getenv('DB_NAME', 'flask_db'),
    'user': os.getenv('DB_USER', 'postgres'),
    'password': DB_PASSWORD
}

REDIS_HOST = os.getenv('REDIS_HOST', 'redis')

# Redis client
redis_client = redis.Redis(host=REDIS_HOST, port=6379, decode_responses=True)

def get_db_connection():
    return psycopg2.connect(**DB_CONFIG)

@app.route('/health', methods=['GET'])
def health():
    """Healthcheck endpoint"""
    try:
        # Test DB
        conn = get_db_connection()
        conn.close()
        # Test Redis
        redis_client.ping()
        return jsonify({'status': 'healthy', 'db': 'ok', 'redis': 'ok'}), 200
    except Exception as e:
        return jsonify({'status': 'unhealthy', 'error': str(e)}), 503

@app.route('/api/items', methods=['GET'])
def get_items():
    """Get all items"""
    # Try cache first
    cached = redis_client.get('items')
    if cached:
        return jsonify({'source': 'cache', 'items': eval(cached)})

    # Query DB
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute('SELECT id, name, description FROM items ORDER BY id')
    items = [{'id': row[0], 'name': row[1], 'description': row[2]} for row in
cur.fetchall()]
    cur.close()
    conn.close()

    # Cache for 60 seconds
    redis_client.setex('items', 60, str(items))

    return jsonify({'source': 'database', 'items': items})

@app.route('/api/items', methods=['POST'])
def create_item():
```

```
    """Create a new item"""
    data = request.json
    name = data.get('name')
    description = data.get('description', '')

    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute(
        'INSERT INTO items (name, description) VALUES (%s, %s) RETURNING id',
        (name, description)
    )
    item_id = cur.fetchone()[0]
    conn.commit()
    cur.close()
    conn.close()

    # Invalidate cache
    redis_client.delete('items')

    return jsonify({'id': item_id, 'name': name, 'description': description}), 201

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Étape 4 : Backend - Dockerfile

```
FROM python:3.11-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy app
COPY app.py .

# Healthcheck
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD python -c "import requests; requests.get('http://localhost:5000/health')" || exit 1

EXPOSE 5000

CMD ["python", "app.py"]
```

Étape 5 : Frontend - index.html

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>Flask Swarm App</title>
    <style>
        body { font-family: Arial, sans-serif; max-width: 800px; margin: 50px auto;
padding: 20px; }
        .item { border: 1px solid #ddd; padding: 10px; margin: 10px 0; border-radius: 5px;
}
        button { background: #007bff; color: white; border: none; padding: 10px 20px;
cursor: pointer; border-radius: 5px; }
        button:hover { background: #0056b3; }
        input, textarea { width: 100%; padding: 10px; margin: 5px 0; border: 1px solid
#ddd; border-radius: 5px; }
    </style>
</head>
<body>
    <h1>Items Manager</h1>

    <div>
        <h2>Add Item</h2>
        <input type="text" id="itemName" placeholder="Name">
        <textarea id="itemDescription" placeholder="Description"></textarea>
        <button onclick="addItem()">Add Item</button>
    </div>

    <div>
        <h2>Items List</h2>
        <div id="items"></div>
        <button onclick="loadItems()">Refresh</button>
    </div>

    <script src="app.js"></script>
</body>
</html>
```

Étape 6 : Frontend - app.js

```
const API_URL = 'http://localhost:5000/api';

async function loadItems() {
    try {
        const response = await fetch(`${API_URL}/items`);
        const data = await response.json();

        const itemsDiv = document.getElementById('items');
        itemsDiv.innerHTML = `<p><em>Source: ${data.source}</em></p>`;

        if (data.items.length === 0) {
            itemsDiv.innerHTML += '<p>No items yet.</p>';
        } else {
            data.items.forEach(item => {
                itemsDiv.innerHTML += `
                    <div class="item">
                        <h3>${item.name}</h3>
                        <p>${item.description}</p>
```

```
                    </div>
                `;
            });
        }
    } catch (error) {
        document.getElementById('items').innerHTML = `<p style="color: red;">Error: ${error.message}</p>`;
    }
}

async function addItem() {
    const name = document.getElementById('itemName').value;
    const description = document.getElementById('itemDescription').value;

    if (!name) {
        alert('Name is required');
        return;
    }

    try {
        const response = await fetch(`${API_URL}/items`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ name, description })
        });

        if (response.ok) {
            document.getElementById('itemName').value = '';
            document.getElementById('itemDescription').value = '';
            loadItems();
        }
    } catch (error) {
        alert(`Error: ${error.message}`);
    }
}

// Load items on page load
window.onload = loadItems;
```

Étape 7 : Frontend - Dockerfile

```
FROM nginx:alpine

# Copy static files
COPY index.html /usr/share/nginx/html/
COPY app.js /usr/share/nginx/html/

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Phase 2 : Configuration Swarm et Déploiement

Étape 8 : Créer le secret pour la DB

```
echo "flask_secret_password" > secrets/db_password.txt
docker secret create db_password secrets/db_password.txt
```

Étape 9 : Créer stack.yml

```
version: '3.8'

services:
  backend:
    image: flask-api:latest
    build:
      context: ./backend
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
    ports:
      - "5000:5000"
    environment:
      DB_HOST: db
      DB_NAME: flask_db
      DB_USER: postgres
      REDIS_HOST: redis
    secrets:
      - db_password
    networks:
      - backend
    depends_on:
      - db
      - redis

  db:
    image: postgres:16-alpine
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager
    environment:
      POSTGRES_DB: flask_db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password
    volumes:
```

```
      - db_data:/var/lib/postgresql/data
    networks:
      - backend
    # Script init SQL
    command: >
      sh -c "
      docker-entrypoint.sh postgres &
      sleep 10 &&
      PGPASSWORD=$$(cat /run/secrets/db_password) psql -U postgres -d flask_db -c '
      CREATE TABLE IF NOT EXISTS items (
        id SERIAL PRIMARY KEY,
        name VARCHAR(255) NOT NULL,
        description TEXT,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      );
      INSERT INTO items (name, description) VALUES
      (''Sample Item 1'', ''This is a sample item''),
      (''Sample Item 2'', ''Another sample item''),
      (''Sample Item 3'', ''Yet another sample'')
      ON CONFLICT DO NOTHING;
      ' || wait
      "

  redis:
    image: redis:7-alpine
    deploy:
      replicas: 2
    networks:
      - backend

  frontend:
    image: flask-frontend:latest
    build:
      context: ./frontend
    deploy:
      replicas: 2
    ports:
      - "8080:80"
    networks:
      - frontend

networks:
  backend:
    driver: overlay
  frontend:
    driver: overlay

volumes:
  db_data:

secrets:
  db_password:
    external: true
```

Étape 10 : Build les images

```
# Backend
docker build -t flask-api:latest ./backend

# Frontend
docker build -t flask-frontend:latest ./frontend
```

Étape 11 : Déployer la stack

```
docker stack deploy -c stack.yml flaskapp
```

Étape 12 : Vérifier le déploiement

```
# Voir la stack
docker stack ls

# Voir les services
docker stack services flaskapp

# Voir les tasks
docker stack ps flaskapp

# Attendre que tout soit Running
watch docker stack ps flaskapp
```

Étape 13 : Tester l'application

Ouvrir le navigateur : http://localhost:8080

Vous devriez voir :

- 3 items pré-remplis
- Formulaire pour ajouter des items
- Les données sont cachées dans Redis

Étape 14 : Tester le healthcheck

```
curl http://localhost:5000/health
```

Devrait retourner :

```
{"status": "healthy", "db": "ok", "redis": "ok"}
```

Phase 3 : Tests de Scaling et Résilience

Étape 15 : Scaler le backend

```
docker service scale flaskapp_backend=5
```

Observer :

```
docker service ps flaskapp_backend
```

5 réplicas au lieu de 3.

Étape 16 : Tester le load balancing

```
# Faire plusieurs requêtes
for i in {1..10}; do curl -s http://localhost:5000/api/items | grep source; done
```

Vous verrez alterner entre "cache" et "database" selon le cache Redis.

Étape 17 : Tester la résilience (self-healing)

```
# Trouver un conteneur backend
docker ps | grep flask-api

# Le tuer
docker kill <container_id>

# Observer que Swarm le redémarre
docker service ps flaskapp_backend
```

Une nouvelle task va démarrer automatiquement.

Étape 18 : Test de persistance des données

```
# Ajouter un item via le frontend
# Ou via curl
curl -X POST http://localhost:5000/api/items \
  -H "Content-Type: application/json" \
  -d '{"name":"Test Item","description":"Created via curl"}'

# Supprimer toute la stack
docker stack rm flaskapp

# Attendre que tout soit supprimé
docker stack ps flaskapp  # Devrait être vide

# Redéployer
docker stack deploy -c stack.yml flaskapp

# Attendre que tout soit Running
watch docker stack ps flaskapp

# Vérifier que les données existent toujours
curl http://localhost:5000/api/items
```

Les items devraient toujours être là (volume persistant).

Phase 4 : Rolling Update

Étape 19 : Modifier le backend

Modifier `backend/app.py` : changer le message de health :

```
@app.route('/health', methods=['GET'])
def health():
    return jsonify({'status': 'healthy', 'version': '2.0', 'db': 'ok', 'redis': 'ok'}),
200
```

Étape 20 : Rebuild l'image

```
docker build -t flask-api:v2 ./backend
```

Étape 21 : Mettre à jour stack.yml

Changer `image: flask-api:latest` à `image: flask-api:v2`

Étape 22 : Déployer l'update

```
docker stack deploy -c stack.yml flaskapp
```

Observer le rolling update :

```
watch docker stack ps flaskapp_backend
```

Les anciennes tasks vont être remplacées progressivement (1 par 1, avec 10s de délai).

Étape 23 : Vérifier la nouvelle version

```
curl http://localhost:5000/health
```

Devrait afficher `"version": "2.0"`.

Phase 5 : Nettoyage et Documentation

Étape 24 : Créer le README.md

```
# Flask Swarm Application

Stack Docker Swarm complète avec Flask, PostgreSQL, Redis et Nginx.

## Architecture

- **Backend** : Flask API (3 replicas)
- **Database** : PostgreSQL (1 replica, volume persistant)
- **Cache** : Redis (2 replicas)
- **Frontend** : Nginx (2 replicas)

## Prérequis
```

```
- Docker Swarm initialisé
- Ports 5000 et 8080 disponibles

## Déploiement

```bash
# 1. Créer le secret
echo "votre_mot_de_passe" > secrets/db_password.txt
docker secret create db_password secrets/db_password.txt

# 2. Build les images
docker build -t flask-api:latest ./backend
docker build -t flask-frontend:latest ./frontend

# 3. Déployer la stack
docker stack deploy -c stack.yml flaskapp

# 4. Vérifier
docker stack services flaskapp
```

Utilisation

- Frontend : http://localhost:8080
- API : http://localhost:5000/api/items
- Health : http://localhost:5000/health

Scaling

```
docker service scale flaskapp_backend=5
```

Suppression

```
docker stack rm flaskapp
docker secret rm db_password
```

```
**Étape 25 : Cleanup final**

```bash
# Supprimer la stack
docker stack rm flaskapp

# Supprimer le secret
docker secret rm db_password

# Optionnel : supprimer les images
docker rmi flask-api:latest flask-api:v2 flask-frontend:latest
```

Validation et Questions finales

Checklist de validation :

- [ ] Stack déployée avec `docker stack deploy`
- [ ] 3 réplicas backend fonctionnels
- [ ] PostgreSQL avec données persistantes
- [ ] Redis cache fonctionnel
- [ ] Frontend accessible et fonctionnel
- [ ] Healthcheck OK sur `/health`
- [ ] Ajout d'items via l'interface
- [ ] Données persistent après `docker stack rm` puis redéploiement
- [ ] Scaling testé (passage à 5 réplicas)
- [ ] Self-healing testé (kill container → redémarrage auto)
- [ ] Rolling update testé (v1 → v2 sans downtime)
- [ ] README.md complet

P'tit quizz de fin de journée

Questions :

1. Pourquoi utiliser `docker stack deploy` plutôt que `docker-compose up` ?
2. Comment Swarm assure-t-il le load balancing entre les 3 réplicas backend ?
3. Que se passe-t-il si on supprime le volume après `docker stack rm` ?
4. Pourquoi placer PostgreSQL sur un manager node ?
5. Comment fonctionne le cache Redis dans cette architecture ?
6. Que se passerait-il avec 0 replica de backend pendant un rolling update ?

Pour aller plus loin

Documentation officielle :

- Docker Swarm : https://docs.docker.com/engine/swarm/
- Swarm mode CLI : https://docs.docker.com/engine/reference/commandline/swarm/
- Services : https://docs.docker.com/engine/reference/commandline/service/
- Stacks : https://docs.docker.com/engine/reference/commandline/stack/

Tutoriels avancés :

- Swarm avec Traefik (reverse proxy) : https://doc.traefik.io/traefik/providers/docker/
- Multi-node Swarm sur cloud providers
- Monitoring avec Prometheus + Grafana

- CI/CD avec Swarm

Commandes essentielles à retenir

```
# Swarm
docker swarm init
docker swarm join
docker node ls

# Services
docker service create
docker service ls
docker service ps <service>
docker service scale <service>=<replicas>
docker service update
docker service rollback
docker service logs <service>
docker service rm <service>

# Stacks
docker stack deploy -c <file> <stack>
docker stack ls
docker stack services <stack>
docker stack ps <stack>
docker stack rm <stack>

# Secrets
docker secret create <name> <file>
docker secret ls
docker secret inspect <name>
docker secret rm <name>

# Nodes
docker node update --label-add <key>=<value> <node>
docker node inspect <node>
```