Cours DevOps - Partie 2 : Approfondissement Docker & Architecture

Vue d'ensemble - Points abordés dans cette Partie 2

Aperçu rapide de tout ce qui est vu aujourd'hui

📚 Théories

- **Debug et Troubleshooting**
  - Commandes de diagnostic essentielles (inspect, logs, stats, exec, top, history)
  - Erreurs classiques et leurs solutions
  - Lecture et interprétation des logs Docker
  - Techniques de debugging progressif
- **Optimisation d'images Docker**
  - Impact de la taille des images (stockage, bande passante, sécurité)
  - Images Alpine vs images standard
  - Multi-stage builds : concept et avantages
  - Layer caching et invalidation du cache
  - Principe du .dockerignore
- **Healthchecks et availability**
  - Différence entre "started" et "ready"
  - Configuration de healthchecks pour les services
  - Conditions de dépendance (service_healthy vs service_started)
- **Architecture multi-services**
  - Reverse proxy avec Nginx
  - Séparation frontend/backend/database
  - Communication inter-conteneurs
  - Exposition sélective des ports

🛠️ Mises en pratique

- **Debug de Dockerfiles cassés**
  - Dockerfile #1 : correction de la syntaxe CMD
  - Dockerfile #2 : optimisation de l'ordre des COPY pour le cache
  - Dockerfile #3 : réduction de taille avec Alpine et multi-stage
  - Utilisation de docker build pour identifier les erreurs
  - Lecture des messages d'erreur et logs de build

- **Debug de docker-compose.yml**
  - Identification d'erreurs de configuration (noms de services, variables d'environnement)
  - Ajout de healthchecks pour Postgres
  - Utilisation de docker compose logs pour diagnostiquer
  - Correction de problèmes de connectivité entre services
- **Challenge d'optimisation d'images**
  - Création d'un .dockerignore
  - Conversion vers image Alpine
  - Implémentation de multi-stage build
  - Combinaison de commandes RUN
  - Utilisation de npm ci --only=production
  - Comparaison des tailles avant/après (docker images)
  - Calcul du pourcentage de gain
- **Mini-projet : Stack complète**
  - Création d'une architecture Nginx + API + Postgres + Redis
  - Configuration de Nginx comme reverse proxy
  - Mise en place de volumes persistants pour tous les services
  - Configuration des healthchecks
  - Tests de l'ensemble de la stack
  - Validation de la persistance après redémarrage complet
  - Optimisation de toutes les images
- **Commandes Docker utilisées**
  - docker --version, docker compose version
  - docker run (avec options -it, --name, -v, -p, --rm, --mount)
  - docker build -t
  - docker images, docker ps, docker stop, docker rm
  - docker logs -f, docker inspect, docker stats, docker top
  - docker exec -it
  - docker volume (create, ls, inspect, rm)
  - docker compose (up, down, build, logs, ps)
  - docker system prune
  - docker cp (copie de fichiers)

1 - Networks Docker : Maîtriser la Communication entre Conteneurs

CORE NOTIONS

C'est quoi un network Docker ?

Un network Docker est un réseau virtuel qui permet à vos conteneurs de communiquer entre eux de manière isolée et sécurisée. Sans network, vos conteneurs sont comme des îles isolées qui ne peuvent pas se parler.

Quand vous lancez plusieurs conteneurs (API, base de données, cache), ils doivent pouvoir échanger des données. Le network Docker crée un **pont de communication** entre eux.

Pourquoi utiliser des networks Docker ?

Problème sans network personnalisé :

```
# Vous lancez un conteneur PostgreSQL
docker run --name example-api node:22-alpine sleep infinity

# Vous lancez votre API
docker run --name example-api-2 node:22-alpine sleep infinity
```

Ici L'API ne peut pas se connecter à ma-db
Pourquoi ? Ils ne sont pas sur le même réseau !

Solution avec network :

```
# Créer un network
docker network create shared-network

# Lancer les conteneurs sur ce network
docker run -d -p 3000:3000 --name example-api --network example-network node:22-alpine
sleep infinity
docker run -d -p 8080:8080 --name example-api-2 --network example-network node:22-alpine
sleep infinity

# Maintenant l'API peut se connecter à "ma-db" directement !

# Essayer de se connecter à l'autre api avec curl
# Installation
apk add curl
# Essai
curl http://example-api-2:8080

=> faire 2 api simples dockerisées
```

Les 4 types de networks Docker

1. Bridge (par défaut)

- Network le plus courant
- Conteneurs sur le même host peuvent communiquer
- Isolés du réseau externe (sauf si ports exposés)
- Docker crée automatiquement un bridge network appelé "bridge"

2. Host

- Conteneur partage le réseau de l'hôte
- Pas d'isolation réseau
- Performance maximale (pas de traduction de ports)
- Attention aux conflits de ports

3. Overlay

- Pour Docker Swarm (orchestration multi-serveurs)
- Permet la communication entre conteneurs sur différents hosts
- On ne l'utilisera pas dans ce cours

4. None

- Aucun réseau
- Conteneur complètement isolé
- Utile pour des tâches qui ne nécessitent pas de réseau

Comment ça marche concrètement ?

DNS automatique : Quand vous mettez des conteneurs sur un même network custom, Docker active un DNS automatique. Chaque conteneur peut être atteint par son nom.

```yaml
# docker-compose.yml
services:
  api:
    image: mon-api
    networks:
      - mon-network

  database:
    image: postgres
    networks:
      - mon-network

networks:
  mon-network:
```

Dans le conteneur `api`, vous pouvez faire :

```
// Se connecter à la base de données par son nom
const db = new Database({
  host: 'database',  // Le nom du service !
  port: 5432
});
```

Isolation réseau

Les conteneurs sur des networks différents ne peuvent PAS communiquer entre eux (sauf configuration explicite).

```
Network A           Network B
  ├── api              ├── front
  └── db               └── nginx

api peut parler à db
front peut parler à nginx
Mais api ne peut PAS parler à front
```

Commandes essentielles

```
# Lister les networks
docker network ls

# Créer un network
docker network create mon-network

# Inspecter un network (voir quels conteneurs sont dessus)
docker network inspect mon-network

# Connecter un conteneur existant à un network
docker network connect mon-network mon-conteneur

# Déconnecter un conteneur d'un network
docker network disconnect mon-network mon-conteneur

# Supprimer un network (aucun conteneur ne doit l'utiliser)
docker network rm mon-network

# Nettoyer tous les networks non utilisés
docker network prune
```

EXERCICE GUIDÉ 1 : Créer et utiliser un network custom

Objectif : Créer deux conteneurs qui communiquent via un network personnalisé

Étape 1 : Créer le network

```
docker network create app-network
```

Étape 2 : Lancer un conteneur "serveur"

```
# On lance un conteneur Alpine qui restera actif
docker run -dit --name serveur --network app-network alpine sh

# Installer des outils réseau dedans
docker exec serveur apk add --no-cache curl
```

Étape 3 : Lancer un conteneur "client"

```
docker run -dit --name client --network app-network alpine sh

# Installer les outils
docker exec client apk add --no-cache curl
```

Étape 4 : Tester la communication

```
# Depuis le client, pinguer le serveur par son nom
docker exec client ping -c 3 serveur

# Vous devriez voir des réponses !
# PING serveur (172.X.X.X): 56 data bytes
# 64 bytes from 172.X.X.X: seq=0 ttl=64 time=0.XXX ms
```

Étape 5 : Inspecter le network

```
docker network inspect app-network

# Vous verrez les deux conteneurs listés avec leurs IPs
```

Étape 6 : Tester l'isolation

```
# Créer un second network
docker network create other-network

# Lancer un conteneur sur cet autre network
docker run -dit --name isole --network other-network alpine sh

# Essayer de pinguer depuis le client
docker exec client ping -c 3 isole

# Ça échoue ! "ping: bad address 'isole'"
# Les conteneurs sur des networks différents ne se voient pas
```

Nettoyage :

```
docker stop serveur client isole
docker rm serveur client isole
docker network rm app-network other-network
```

EXERCICE GUIDÉ 2 : Network dans docker-compose

Objectif : Créer une architecture avec plusieurs networks pour isoler les services

Scénario : On veut une API qui communique avec une BDD, mais on veut que la BDD ne soit PAS accessible depuis l'extérieur.

Créer un fichier docker-compose.yml :

```yaml
version: '3.8'

services:
  api:
    image: node:18-alpine
    command: sh -c "while true; do sleep 3600; done"
    networks:
      - frontend
      - backend
    ports:
      - "3000:3000"

  database:
    image: postgres:15-alpine
    environment:
      POSTGRES_PASSWORD: secret
    networks:
      - backend  # Seulement sur le backend

  nginx:
    image: nginx:alpine
    networks:
      - frontend  # Seulement sur le frontend
    ports:
      - "80:80"

networks:
  frontend:  # Network pour l'exposition publique
  backend:   # Network privé pour la BDD
```

Analyse de l'architecture :

- `nginx` et `api` sont sur le network `frontend`
- `api` et `database` sont sur le network `backend`
- `nginx` ne peut PAS communiquer directement avec `database` (isolation)
- `api` est sur les deux networks (pont entre les deux)

Tester :

```
# Lancer la stack
docker compose up -d

# Vérifier les networks créés
```

```
docker network ls | grep network

# Nginx peut atteindre l'API
docker compose exec nginx ping -c 2 api

# Nginx ne peut PAS atteindre la BDD
docker compose exec nginx ping -c 2 database
# Erreur attendue !

# Mais l'API peut atteindre la BDD
docker compose exec api ping -c 2 database
```

Nettoyage :

```
docker compose down
```

---

2 - Variables d'Environnement et Configuration

CORE NOTIONS

C'est quoi une variable d'environnement ?

Une variable d'environnement est une valeur (texte, nombre, URL) que vous passez à votre conteneur pour configurer son comportement sans modifier le code source.

Au lieu de coder en dur :

```
const dbHost = "localhost"; // En dur, mauvaise pratique
```

On utilise une variable d'environnement :

```
const dbHost = process.env.DB_HOST; // Flexible !
```

Pourquoi utiliser des variables d'environnement ?

1. Séparation du code et de la configuration

- Le même code peut tourner en développement, staging, production
- Juste en changeant les variables

2. Sécurité

- Les secrets (mots de passe, clés API) ne sont jamais dans le code
- Pas de risque de commit accidentel sur GitHub

3. Flexibilité

- Changement de configuration sans rebuild de l'image
- Différentes configurations selon l'environnement

Comment passer des variables d'environnement ?

Méthode 1 : Via docker run

```
docker run -e DB_HOST=postgres -e DB_PORT=5432 mon-image
```

Méthode 2 : Via docker-compose.yml (inline)

```
services:
  api:
    image: mon-api
    environment:
      - DB_HOST=postgres
      - DB_PORT=5432
      - NODE_ENV=production
```

Méthode 3 : Via docker-compose.yml (syntaxe objet)

```
services:
  api:
    image: mon-api
    environment:
      DB_HOST: postgres
      DB_PORT: 5432
      NODE_ENV: production
```

Méthode 4 : Via un fichier .env

```
# Créer un fichier .env
DB_HOST=postgres
DB_PORT=5432
NODE_ENV=production
# docker-compose.yml
services:
  api:
    image: mon-api
    env_file:
      - .env
```

Différence entre environment et env_file

environment : Variables explicites dans docker-compose.yml

- Avantage : tout est visible dans le fichier
- Inconvénient : on commit souvent le docker-compose.yml (risque de leak)

env_file : Variables dans un fichier séparé

- Avantage : on peut ajouter .env dans .gitignore (sécurité)
- Inconvénient : moins visible, fichier à maintenir séparément

Bonne pratique : Utiliser les deux !

```yaml
services:
  api:
    image: mon-api
    environment:
      NODE_ENV: production  # Pas sensible, peut être commité
    env_file:
      - .env  # Contient les secrets, dans .gitignore
```

Ordre de priorité des variables

Si une même variable est définie à plusieurs endroits, voici l'ordre (du plus prioritaire au moins) :

1. Variables passées avec `docker run -e`
2. Variables dans `environment:` du docker-compose.yml
3. Variables dans le fichier référencé par `env_file:`
4. Variables dans le Dockerfile (ENV)
5. Valeurs par défaut dans le code

Sécurité des secrets

Mauvaise pratique :

```yaml
# docker-compose.yml (commité sur GitHub)
services:
  api:
    environment:
      DB_PASSWORD: super_secret_123  # Visible sur GitHub !
```

Bonne pratique :

```yaml
# .env (dans .gitignore)
DB_PASSWORD=super_secret_123
# docker-compose.yml
services:
  api:
    env_file:
      - .env  # Le fichier .env n'est jamais commité
# .gitignore
.env
```

Encore mieux (pour la production) : Utiliser Docker Secrets ou des outils comme Vault, mais on ne le verra pas dans ce cours.

Utiliser les variables dans le code

Node.js :

```
const express = require('express');
const app = express();

const PORT = process.env.PORT || 3000;
const DB_HOST = process.env.DB_HOST || 'localhost';
const DB_PASSWORD = process.env.DB_PASSWORD;

if (!DB_PASSWORD) {
  throw new Error('DB_PASSWORD must be defined');
}

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
  console.log(`Connecting to database at ${DB_HOST}`);
});
```

Variables avec des valeurs par défaut

```
services:
  api:
    environment:
      DB_HOST: ${DB_HOST:-postgres}  # Si DB_HOST n'existe pas, utilise "postgres"
      DB_PORT: ${DB_PORT:-5432}
```

EXERCICE GUIDÉ 1 : Passer des variables d'environnement

Objectif : Créer une petite API qui lit des variables d'environnement

Étape 1 : Créer le code de l'API

```
// server.js
const express = require('express');
const app = express();

const PORT = process.env.PORT || 3000;
const API_KEY = process.env.API_KEY;
const ENVIRONMENT = process.env.NODE_ENV || 'development';

app.get('/config', (req, res) => {
  res.json({
    port: PORT,
    hasApiKey: !!API_KEY,
    environment: ENVIRONMENT,
    message: `Running in ${ENVIRONMENT} mode`
  });
});

app.listen(PORT, () => {
```

```
  console.log(`Server started on port ${PORT}`);
  console.log(`Environment: ${ENVIRONMENT}`);
  console.log(`API Key provided: ${!!API_KEY}`);
});
```

Étape 2 : Créer le Dockerfile

```
FROM node:18-alpine
WORKDIR /app
RUN npm init -y && npm install express
COPY server.js .
CMD ["node", "server.js"]
```

Étape 3 : Tester avec docker run

```
# Build l'image
docker build -t env-test .

# Lancer sans variables
docker run -p 3000:3000 env-test
# Regarder les logs : mode development, pas d'API key

# Lancer avec variables
docker run -p 3000:3000 \
  -e NODE_ENV=production \
  -e API_KEY=secret123 \
  -e PORT=3000 \
  env-test

# Tester
curl http://localhost:3000/config
```

Étape 4 : Utiliser docker-compose

```
version: '3.8'

services:
  api:
    build: .
    ports:
      - "3000:3000"
    environment:
      NODE_ENV: production
      API_KEY: secret123
      PORT: 3000
docker compose up
curl http://localhost:3000/config
```

EXERCICE GUIDÉ 2 : Utiliser un fichier .env

Objectif : Séparer les secrets dans un fichier .env

Étape 1 : Créer le fichier .env

```
# .env
NODE_ENV=production
API_KEY=super_secret_key_123
DB_HOST=postgres
DB_PORT=5432
DB_PASSWORD=db_secret_password
```

Étape 2 : Modifier docker-compose.yml

```
version: '3.8'

services:
  api:
    build: .
    ports:
      - "${PORT:-3000}:3000"  # Utilise PORT du .env ou 3000 par défaut
    env_file:
      - .env
```

Étape 3 : Ajouter .env au .gitignore

```
echo ".env" >> .gitignore
```

Étape 4 : Créer un .env.example (template pour l'équipe)

```
# .env.example (ce fichier EST commité)
NODE_ENV=development
API_KEY=your_api_key_here
DB_HOST=postgres
DB_PORT=5432
DB_PASSWORD=your_password_here
```

Étape 5 : Tester

```
docker compose up
curl http://localhost:3000/config
```

Chaque membre de l'équipe :

1. Clone le repo
2. Copie .env.example vers .env
3. Remplit ses propres valeurs
4. Lance docker compose up

3 - Expose vs Ports : Comprendre l'Exposition des Services

CORE NOTIONS

C'est quoi la différence ?

expose : Documente quel port le conteneur écoute, mais ne l'expose PAS à l'extérieur **ports :** Expose réellement le port à la machine hôte

C'est une source de confusion fréquente, alors clarifions.

La directive EXPOSE dans le Dockerfile

```
FROM node:18-alpine
WORKDIR /app
COPY . .
EXPOSE 3000  # Documentation uniquement !
CMD ["node", "server.js"]
```

Cette ligne EXPOSE 3000 est **purement documentaire**. Elle dit : "Hey, cette application écoute sur le port 3000 à l'intérieur du conteneur"

Mais elle ne rend PAS le port accessible depuis votre machine.

La directive ports dans docker-compose.yml

```
services:
  api:
    build: .
    ports:
      - "3000:3000"  # Exposition réelle !
```

Cette ligne crée un **mapping de port** :

- Port 3000 de votre machine → Port 3000 du conteneur
- Maintenant vous pouvez accéder à http://localhost:3000

Pourquoi deux mécanismes différents ?

EXPOSE (dans Dockerfile) :

- Documentation pour les développeurs
- Indique l'intention du créateur de l'image
- Utilisé par des outils d'orchestration (Kubernetes)
- N'a aucun effet sur l'accessibilité réseau

ports (dans docker-compose ou -p avec docker run) :

- Exposition réelle du service
- Décision de déploiement (pas de l'image)

- Permet l'accès depuis l'hôte ou l'extérieur

Syntaxes de ports dans docker-compose

Syntaxe courte :

```
ports:
  - "3000:3000"  # host:conteneur
  - "8080:80"    # hôte sur 8080, conteneur sur 80
```

Syntaxe longue (plus explicite) :

```
ports:
  - target: 3000      # Port du conteneur
    published: 3000   # Port de l'hôte
    protocol: tcp
```

Exposer sur localhost uniquement (sécurité) :

```
ports:
  - "127.0.0.1:3000:3000"  # Accessible seulement depuis la machine locale
```

Port dynamique (assigné par Docker) :

```
ports:
  - "3000"  # Docker choisit un port aléatoire sur l'hôte
```

Cas d'usage : quand utiliser expose vs ports

Utilisez expose quand :

- Services internes qui communiquent uniquement entre conteneurs
- Base de données accessible seulement par l'API
- Services dans un même docker-compose

Exemple :

```
services:
  api:
    ports:
      - "3000:3000"  # Accessible depuis l'extérieur

  database:
    image: postgres
    expose:
      - "5432"  # Seulement accessible par api, pas depuis l'hôte
```

Utilisez ports quand :

- Services qui doivent être accessibles depuis votre machine

- APIs publiques
- Interfaces web
- Services de développement

Sécurité et isolation

Mauvaise pratique :

```
services:
  database:
    image: postgres
    ports:
      - "5432:5432"  # La BDD est accessible depuis l'extérieur !
```

N'importe qui sur votre réseau peut tenter de se connecter à votre base de données.

Bonne pratique :

```
services:
  api:
    ports:
      - "3000:3000"

  database:
    image: postgres
    # Pas de ports ! Seulement accessible via le network Docker
```

L'API peut se connecter à la BDD via le network Docker, mais la BDD n'est pas exposée à l'extérieur.

EXERCICE GUIDÉ : Comprendre expose vs ports

Objectif : Créer une architecture où la BDD n'est pas exposée à l'hôte

Étape 1 : Créer une API simple qui se connecte à PostgreSQL

```
// app.js
const express = require('express');
const { Client } = require('pg');

const app = express();
const PORT = process.env.PORT || 3000;

const client = new Client({
  host: process.env.DB_HOST || 'database',
  port: 5432,
  user: 'postgres',
  password: process.env.DB_PASSWORD,
  database: 'testdb'
});

app.get('/health', (req, res) => {
  res.json({ status: 'ok' });
```

```
});

app.get('/db-test', async (req, res) => {
  try {
    await client.connect();
    const result = await client.query('SELECT NOW()');
    await client.end();
    res.json({
      success: true,
      time: result.rows[0].now
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: error.message
    });
  }
});

app.listen(PORT, () => {
  console.log(`API running on port ${PORT}`);
});
```

Étape 2 : Créer le docker-compose.yml

```
version: '3.8'

services:
  api:
    build: .
    ports:
      - "3000:3000"  # Exposé à l'hôte
    environment:
      DB_HOST: database
      DB_PASSWORD: secret
    depends_on:
      - database

  database:
    image: postgres:15-alpine
    environment:
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: testdb
    # Pas de ports ! Seulement accessible via le network interne
```

Étape 3 : Tester l'isolation

```
# Lancer la stack
docker compose up -d

# L'API est accessible
curl http://localhost:3000/health

# L'API peut se connecter à la BDD
curl http://localhost:3000/db-test

# Mais vous ne pouvez PAS vous connecter à la BDD depuis votre machine
psql -h localhost -U postgres -d testdb
# Connection refused ! (normal, le port n'est pas exposé)
```

Étape 4 : Exposer la BDD pour le debug (temporaire)

Modifiez le docker-compose.yml :

```
database:
  image: postgres:15-alpine
  ports:
    - "5432:5432"  # Maintenant exposé
  environment:
    POSTGRES_PASSWORD: secret
    POSTGRES_DB: testdb
docker compose down
docker compose up -d

# Maintenant vous pouvez vous connecter depuis l'hôte
psql -h localhost -U postgres -d testdb
# Succès !
```

En production, retirez le `ports:` pour sécuriser la base de données.

---

4 - Redis en Détail : Cache et Persistance

CORE NOTIONS

C'est quoi Redis ?

Redis (Remote Dictionary Server) est une base de données **en mémoire** ultra-rapide qui stocke des paires clé-valeur.

Contrairement à PostgreSQL ou MySQL qui écrivent sur le disque, Redis stocke tout en RAM. C'est pour ça qu'il est extrêmement rapide (lecture/écriture en microsecondes).

Pourquoi utiliser Redis ?

1. Cache Le use case le plus courant. Au lieu de requêter votre base de données à chaque fois, vous mettez les résultats en cache dans Redis.

```
Sans cache :
Requête → API → PostgreSQL (50ms) → API → Réponse

Avec cache :
Requête → API → Redis (1ms) → API → Réponse
```

2. Session store Stocker les sessions utilisateur (qui est connecté, leur panier, etc.)

3. Rate limiting Limiter le nombre de requêtes par utilisateur/IP

4. File d'attente (queue) Gérer des tâches asynchrones (envoi d'emails, traitement d'images)

5. Pub/Sub Messagerie en temps réel entre services

Comment ça marche ?

Redis stocke des structures de données simples :

Strings :

```
SET user:1:name "Alice"
GET user:1:name  # Retourne "Alice"
```

Hash (objet) :

```
HSET user:1 name "Alice" age "30"
HGET user:1 name  # Retourne "Alice"
```

Lists :

```
LPUSH queue:emails "email1@example.com"
LPUSH queue:emails "email2@example.com"
RPOP queue:emails  # Retourne "email1@example.com"
```

Sets :

```
SADD tags:article:1 "docker" "devops" "tutorial"
SMEMBERS tags:article:1  # Retourne ["docker", "devops", "tutorial"]
```

Expiration automatique (TTL) :

```
SET session:abc123 "user_data" EX 3600  # Expire dans 1 heure
```

Utiliser Redis avec Node.js

```
const redis = require('redis');
const client = redis.createClient({
  host: process.env.REDIS_HOST || 'localhost',
  port: 6379
});
```

```
await client.connect();

// Exemple 1 : Cache simple
async function getUser(userId) {
  // Vérifier le cache d'abord
  const cached = await client.get(`user:${userId}`);
  if (cached) {
    return JSON.parse(cached);
  }

  // Sinon, requête à la BDD
  const user = await database.query('SELECT * FROM users WHERE id = $1', [userId]);

  // Mettre en cache pour 5 minutes
  await client.setEx(`user:${userId}`, 300, JSON.stringify(user));

  return user;
}

// Exemple 2 : Rate limiting
async function checkRateLimit(ip) {
  const key = `ratelimit:${ip}`;
  const requests = await client.incr(key);

  if (requests === 1) {
    // Première requête, on définit l'expiration à 1 minute
    await client.expire(key, 60);
  }

  if (requests > 100) {
    throw new Error('Too many requests');
  }

  return requests;
}
```

Configuration de persistance Redis

Par défaut, Redis stocke tout en RAM. Si le serveur redémarre, tout est perdu.

Pour éviter ça, Redis propose deux modes de persistance :

1. RDB (Redis Database Backup)

- Snapshots périodiques du dataset
- Sauvegarde compacte
- Rapide à charger
- Peut perdre les dernières minutes de données

Configuration dans `redis.conf` :

```
save 900 1      # Sauvegarde si au moins 1 clé changée en 15 min
save 300 10     # Sauvegarde si au moins 10 clés changées en 5 min
save 60 10000   # Sauvegarde si au moins 10000 clés changées en 1 min
```

2. AOF (Append Only File)

- Log de toutes les opérations d'écriture
- Plus sûr (perte maximale de 1 seconde de données)
- Fichier plus volumineux
- Plus lent à charger

Configuration dans `redis.conf` :

```
appendonly yes
appendfsync everysec  # Synchronise toutes les secondes
```

Avec Docker, utiliser un volume pour la persistance :

```yaml
services:
  redis:
    image: redis:7-alpine
    command: redis-server --appendonly yes  # Active AOF
    volumes:
      - redis-data:/data  # Le dossier /data contient les fichiers de persistance

volumes:
  redis-data:
```

Commandes Redis utiles

```
# Se connecter au CLI Redis
docker exec -it mon-redis redis-cli

# Lister toutes les clés
KEYS *

# Voir une valeur
GET ma-cle

# Supprimer une clé
DEL ma-cle

# Voir le temps restant avant expiration (en secondes)
TTL ma-cle

# Vider toute la base de données (attention !)
FLUSHDB

# Voir les statistiques
INFO
```

```
# Tester la connexion
PING  # Retourne PONG
```

EXERCICE GUIDÉ 1 : Utiliser Redis comme cache

Objectif : Créer une API qui utilise Redis pour mettre en cache des résultats lents

Étape 1 : Créer l'API avec cache

```
// app.js
const express = require('express');
const redis = require('redis');

const app = express();
const redisClient = redis.createClient({
  host: 'redis',
  port: 6379
});

redisClient.connect();

// Simulation d'une requête lente (BDD, API externe, etc.)
function slowQuery() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ data: 'Résultat de la requête lente', timestamp: Date.now() });
    }, 3000); // 3 secondes
  });
}

app.get('/data', async (req, res) => {
  const cacheKey = 'slow-data';

  // Vérifier le cache
  const cached = await redisClient.get(cacheKey);

  if (cached) {
    return res.json({
      source: 'cache',
      data: JSON.parse(cached)
    });
  }

  // Pas en cache, exécuter la requête lente
  const result = await slowQuery();

  // Mettre en cache pour 60 secondes
  await redisClient.setEx(cacheKey, 60, JSON.stringify(result));

  res.json({
    source: 'database',
    data: result
```

```
  });
});

app.listen(3000, () => {
  console.log('API running on port 3000');
});
```

Étape 2 : Créer le docker-compose.yml

```yaml
version: '3.8'

services:
  api:
    build: .
    ports:
      - "3000:3000"
    depends_on:
      - redis

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"  # Exposé pour debug
```

Étape 3 : Tester le cache

```
docker compose up -d

# Première requête (lente, pas de cache)
time curl http://localhost:3000/data
# Devrait prendre ~3 secondes, source: "database"

# Deuxième requête (rapide, depuis le cache)
time curl http://localhost:3000/data
# Devrait prendre quelques millisecondes, source: "cache"

# Attendre 60 secondes et re-tester
sleep 61
curl http://localhost:3000/data
# Le cache a expiré, redevient lent et source: "database"
```

Étape 4 : Explorer Redis

```
# Se connecter au CLI Redis
docker exec -it <redis-container-id> redis-cli

# Lister les clés
KEYS *

# Voir la valeur
GET slow-data
```

```
# Voir le temps restant avant expiration
TTL slow-data

# Supprimer la clé manuellement
DEL slow-data

# Retester l'API, le cache est vide
```

EXERCICE GUIDÉ 2 : Persistance Redis avec volume

Objectif : Configurer Redis pour que les données survivent au redémarrage

Étape 1 : Modifier docker-compose.yml

```yaml
version: '3.8'

services:
  api:
    build: .
    ports:
      - "3000:3000"
    depends_on:
      - redis

  redis:
    image: redis:7-alpine
    command: redis-server --appendonly yes  # Active AOF
    volumes:
      - redis-data:/data
    ports:
      - "6379:6379"

volumes:
  redis-data:
```

Étape 2 : Tester la persistance

```
docker compose up -d

# Créer des données
curl http://localhost:3000/data

# Vérifier dans Redis
docker exec -it <redis-container> redis-cli
KEYS *

# Arrêter tout
docker compose down

# Relancer
docker compose up -d
```

```
# Vérifier que les données existent toujours
docker exec -it <redis-container> redis-cli
KEYS *  # Les clés sont toujours là !
```

Étape 3 : Voir les fichiers de persistance

```
# Inspecter le volume
docker volume inspect redis-data

# Les fichiers sont dans /var/lib/docker/volumes/redis-data/_data
# Contient appendonly.aof (log de toutes les opérations)
```

---

5 - Nginx comme Reverse Proxy

CORE NOTIONS

C'est quoi un reverse proxy ?

Un reverse proxy est un serveur qui se place devant vos applications et qui redistribue les requêtes.

```
Client → Reverse Proxy → Backend 1
                       → Backend 2
                       → Backend 3
```

C'est l'inverse d'un proxy classique :

- **Proxy classique :** protège le client (VPN, cache)
- **Reverse proxy :** protège les serveurs backend

Pourquoi utiliser Nginx comme reverse proxy ?

1. Point d'entrée unique

```
http://monsite.com/api → API Node.js (port 3000)
http://monsite.com/admin → Interface admin (port 4000)
http://monsite.com/ → Frontend React (port 5000)
```

Un seul domaine, plusieurs services derrière.

2. Load balancing Répartir la charge entre plusieurs instances d'une même application :

```
Client → Nginx → API instance 1
              → API instance 2
              → API instance 3
```

3. SSL/TLS (HTTPS) Nginx gère le certificat SSL, les backends peuvent rester en HTTP

4. Cache Nginx peut mettre en cache des réponses pour réduire la charge

5. Sécurité

- Rate limiting
- Protection DDoS
- Headers de sécurité
- Les backends ne sont jamais exposés directement

6. Logs et monitoring Centraliser les logs d'accès

Configuration Nginx de base

Un fichier de configuration Nginx se compose de blocs :

```
# nginx.conf

events {
    worker_connections 1024;  # Nombre de connexions simultanées
}

http {
    # Configuration HTTP globale

    server {
        listen 80;  # Écoute sur le port 80

        location / {
            # Configuration pour une route
        }
    }
}
```

Exemple : Reverse proxy vers une API

```
events {
    worker_connections 1024;
}

http {
    upstream api {
        server api:3000;  # Nom du service dans docker-compose
    }

    server {
        listen 80;

        location /api {
            proxy_pass http://api;
            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;
        }
```

```
    }
}
```

Explications :

- `upstream api` : définit un groupe de serveurs backend
- `server api:3000` : le service "api" du docker-compose sur le port 3000
- `location /api` : toutes les requêtes vers /api sont redirigées
- `proxy_pass http://api` : redirige vers le backend défini dans upstream
- `proxy_set_header` : transmet les headers importants au backend

Headers importants

X-Real-IP : L'IP réelle du client (sinon le backend voit l'IP de Nginx) **X-Forwarded-For :** Liste des IPs dans la chaîne de proxies **X-Forwarded-Proto :** Le protocole original (http ou https) **Host :** Le nom de domaine demandé par le client

Ces headers permettent à votre API de savoir qui fait réellement la requête.

Plusieurs services derrière Nginx

```
events {
    worker_connections 1024;
}

http {
    upstream api {
        server api:3000;
    }

    upstream frontend {
        server frontend:5000;
    }

    server {
        listen 80;

        # API
        location /api/ {
            proxy_pass http://api/;
        }

        # Frontend
        location / {
            proxy_pass http://frontend/;
        }
    }
}
```

Gestion des erreurs

```
server {
    listen 80;

    location /api {
        proxy_pass http://api;

        # Si le backend est down, retourner une erreur custom
        proxy_intercept_errors on;
        error_page 502 503 504 /50x.html;
    }

    location = /50x.html {
        return 503 "Service temporarily unavailable";
    }
}
```

Logs

Nginx crée deux types de logs :

access.log : Toutes les requêtes

```
127.0.0.1 - - [01/Jan/2025:10:00:00 +0000] "GET /api/users HTTP/1.1" 200 1234
```

error.log : Erreurs uniquement

```
2025/01/01 10:00:00 [error] 1#1: *1 connect() failed (111: Connection refused)
```

Configuration :

```
http {
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;

    server {
        # ...
    }
}
```

Load balancing

Répartir la charge entre plusieurs instances :

```
upstream api {
    server api1:3000;
    server api2:3000;
    server api3:3000;
}

server {
    listen 80;

    location /api {
        proxy_pass http://api;  # Nginx répartit automatiquement
    }
}
```

Stratégies de load balancing :

```
# Round-robin (par défaut) : tour à tour
upstream api {
    server api1:3000;
    server api2:3000;
}

# Least connections : vers le serveur le moins chargé
upstream api {
    least_conn;
    server api1:3000;
    server api2:3000;
}

# IP hash : même client → même serveur (pour les sessions)
upstream api {
    ip_hash;
    server api1:3000;
    server api2:3000;
}
```

EXERCICE GUIDÉ 1 : Nginx devant une API

Objectif : Configurer Nginx comme reverse proxy devant une API Node.js

Étape 1 : Créer une API simple

```
// app.js
const express = require('express');
const app = express();

app.get('/api/hello', (req, res) => {
  res.json({
    message: 'Hello from API',
    headers: req.headers  // On verra les headers transmis par Nginx
  });
});
```

```
app.listen(3000, () => {
  console.log('API running on port 3000');
});
```

Étape 2 : Créer la configuration Nginx

```
# nginx/nginx.conf
events {
    worker_connections 1024;
}

http {
    upstream api {
        server api:3000;
    }

    server {
        listen 80;

        location /api/ {
            proxy_pass http://api/;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }

        location /health {
            return 200 "Nginx OK";
        }
    }
}
```

Étape 3 : Créer docker-compose.yml

```
version: '3.8'

services:
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - api

  api:
    build: .
    expose:
      - "3000"  # Pas de ports, seulement accessible via Nginx
```

Étape 4 : Tester

```
docker compose up -d

# Tester Nginx directement
curl http://localhost/health

# Tester l'API via Nginx
curl http://localhost/api/hello

# Essayer d'accéder directement à l'API (échec)
curl http://localhost:3000/api/hello
# Connection refused (normal, le port n'est pas exposé)
```

Étape 5 : Observer les headers transmis

```
curl http://localhost/api/hello

# Vous verrez dans la réponse les headers ajoutés par Nginx :
# {
#   "message": "Hello from API",
#   "headers": {
#     "x-real-ip": "172.18.0.1",
#     "x-forwarded-for": "172.18.0.1",
#     "host": "localhost"
#   }
# }
```

EXERCICE GUIDÉ 2 : Nginx avec plusieurs services

Objectif : Router différents chemins vers différents services

Étape 1 : Créer deux APIs

```
// api1/app.js
const express = require('express');
const app = express();

app.get('/users', (req, res) => {
  res.json({ service: 'users-api', data: ['Alice', 'Bob'] });
});

app.listen(3000);

// api2/app.js
const express = require('express');
const app = express();

app.get('/products', (req, res) => {
  res.json({ service: 'products-api', data: ['Laptop', 'Phone'] });
});
```

```
app.listen(4000);
```

Étape 2 : Configuration Nginx

```
# nginx/nginx.conf
events {
    worker_connections 1024;
}

http {
    upstream users_api {
        server api1:3000;
    }

    upstream products_api {
        server api2:4000;
    }

    server {
        listen 80;

        # Route /users vers api1
        location /users {
            proxy_pass http://users_api;
        }

        # Route /products vers api2
        location /products {
            proxy_pass http://products_api;
        }

        # Page d'accueil
        location / {
            return 200 "Welcome to the API Gateway\n";
        }
    }
}
```

Étape 3 : docker-compose.yml

```
version: '3.8'

services:
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - api1
      - api2
```

```
  api1:
    build: ./api1
    expose:
      - "3000"

  api2:
    build: ./api2
    expose:
      - "4000"
```

Étape 4 : Tester

```
docker compose up -d

# Router vers users-api
curl http://localhost/users

# Router vers products-api
curl http://localhost/products

# Page d'accueil
curl http://localhost/
```

---

6 - Debug & Troubleshooting (Approfondissement)

CORE NOTIONS

Vous connaissez déjà les commandes de base vues hier. Maintenant, allons plus loin dans le diagnostic de problèmes complexes.

Analyser les logs comme un pro

Logs en temps réel avec filtrage :

```
# Suivre les logs d'un seul service
docker compose logs -f api

# Voir les 100 dernières lignes
docker compose logs --tail=100 api

# Filtrer par timestamp
docker compose logs --since 30m api  # Dernières 30 minutes
docker compose logs --until 2h api   # Il y a 2 heures

# Combiner plusieurs services
docker compose logs -f api database redis
```

Logs avec grep :

```
# Trouver les erreurs
docker logs mon-conteneur 2>&1 | grep -i error

# Compter les erreurs
docker logs mon-conteneur 2>&1 | grep -i error | wc -l

# Trouver les requêtes lentes (>1000ms)
docker logs mon-conteneur | grep "took [0-9]\{4,\}ms"
```

Inspecter un conteneur en profondeur

```
# Voir toute la configuration d'un conteneur
docker inspect mon-conteneur

# Extraire des infos spécifiques avec --format
docker inspect --format='{{.State.Status}}' mon-conteneur
docker inspect --format='{{.NetworkSettings.IPAddress}}' mon-conteneur
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' mon-
conteneur

# Voir les variables d'environnement
docker inspect --format='{{.Config.Env}}' mon-conteneur

# Voir les volumes montés
docker inspect --format='{{.Mounts}}' mon-conteneur
```

Débugger les problèmes réseau

Tester la connectivité entre conteneurs :

```
# Depuis un conteneur, pinguer un autre
docker exec conteneur1 ping -c 3 conteneur2

# Tester un port spécifique avec telnet
docker exec conteneur1 apk add busybox-extras  # Installer telnet
docker exec conteneur1 telnet conteneur2 5432

# Tester avec curl
docker exec conteneur1 curl http://conteneur2:3000/health

# Voir les connexions actives
docker exec conteneur1 netstat -an
```

Inspecter un network :

```
# Voir tous les conteneurs sur un network
docker network inspect mon-network

# Trouver l'IP d'un conteneur
docker network inspect mon-network | grep -A 3 "nom-conteneur"
```

Débugger les problèmes de volumes

Vérifier qu'un volume est bien monté :

```
# Lister les volumes
docker volume ls

# Voir les détails d'un volume
docker volume inspect mon-volume

# Trouver où le volume est stocké sur l'hôte
docker volume inspect --format '{{ .Mountpoint }}' mon-volume

# Voir quels conteneurs utilisent un volume
docker ps -a --filter volume=mon-volume
```

Accéder au contenu d'un volume :

```
# Via un conteneur temporaire
docker run --rm -v mon-volume:/data alpine ls -la /data

# Copier des fichiers depuis un volume
docker run --rm -v mon-volume:/data -v $(pwd):/backup alpine cp -r /data /backup

# Copier des fichiers vers un volume
docker run --rm -v mon-volume:/data -v $(pwd):/backup alpine cp /backup/fichier.txt /data/
```

Débugger les problèmes de build

Voir l'historique des layers :

```
docker history mon-image

# Avec les tailles
docker history --no-trunc mon-image
```

Build avec plus de verbosité :

```
# Voir toutes les étapes
docker build --progress=plain -t mon-image .

# Sans cache (pour debugger des problèmes de cache)
docker build --no-cache -t mon-image .

# Build jusqu'à une étape spécifique (multi-stage)
docker build --target builder -t mon-image .
```

Analyser les performances

Voir l'utilisation des ressources :

```
# Stats en temps réel
docker stats

# Stats d'un conteneur spécifique
docker stats mon-conteneur --no-stream

# Format custom
docker stats --format "table {{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}"
```

Voir les processus dans un conteneur :

```
docker top mon-conteneur

# Avec plus de détails
docker top mon-conteneur aux
```

Les erreurs cryptiques et leurs solutions

Erreur : "driver failed programming external connectivity"

```
Cause : Le port est déjà utilisé sur l'hôte
Solution :
- Trouver qui utilise le port : lsof -i :3000
- Changer le port dans docker-compose : "3001:3000"
- Stopper le service qui utilise le port
```

Erreur : "no space left on device"

```
Cause : Docker a rempli le disque
Solution :
- docker system df  # Voir l'espace utilisé
- docker system prune -a  # Nettoyer tout
- docker volume prune  # Nettoyer les volumes non utilisés
```

Erreur : "network not found"

```
Cause : Le network a été supprimé ou n'existe pas
Solution :
- docker network ls  # Lister les networks
- docker network create mon-network  # Recréer le network
- docker compose down && docker compose up  # Recréer via compose
```

Erreur : "service unhealthy"

```
Cause : Le healthcheck échoue
Solution :
- docker logs mon-conteneur  # Voir pourquoi le service crash
- docker inspect mon-conteneur  # Voir la config du healthcheck
- Tester le healthcheck manuellement :
  docker exec mon-conteneur curl http://localhost/health
```

Débugguer un conteneur qui crash immédiatement

```
# Voir les logs même si le conteneur est arrêté
docker logs conteneur-qui-crash

# Lancer le conteneur avec un shell au lieu de la commande par défaut
docker run -it --entrypoint sh mon-image

# Avec docker-compose
docker compose run --entrypoint sh mon-service
```

Nettoyer Docker complètement

```
# Nettoyer les conteneurs arrêtés
docker container prune

# Nettoyer les images non utilisées
docker image prune -a

# Nettoyer les volumes non utilisés
docker volume prune

# Nettoyer les networks non utilisés
docker network prune

# Tout nettoyer d'un coup (attention, destructif !)
docker system prune -a --volumes
```

---

7 - Optimisation d'Images (Approfondissement)

CORE NOTIONS

Vous avez vu les bases de l'optimisation hier. Allons plus loin avec des techniques avancées.

Analyse de la taille d'une image

Voir les layers et leur taille :

```
docker history mon-image

# Sortie exemple :
# IMAGE          CREATED        SIZE      COMMENT
# abc123         2 mins ago     50MB      CMD ["node", "server.js"]
# def456         5 mins ago     200MB     RUN npm install
# ghi789         10 mins ago    100MB     COPY . .
```

Chaque ligne est un layer. Les plus gros layers sont vos cibles d'optimisation.

Outils d'analyse :

```
# Dive : analyser une image layer par layer
# Installation : https://github.com/wagoodman/dive
dive mon-image

# Docker Slim : réduire automatiquement la taille
# Installation : https://github.com/slimtoolkit/slim
slim build mon-image
```

Multi-stage builds avancés

Exemple avec tests et build :

```
# Stage 1 : Tests
FROM node:18 AS tester
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm test

# Stage 2 : Build
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
RUN npm run build

# Stage 3 : Production
FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
USER node  # Ne pas run en root
CMD ["node", "dist/server.js"]
```

Les stages "tester" et "builder" ne sont pas dans l'image finale, seulement leurs artefacts copiés.

Optimiser node_modules

Problème : node_modules peut être énorme (500MB+)

Solution 1 : npm ci au lieu de npm install

```
# Lent et variable
RUN npm install

# Rapide et déterministe
RUN npm ci --only=production
```

Solution 2 : Nettoyer le cache npm

```
RUN npm ci --only=production && \
    npm cache clean --force
```

Solution 3 : Utiliser .dockerignore pour node_modules

```
# .dockerignore
node_modules
npm-debug.log
```

Toujours installer depuis package.json dans le conteneur, jamais copier node_modules local.

Combiner les commandes RUN intelligemment

Mauvais (3 layers) :

```
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get clean
```

Bon (1 layer) :

```
RUN apt-get update && \
    apt-get install -y curl && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

Chaque RUN crée un layer. Moins de layers = image plus petite.

Ordre optimal des instructions

Docker cache les layers. Si un layer change, tous les layers suivants sont reconstruits.

Mauvais ordre :

```
FROM node:18-alpine
WORKDIR /app
COPY . .                  # Change souvent
RUN npm install           # Recalculé à chaque changement de code
CMD ["node", "server.js"]
```

Bon ordre :

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./     # Change rarement
RUN npm install           # Utilisé depuis le cache si package.json inchangé
COPY . .                  # Change souvent
CMD ["node", "server.js"]
```

.dockerignore exhaustif

```
# .dockerignore

# Dependencies
node_modules
npm-debug.log
yarn-error.log

# Tests
tests/
**/*.test.js
coverage/
.nyc_output/

# Documentation
README.md
CHANGELOG.md
docs/

# Git
.git
.gitignore
.gitattributes

# CI/CD
.github/
.gitlab-ci.yml
Jenkinsfile

# Editor
.vscode/
.idea/
*.swp
*.swo
```

```
# OS
.DS_Store
Thumbs.db

# Environment
.env
.env.local
.env.*.local

# Build artifacts
dist/
build/
*.log
```

Utiliser des images de base plus petites

Comparaison de tailles :

```
node:18             → 910 MB
node:18-slim        → 170 MB
node:18-alpine      → 110 MB
```

Attention avec Alpine : Certains packages npm nécessitent des outils de compilation. Si npm install échoue sur Alpine :

```
FROM node:18-alpine

# Installer les outils de build nécessaires
RUN apk add --no-cache python3 make g++

WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production

# Optionnel : supprimer les outils de build après installation
RUN apk del python3 make g++

COPY . .
CMD ["node", "server.js"]
```

Créer des images spécifiques par environnement

```
# Dockerfile.dev
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install  # Inclut devDependencies
COPY . .
CMD ["npm", "run", "dev"]
# Dockerfile.prod
FROM node:18-alpine AS builder
```

```
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
RUN npm run build

FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
USER node
CMD ["node", "dist/server.js"]
# docker-compose.yml
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile.${ENV:-dev}  # Utilise Dockerfile.dev ou Dockerfile.prod
# Développement
docker compose up

# Production
ENV=prod docker compose up
```

BuildKit pour des builds plus rapides

BuildKit est le nouveau moteur de build Docker, plus rapide et avec plus de fonctionnalités.

Activer BuildKit :

```
export DOCKER_BUILDKIT=1
docker build -t mon-image .
```

Avantages :

- Build parallèle des layers indépendants
- Cache plus intelligent
- Syntaxe avancée (secrets, ssh)

Exemple avec cache mount :

```
# syntax=docker/dockerfile:1

FROM node:18-alpine

WORKDIR /app

# Utiliser un cache mount pour npm
RUN --mount=type=cache,target=/root/.npm \
    npm ci --only=production

COPY . .
CMD ["node", "server.js"]
```

Le cache npm est partagé entre builds, accélérant les installations suivantes.

---

8 - Healthchecks et Availability (Approfondissement)

CORE NOTIONS

Un healthcheck est un test automatique que Docker exécute régulièrement pour vérifier qu'un service fonctionne correctement.

Pourquoi les healthchecks sont critiques

Sans healthcheck :

```
Docker lance le conteneur → RUNNING
Mais l'application peut :
- Crasher après 2 secondes
- Être bloquée dans une boucle infinie
- Ne pas répondre aux requêtes
- Avoir une connexion BDD cassée

Docker pense que tout va bien !
```

Avec healthcheck :

```
Docker lance le conteneur → STARTING
Healthcheck réussit → HEALTHY
Healthcheck échoue 3 fois → UNHEALTHY
Docker peut redémarrer le conteneur automatiquement
```

Différence entre "started" et "healthy"

depends_on sans healthcheck :

```yaml
services:
  api:
    depends_on:
      - database  # API démarre dès que le conteneur database est STARTED
```

Problème : PostgreSQL prend 5-10 secondes pour être prêt. L'API démarre trop tôt et crash car la BDD n'est pas prête.

depends_on avec healthcheck :

```yaml
services:
  api:
    depends_on:
      database:
        condition: service_healthy  # API attend que database soit HEALTHY

  database:
    image: postgres:15-alpine
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5
      start_period: 10s
```

Maintenant l'API démarre seulement quand PostgreSQL répond vraiment.

Configurer un healthcheck dans le Dockerfile

```dockerfile
FROM node:18-alpine

WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .

EXPOSE 3000

# Installer curl pour le healthcheck
RUN apk add --no-cache curl

# Définir le healthcheck
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1

CMD ["node", "server.js"]
```

Paramètres :

- `--interval=30s` : Test toutes les 30 secondes
- `--timeout=3s` : Le test doit répondre en 3 secondes max

- `--start-period=5s` : Attendre 5 secondes avant de commencer les tests (temps de démarrage)
- `--retries=3` : Marquer comme UNHEALTHY après 3 échecs consécutifs

Configurer un healthcheck dans docker-compose

```
services:
  api:
    build: .
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 3s
      retries: 3
      start_period: 5s
```

Syntaxes alternatives :

```
# Avec shell
healthcheck:
  test: curl -f http://localhost:3000/health || exit 1

# Avec CMD-SHELL
healthcheck:
  test: ["CMD-SHELL", "curl -f http://localhost:3000/health"]

# Sans curl (Node.js peut faire une requête à lui-même)
healthcheck:
  test: ["CMD", "node", "-e", "require('http').get('http://localhost:3000/health', (r) =>
process.exit(r.statusCode === 200 ? 0 : 1))"]
```

Créer un endpoint /health dans votre API

Node.js/Express :

```
const express = require('express');
const app = express();

// Healthcheck simple
app.get('/health', (req, res) => {
  res.status(200).json({ status: 'ok' });
});

// Healthcheck avancé (vérifie les dépendances)
app.get('/health', async (req, res) => {
  try {
    // Vérifier la connexion BDD
    await database
```

.query('SELECT 1');

```
// Vérifier Redis
await redis.ping();

res.status(200).json({
  status: 'healthy',
  timestamp: new Date().toISOString(),
  checks: {
    database: 'ok',
    redis: 'ok'
  }
});
```

} catch (error) { res.status(503).json({ status: 'unhealthy', error: error.message }); } });

app.listen(3000);

```
#### Healthchecks pour différents types de services

**PostgreSQL :**
```yaml
healthcheck:
  test: ["CMD-SHELL", "pg_isready -U postgres"]
  interval: 10s
  timeout: 5s
  retries: 5
```

MySQL :

```
healthcheck:
  test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
  interval: 10s
  timeout: 5s
  retries: 5
```

Redis :

```
healthcheck:
  test: ["CMD", "redis-cli", "ping"]
  interval: 5s
  timeout: 3s
  retries: 5
```

Nginx :
```

```yaml
healthcheck:
  test: ["CMD-SHELL", "curl -f http://localhost/health || exit 1"]
  interval: 30s
  timeout: 3s
  retries: 3
```

Voir le statut des healthchecks

```bash
# Voir le statut dans docker ps
docker ps

# La colonne STATUS affiche :
# Up 2 minutes (healthy)
# Up 2 minutes (health: starting)
# Up 2 minutes (unhealthy)

# Inspecter les détails du healthcheck
docker inspect --format='{{json .State.Health}}' mon-conteneur | jq

# Sortie exemple :
# {
#   "Status": "healthy",
#   "FailingStreak": 0,
#   "Log": [
#     {
#       "Start": "2025-01-01T10:00:00Z",
#       "End": "2025-01-01T10:00:01Z",
#       "ExitCode": 0,
#       "Output": "..."
#     }
#   ]
# }
```

Conditions de dépendance avancées

```yaml
version: '3.8'

services:
  frontend:
    depends_on:
      api:
        condition: service_healthy  # Attend que l'API soit healthy

  api:
    depends_on:
      database:
        condition: service_healthy  # Attend que la BDD soit healthy
      redis:
        condition: service_started  # Attend juste que Redis démarre
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
```

```
      interval: 10s
      timeout: 5s
      retries: 3

  database:
    image: postgres:15-alpine
    healthcheck:
      test: ["CMD-SHELL", "pg_isready"]
      interval: 5s
      timeout: 5s
      retries: 5

  redis:
    image: redis:7-alpine
    # Pas de healthcheck, on attend juste qu'il démarre
```

Ordre de démarrage :

1. database et redis démarrent en parallèle
2. database devient healthy
3. api démarre et attend de devenir healthy
4. api devient healthy
5. frontend démarre

Restart policies avec healthcheck

```
services:
  api:
    restart: unless-stopped  # Redémarre automatiquement si le conteneur s'arrête
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 3s
      retries: 3
```

Options de restart :

- `no` : Ne jamais redémarrer (défaut)
- `always` : Toujours redémarrer
- `on-failure` : Redémarrer seulement si le conteneur crash
- `unless-stopped` : Redémarrer sauf si stoppé manuellement

Combiné avec healthcheck : Si le service devient unhealthy, Docker ne le redémarre pas automatiquement. Il faut utiliser un orchestrateur comme Kubernetes pour ça.

Désactiver temporairement un healthcheck

```yaml
services:
  api:
    healthcheck:
      disable: true  # Utile pour le debug
```

---

9 - Architecture Multi-Services (Approfondissement)

CORE NOTIONS

Une architecture multi-services (microservices) sépare votre application en plusieurs services indépendants qui communiquent entre eux.

Principes d'une bonne architecture

1. Séparation des responsabilités

Chaque service a une responsabilité unique et bien définie.

```
Mauvais (monolithe) :
[Application] → Base de données

Bon (multi-services) :
[Frontend]
    ↓
[API Gateway / Nginx]
    ↓
[Service Users] → [DB Users]
[Service Products] → [DB Products]
[Service Orders] → [DB Orders]
    ↓
[Service Notifications]
[Service Payments]
```

2. Isolation des données

Chaque service a sa propre base de données (ou schéma). Les services ne partagent jamais de BDD directement.

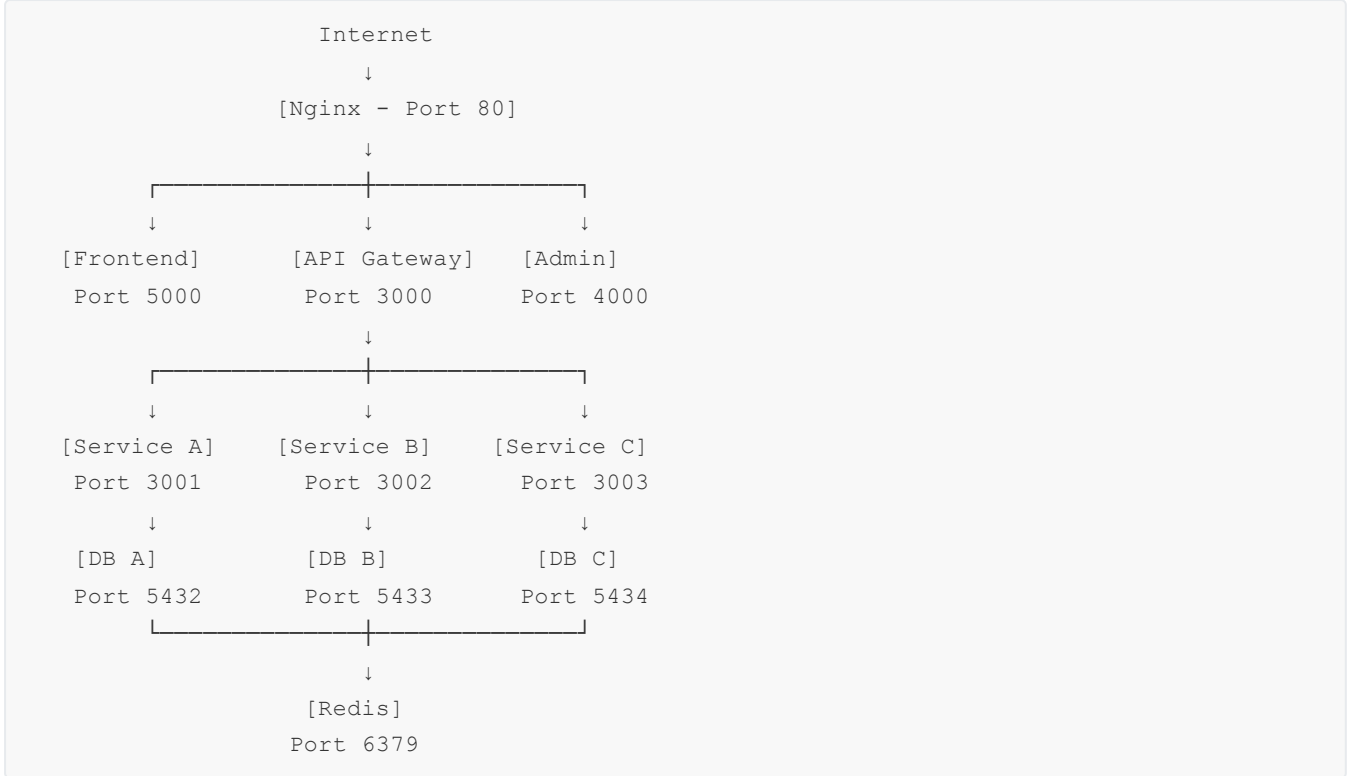
3. Communication via API

Les services communiquent uniquement par HTTP/REST, message queues, ou gRPC. Jamais d'accès direct à la BDD d'un autre service.

4. Résilience

Si un service tombe, les autres continuent de fonctionner (mode dégradé).

Architecture typique avec Docker



Networks pour isoler les services

```yaml
version: '3.8'

services:
  nginx:
    image: nginx:alpine
    networks:
      - public
    ports:
      - "80:80"

  frontend:
    build: ./frontend
    networks:
      - public

  api-gateway:
    build: ./api-gateway
    networks:
      - public
      - backend

  service-users:
    build: ./services/users
    networks:
      - backend

  service-products:
```

```
    build: ./services/products
    networks:
      - backend

  db-users:
    image: postgres:15-alpine
    networks:
      - backend

  db-products:
    image: postgres:15-alpine
    networks:
      - backend

networks:
  public:    # Frontend, Nginx
  backend:   # API Gateway, Services, Databases
```

Principe :

- Le `public` network expose seulement Nginx et Frontend
- Le `backend` network isole les services et bases de données
- L'API Gateway fait le pont entre les deux

Communication inter-services

Service-to-service via HTTP :

```
// Dans service-orders
const axios = require('axios');

async function createOrder(userId, productId) {
  // Appeler le service users pour vérifier l'utilisateur
  const user = await axios.get(`http://service-users:3000/users/${userId}`);

  // Appeler le service products pour vérifier le produit
  const product = await axios.get(`http://service-products:3001/products/${productId}`);

  // Créer la commande
  const order = await database.createOrder({ userId, productId });

  return order;
}
```

Les services communiquent par leur nom Docker (DNS automatique sur le network).

Service discovery

Dans Docker Compose, le service discovery est automatique via DNS. Chaque service est accessible par son nom.

```yaml
services:
  api:
    # Accessible via http://api:3000

  database:
    # Accessible via database:5432

  redis:
    # Accessible via redis:6379
```

Load balancing entre plusieurs instances

```yaml
version: '3.8'

services:
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf

  api:
    build: ./api
    deploy:
      replicas: 3  # 3 instances de l'API
# nginx.conf
upstream api {
    server api:3000;  # Docker fait automatiquement le load balancing
}
```

Docker Compose en mode swarm peut gérer le load balancing automatiquement.

Patterns de résilience

1. Retry avec backoff

Si un service est temporairement indisponible, réessayer plusieurs fois avec un délai croissant.

```javascript
async function callServiceWithRetry(url, maxRetries = 3) {
  for (let i = 0; i < maxRetries; i++) {
    try {
      return await axios.get(url);
    } catch (error) {
      if (i === maxRetries - 1) throw error;
      await sleep(1000 * (i + 1)); // 1s, 2s, 3s
    }
  }
}
```

2. Circuit breaker

Si un service échoue trop souvent, arrêter de l'appeler temporairement pour éviter la surcharge.

```
class CircuitBreaker {
  constructor(threshold = 5, timeout = 60000) {
    this.failures = 0;
    this.threshold = threshold;
    this.timeout = timeout;
    this.state = 'CLOSED'; // CLOSED, OPEN, HALF_OPEN
  }

  async call(fn) {
    if (this.state === 'OPEN') {
      throw new Error('Circuit breaker is OPEN');
    }

    try {
      const result = await fn();
      this.onSuccess();
      return result;
    } catch (error) {
      this.onFailure();
      throw error;
    }
  }

  onSuccess() {
    this.failures = 0;
    this.state = 'CLOSED';
  }

  onFailure() {
    this.failures++;
    if (this.failures >= this.threshold) {
      this.state = 'OPEN';
      setTimeout(() => {
        this.state = 'HALF_OPEN';
        this.failures = 0;
      }, this.timeout);
    }
  }
}
```

3. Fallback

Si un service est indisponible, utiliser une valeur par défaut ou un cache.

```
async function getUserProfile(userId) {
  try {
    return await axios.get(`http://service-users:3000/users/${userId}`);
  } catch (error) {
    // Fallback : retourner un profil par défaut
    return {
      id: userId,
      name: 'Unknown User',
      status: 'unavailable'
    };
  }
}
```

Monitoring et observabilité

Logs centralisés :

```
services:
  api:
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
```

Tous les logs au même endroit :

```
docker compose logs -f
```

Traces distribuées : Ajouter un `request-id` qui suit la requête à travers tous les services.

```
// Middleware Express
app.use((req, res, next) => {
  req.requestId = req.headers['x-request-id'] || uuid();
  res.setHeader('x-request-id', req.requestId);
  next();
});

// Logger avec request ID
console.log(`[${req.requestId}] User ${userId} fetched`);

// Transmettre aux autres services
axios.get('http://other-service', {
  headers: { 'x-request-id': req.requestId }
});
```

Maintenant vous pouvez tracer une requête à travers tous les services.

Exemple complet d'architecture

```yaml
version: '3.8'

services:
  # Reverse proxy
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    networks:
      - public
    depends_on:
      - api-gateway

  # API Gateway
  api-gateway:
    build: ./api-gateway
    expose:
      - "3000"
    networks:
      - public
      - backend
    environment:
      USERS_SERVICE_URL: http://service-users:3001
      PRODUCTS_SERVICE_URL: http://service-products:3002
      ORDERS_SERVICE_URL: http://service-orders:3003
    depends_on:
      - service-users
      - service-products
      - service-orders

  # Microservices
  service-users:
    build: ./services/users
    expose:
      - "3001"
    networks:
      - backend
    environment:
      DB_HOST: db-users
    depends_on:
      db-users:
        condition: service_healthy

  service-products:
    build: ./services/products
    expose:
      - "3002"
    networks:
      - backend
```

```yaml
    environment:
      DB_HOST: db-products
    depends_on:
      db-products:
        condition: service_healthy

  service-orders:
    build: ./services/orders
    expose:
      - "3003"
    networks:
      - backend
    environment:
      DB_HOST: db-orders
      REDIS_HOST: redis
    depends_on:
      db-orders:
        condition: service_healthy
      redis:
        condition: service_started

  # Databases
  db-users:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: users_db
      POSTGRES_PASSWORD: secret
    volumes:
      - db-users-data:/var/lib/postgresql/data
    networks:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "pg_isready"]
      interval: 5s
      timeout: 5s
      retries: 5

  db-products:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: products_db
      POSTGRES_PASSWORD: secret
    volumes:
      - db-products-data:/var/lib/postgresql/data
    networks:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "pg_isready"]
      interval: 5s
      timeout: 5s
      retries: 5

  db-orders:
```

```
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: orders_db
      POSTGRES_PASSWORD: secret
    volumes:
      - db-orders-data:/var/lib/postgresql/data
    networks:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "pg_isready"]
      interval: 5s
      timeout: 5s
      retries: 5

  # Cache
  redis:
    image: redis:7-alpine
    command: redis-server --appendonly yes
    volumes:
      - redis-data:/data
    networks:
      - backend

networks:
  public:
  backend:

volumes:
  db-users-data:
  db-products-data:
  db-orders-data:
  redis-data:
```

Cette architecture complète montre :

- Isolation réseau (public vs backend)
- Bases de données dédiées par service
- Cache partagé (Redis)
- Healthchecks pour les dépendances
- Persistance des données
- Reverse proxy pour l'entrée unique

Mise en pratique - Après-midi

Voir dans le document à cet effet !