Cours DevOps - Partie 3 : Docker Avancé et Production-Ready

Vue d'ensemble - Points abordés dans cette Partie 3

Aperçu rapide de tout ce qui est vu aujourd'hui

Théories

- **Docker Secrets**
  - Problématique de la sécurité des credentials
  - Différence entre variables d'environnement et secrets
  - Montage des secrets dans /run/secrets/
  - Fichiers secrets et .gitignore
- **Profiles Docker Compose**
  - Séparation environnements dev/staging/prod
  - Fichiers docker-compose avec overrides
  - Activation sélective de services par profil
  - Combinaison de fichiers compose
- **Resource Limits**
  - Impact de la consommation mémoire/CPU incontrôlée
  - Configuration des limits et reservations
  - Monitoring avec docker stats
  - Bonnes pratiques par type de service
- **Restart Policies**
  - Différence entre no/always/on-failure/unless-stopped
  - Stratégies de redémarrage en production
  - Éviter les boucles infinies avec on-failure:N
- **Networks isolés**
  - Segmentation réseau (public/backend/database)
  - Networks internes vs exposés
  - Principe du moindre privilège
  - Communication inter-services contrôlée
- **Multi-stage Builds**
  - Problème de la taille des images de build
  - Séparation builder/production
  - Optimisation des layers
  - Images Alpine vs standard

- **Healthchecks**
  - Différence entre started et healthy
  - Configuration interval/timeout/retries/start_period
  - Endpoints /health dans les APIs
  - Conditions depends_on avec service_healthy
- **Logging Drivers**
  - Rotation des logs (json-file)
  - Configuration max-size et max-file
  - Prévention du remplissage disque
  - Différence dev vs prod

Mises en pratique

- **Projet du jour : Stack Multi-Services Production-Ready**
  - Formation des groupes et setup repository
  - PostgreSQL avec Docker secrets et volumes persistants
  - Rails API avec models User/Post, CORS, healthcheck
  - Next.js frontend avec App Router et gestion d'erreurs
  - Nginx reverse proxy avec upstreams
  - Application des concepts avancés (profiles, limits, networks)
  - Scripts d'automation (setup, dev, prod, Makefile)
  - Validation complète avec tests de persistance
- **Concepts Docker appliqués**
  - Création et utilisation de secrets Docker
  - Configuration de profiles pour dev et prod
  - Définition de resource limits sur tous les services
  - Configuration de restart policies appropriées
  - Isolation réseau avec 3 networks distincts
  - Multi-stage builds pour Rails et Next.js
  - Healthchecks sur tous les services critiques
  - Logging avec rotation en production
- **Travail collaboratif**
  - Repository GitHub par groupe
  - Commits réguliers de chaque membre
  - README avec documentation claire mais concise
  - Tests de validation entre groupes

---

1 - Docker Secrets : Sécuriser les Credentials

CORE NOTIONS

Pourquoi Docker Secrets ?

Jusqu'à maintenant, vous avez peut-être mis des mots de passe dans vos fichiers `.env` ou directement dans `docker-compose.yml`.

Problème :

```
# docker-compose.yml (commité sur GitHub)
services:
  database:
    environment:
      POSTGRES_PASSWORD: mon_super_mot_de_passe  # Visible par tous !
```

Si ce fichier est commité, le mot de passe est exposé publiquement.

Solution : Docker Secrets

Docker Secrets permet de stocker les données sensibles en dehors des fichiers de configuration, et de les monter uniquement dans les conteneurs qui en ont besoin.

Comment ça marche ?

Étape 1 : Créer un fichier secret

```
# Créer le dossier
mkdir secrets

# Créer le secret
echo "mon_mot_de_passe_securise" > secrets/db_password.txt

# Protéger le fichier
chmod 600 secrets/db_password.txt
```

Étape 2 : Ajouter secrets/ au .gitignore

```
# .gitignore
secrets/
```

Le mot de passe n'est jamais commité.

Étape 3 : Utiliser le secret dans docker-compose.yml

```yaml
version: '3.8'

services:
  database:
    image: postgres:15-alpine
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password

secrets:
  db_password:
    file: ./secrets/db_password.txt
```

Étape 4 : Lire le secret dans le conteneur

Le secret est monté dans `/run/secrets/` (en mémoire, pas sur disque).

```bash
# Dans le conteneur
cat /run/secrets/db_password
```

Points clés

- Les secrets sont montés en lecture seule
- Ils sont stockés en mémoire (tmpfs), jamais sur disque
- Seuls les services qui déclarent le secret peuvent y accéder
- Utiliser le suffix `_FILE` pour les variables d'environnement

Lire un secret dans l'application

Node.js :

```javascript
const fs = require('fs');

const dbPassword = fs.readFileSync(
  process.env.DB_PASSWORD_FILE || '/run/secrets/db_password',
  'utf8'
).trim();
```

Rails database.yml :

```yaml
default: &default
  adapter: postgresql
  password: <%= File.read(ENV.fetch("POSTGRES_PASSWORD_FILE",
"/run/secrets/db_password")).strip rescue "" %>
```

---

2 - Profiles Docker Compose : Environnements Multiples

CORE NOTIONS

Le problème

En développement, vous voulez :

- Hot reload pour le code
- Adminer pour débugger la base de données
- Logs détaillés
- Pas de limites de ressources strictes

En production, vous voulez :

- Images optimisées et immuables
- Pas d'outils de debug exposés
- Logging avec rotation
- Limits de ressources strictes

Comment gérer ces deux environnements sans dupliquer tout le code ?

Solution : Compose Profiles

Docker Compose permet de définir des profils qui activent ou désactivent certains services.

Structure des fichiers

```
projet/
├── docker-compose.yml          # Base commune (tous environnements)
├── docker-compose.dev.yml      # Overrides pour développement
├── docker-compose.prod.yml     # Overrides pour production
├── .env.example
├── .env.dev
└── .env.prod
```

Fichier de base : docker-compose.yml

```yaml
version: '3.8'

services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    environment:
      RAILS_ENV: ${RAILS_ENV:-development}
    volumes:
      - rails_bundle:/usr/local/bundle
    networks:
      - backend
```

```
      - database

  database:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - database

volumes:
  postgres_data:
  rails_bundle:

networks:
  backend:
  database:
    internal: true

secrets:
  db_password:
    file: ./secrets/db_password.txt
```

Override dev : `docker-compose.dev.yml`

```
version: '3.8'

services:
  backend:
    build:
      target: development
    volumes:
      - ./backend:/app  # Hot reload
    profiles:
      - dev

  adminer:
    image: adminer:4-standalone
    ports:
      - "8080:8080"
    networks:
      - database
    profiles:
      - dev
```

Override prod : `docker-compose.prod.yml`

```yaml
version: '3.8'

services:
  backend:
    build:
      target: production
    restart: on-failure:3
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
    profiles:
      - prod
```

Lancement selon l'environnement

```bash
# Développement
docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up

# Production
docker compose -f docker-compose.yml -f docker-compose.prod.yml --profile prod up -d
```

Les deux commandes Docker Compose listées servent à lancer des environnements "développement" et "production" à l'aide de fichiers de configuration et de profils ciblés.

Via ces commandes on va donc lancer le docker compose de base + son override.

Cette commande utilise deux fichiers de configuration (`docker-compose.yml` et `docker-compose.dev.yml`) et lance les services sous le profil "dev". Cela permet d'avoir des paramètres spécifiques au développement (modules supplémentaires, montage de volume pour le code, debug, etc.). Idem pour prod !

Le `-f` signifie `--file` -> cela permet de spécifier les "Compose configuration files".

Fichiers .env par environnement

.env.dev :

```
POSTGRES_DB=rails_app_development
POSTGRES_USER=postgres
RAILS_ENV=development
NODE_ENV=development
```

.env.prod :

```
POSTGRES_DB=rails_app_production
POSTGRES_USER=postgres
RAILS_ENV=production
NODE_ENV=production
RAILS_LOG_TO_STDOUT=true
```

---

3 - Resource Limits : Contrôler la Consommation

CORE NOTIONS

Pourquoi limiter les ressources ?

Sans limites, un seul service peut consommer toute la RAM ou le CPU de la machine hôte, ce qui peut :

- Crasher l'hôte complet
- Rendre les autres services inutilisables
- Causer des problèmes de performance imprévisibles

Configuration des limits

```yaml
services:
  backend:
    deploy:
      resources:
        limits:
          cpus: '0.5'       # Maximum 50% d'un CPU
          memory: 512M      # Maximum 512 MB
        reservations:       # Minimum garanti
          cpus: '0.25'
          memory: 256M
```

Différence limits vs reservations

limits : Plafond absolu. Le conteneur ne pourra jamais dépasser.

reservations : Minimum garanti pour le scheduling. Docker s'assure que cette ressource est disponible avant de démarrer le conteneur.

Recommandations par type de service

Service	CPU Limit	Memory Limit	Justification
PostgreSQL	0.5	512M	Base de données (prioritaire)
Rails API	0.5	512M	Backend applicatif
Next.js	0.5	512M	SSR consomme de la mémoire
Nginx	0.25	256M	Proxy léger
Redis	0.25	256M	Cache en mémoire

Service	CPU Limit	Memory Limit	Justification
Adminer	0.25	256M	Dev only, non critique

Surveillance des ressources

```
# Voir l'utilisation en temps réel
docker stats

# Sortie :
# CONTAINER       CPU %     MEM USAGE / LIMIT     MEM %
# backend         2.50%     245MiB / 512MiB       47.85%
# database        1.20%     156MiB / 512MiB       30.47%
# nginx           0.10%     12MiB / 256MiB        4.69%
```

Les limits sont bien appliquées.

Bonnes pratiques

- Toujours définir des limits en production
- Commencer conservateur, ajuster selon les besoins réels
- Monitorer avec `docker stats` après déploiement
- Prévoir de la marge (ne pas mettre limits = usage moyen)

---

4 - Restart Policies : Résilience Automatique

CORE NOTIONS

Les différentes policies

Policy	Comportement	Usage
`no`	Ne jamais redémarrer	Développement, debug
`always`	Toujours redémarrer (même après arrêt)	Rarement recommandé
`on-failure`	Redémarrer si exit code != 0	Production (services critiques)
`on-failure:3`	Redémarrer max 3 fois	Production (évite boucles)
`unless-stopped`	Redémarrer sauf si arrêt manuel	Services long-running

Configuration dans docker-compose

```
services:
  backend:
    restart: on-failure:3  # Max 3 tentatives

  nginx:
    restart: unless-stopped

  database:
    restart: unless-stopped
```

Quand utiliser quelle policy ?

Développement :

```
services:
  backend:
    # Pas de restart policy
    # On veut voir les erreurs immédiatement
```

Production :

```
services:
  backend:
    restart: on-failure:3
    # Si le service crash, réessaie 3 fois
    # Évite les redémarrages infinis si le problème est permanent

  nginx:
    restart: unless-stopped
    # Nginx doit toujours tourner sauf arrêt manuel

  database:
    restart: unless-stopped
    # La base de données doit toujours être disponible
```

Combinaison avec healthchecks

Les restart policies redémarrent le conteneur s'il s'arrête (exit).

Les healthchecks marquent le conteneur comme unhealthy s'il ne répond pas, mais ne le redémarrent pas automatiquement.

Pour un redémarrage automatique en cas de service unhealthy, il faut un orchestrateur comme Kubernetes.

---

5 - Networks Isolés : Segmentation Réseau

CORE NOTIONS

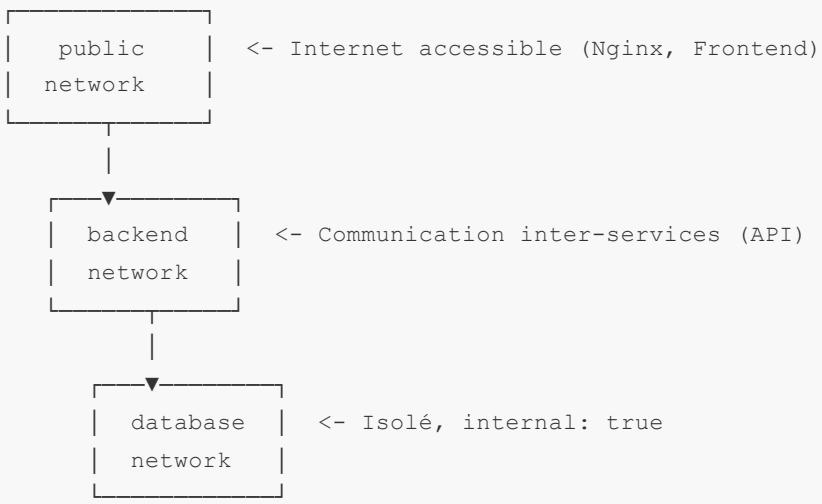
Principe du moindre privilège réseau

Tous les services ne doivent pas pouvoir communiquer entre eux.

Exemple :

- Le frontend ne doit PAS pouvoir accéder directement à la base de données
- La base de données ne doit PAS être accessible depuis internet
- Seul le backend peut parler à la base de données

Architecture à 3 networks



Configuration

```yaml
networks:
  public:
    driver: bridge
    internal: false     # Accès internet

  backend:
    driver: bridge
    internal: false     # Communication services

  database:
    driver: bridge
    internal: true      # ISOLÉ, pas d'internet

services:
  nginx:
    networks:
      - public
      - backend
```

```
  frontend:
    networks:
      - public

  backend:
    networks:
      - backend
      - database

  database:
    networks:
      - database       # UNIQUEMENT accessible par backend
```

Règles d'isolation

- `nginx` et `backend` sont sur `public` : nginx peut atteindre le backend
- `backend` et `database` sont sur `database` : le backend peut atteindre la DB
- `frontend` et `database` ne partagent AUCUN network : impossible de communiquer
- Network `database` avec `internal: true` : pas d'accès internet depuis la DB

Vérification de l'isolation

```
# Depuis frontend, essayer d'atteindre database
docker compose exec frontend ping database
# Échec attendu : "bad address"

# Depuis backend, atteindre database
docker compose exec backend ping database
# Succès
```

---

6 - Multi-stage Builds : Optimisation d'Images

CORE NOTIONS

Le problème

Quand vous buildez une application, vous avez besoin d'outils de compilation (gcc, npm, bundler, etc.).

En production, vous avez seulement besoin du résultat compilé.

Image sans multi-stage :

```
FROM node:18
WORKDIR /app
COPY . .
RUN npm install  # Installe TOUT, y compris devDependencies
RUN npm run build
CMD ["node", "dist/server.js"]
```

Résultat : image de 1.2 GB avec tous les outils de build.

Image avec multi-stage :

```
# Stage 1 : Builder (tous les outils)
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2 : Production (seulement le nécessaire)
FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
CMD ["node", "dist/server.js"]
```

Résultat : image de 150 MB.

Exemple Rails avec multi-stage

```
# Stage builder
FROM ruby:3.2 AS builder
WORKDIR /app
RUN apk add --no-cache build-base postgresql-dev
COPY Gemfile Gemfile.lock ./
RUN bundle install --jobs 4 --retry 3

# Stage production
FROM ruby:3.2-alpine
WORKDIR /app
RUN apk add --no-cache postgresql-client
COPY --from=builder /usr/local/bundle /usr/local/bundle
COPY . .
CMD ["bundle", "exec", "rails", "server", "-b", "0.0.0.0"]
```

Avantages

- Taille réduite de 50 à 80%
- Moins de packages = moins de vulnérabilités
- Temps de pull/push plus rapide
- Meilleure sécurité (pas d'outils de compilation en prod)

Build avec target spécifique

```
# Builder seulement (pour debug)
docker build --target builder -t mon-app:builder .

# Production (défaut, dernier stage)
docker build -t mon-app:latest .
```

Utilisation avec docker-compose

```
services:
  backend:
    build:
      context: ./backend
      target: production  # Choisir le stage
```

---

7 - Healthchecks : Service Ready vs Started

CORE NOTIONS

Le problème

Sans healthcheck, Docker considère qu'un conteneur est "ready" dès qu'il démarre.

Mais :

- PostgreSQL prend 5-10 secondes pour être vraiment prêt
- Une API peut démarrer mais ne pas pouvoir se connecter à la base
- Un service peut être bloqué dans une boucle infinie

Résultat : Votre API démarre, essaie de se connecter à la DB qui n'est pas prête, et crash.

Solution : Healthchecks

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

États possibles :

- `starting` : En phase de démarrage (start_period)
- `healthy` : Le test réussit
- `unhealthy` : Le test échoue après N retries

Configuration d'un healthcheck

```
services:
  database:
    image: postgres:16-alpine
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s      # Tester toutes les 10 secondes
      timeout: 5s        # Le test doit répondre en 5s max
      retries: 5         # 5 échecs consécutifs = unhealthy
      start_period: 10s  # Attendre 10s avant de commencer les tests
```

Healthcheck pour une API

Créer un endpoint /health :

```
// Node.js/Express
app.get('/health', async (req, res) => {
  try {
    // Vérifier la connexion DB
    await database.query('SELECT 1');

    res.status(200).json({ status: 'healthy' });
  } catch (error) {
    res.status(503).json({ status: 'unhealthy', error: error.message });
  }
});
```

Healthcheck dans docker-compose :

```
services:
  backend:
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:3000/health || exit 1"]
      interval: 15s
      timeout: 5s
      retries: 3
      start_period: 40s  # Rails prend du temps à démarrer
```

Dépendances avec conditions

Sans healthcheck :

```
services:
  backend:
    depends_on:
      - database  # Backend démarre dès que database est STARTED
```

Problème : database peut ne pas être ready.

Avec healthcheck :

```yaml
services:
  backend:
    depends_on:
      database:
        condition: service_healthy  # Attend que database soit HEALTHY

  database:
    healthcheck:
      test: ["CMD-SHELL", "pg_isready"]
      interval: 5s
      timeout: 5s
      retries: 5
```

Maintenant backend démarre seulement quand PostgreSQL répond vraiment.

Voir le statut des healthchecks

```bash
# Dans docker ps
docker compose ps

# Colonne STATUS :
# Up 2 minutes (healthy)
# Up 2 minutes (health: starting)
# Up 2 minutes (unhealthy)
```

---

8 - Logging Drivers : Rotation et Gestion

CORE NOTIONS

Le problème

Par défaut, Docker stocke tous les logs de tous les conteneurs sans limite.

Sur un serveur de production qui tourne pendant des mois, les logs peuvent remplir tout le disque.

Solution : Logging avec rotation

Le driver `json-file` avec rotation limite automatiquement la taille des logs.

Configuration

```yaml
services:
  backend:
    logging:
      driver: "json-file"
      options:
        max-size: "10m"    # Maximum 10 MB par fichier
        max-file: "3"      # Garder 3 fichiers (30 MB total)
```

Quand un fichier atteint 10 MB, Docker le renomme et en crée un nouveau. Après 3 fichiers, le plus ancien est supprimé.

Différence dev vs prod

Développement :

```
services:
  backend:
    # Pas de configuration logging
    # Logs illimités pour debug
```

Production :

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

Drivers disponibles

Driver	Usage	Production
`json-file`	Fichiers locaux (défaut)	Oui
`syslog`	Serveur syslog centralisé	Oui
`journald`	systemd journal	Oui
`none`	Pas de logs	Non

Lire les logs avec rotation

```
# Logs récents
docker compose logs -f backend

# Docker gère automatiquement la rotation
# Vous voyez toujours les derniers logs disponibles
```

---

9 - Mise en Pratique : Projet Stack Multi-Services

Objectif du jour

Construire une stack Docker complète et production-ready en appliquant tous les concepts vus aujourd'hui et ces derniers jours.

Architecture cible :

```
Internet → Nginx (80) → Next.js (3000)
                      → Rails API (3000) → PostgreSQL (5432)
                                         → Adminer (8080, dev only)
```

Technologies :

- PostgreSQL 16 Alpine
- Rails 7.1 ou 8 API (Ruby 3.2)
- Next.js 14 ou 15 (Node 20)
- Nginx 1.25 Alpine

Concepts appliqués :

- Docker Secrets pour les mots de passe
- Profiles dev/prod avec overrides
- Resource limits sur tous les services
- Restart policies appropriées
- 3 networks isolés (public/backend/database)
- Multi-stage builds pour Rails et Next
- Healthchecks sur tous les services
- Logging avec rotation en prod

Formation des groupes

Comme pour le jour 2, formez des groupes de 3 à 4 personnes.

Vous pouvez garder les mêmes groupes si vous le souhaitez.

Important :

- Un repository GitHub par groupe
- Tous les membres doivent faire des commits
- README.md avec les noms de tous les membres + description concise du projet

Structure du projet

```
projet-docker-j3/
├── backend/
│   ├── Dockerfile
│   ├── Dockerfile.dev
│   ├── entrypoint.sh
│   ├── Gemfile
│   ├── Gemfile.lock
│   ├── config/
│   └── app/
├── frontend/
│   ├── Dockerfile
│   ├── Dockerfile.dev
```

```
│   ├── package.json
│   ├── next.config.js
│   └── app/
├── nginx/
│   ├── Dockerfile
│   └── nginx.conf
├── database/
│   ├── Dockerfile
│   └── init.sql
├── scripts/
│   ├── setup.sh
│   ├── dev.sh
│   └── prod.sh
├── secrets/
│   └── .gitkeep
├── docker-compose.yml
├── docker-compose.dev.yml
├── docker-compose.prod.yml
├── .env.example
├── .gitignore
├── Makefile
└── README.md
```

Phase 1 : Setup Initial

Objectif

Créer la structure de base du projet et le repository.

Tâches

1. Créer le repository GitHub

Un membre du groupe crée le repository et ajoute les autres comme collaborateurs.

2. Cloner localement

Tous les membres clonent le repository.

3. Créer la structure de dossiers

```
mkdir -p backend frontend nginx database scripts secrets courses
```

4. Créer le .gitignore

```
.env
.env.dev
.env.prod
secrets/*
!secrets/.gitkeep
node_modules/
frontend/node_modules/
frontend/.next/
backend/log/
backend/tmp/
```

5. Créer le README.md

```
# Projet Docker J3 - Stack Production-Ready

## Membres du groupe

- Prénom Nom
- Prénom Nom
- Prénom Nom

## Architecture

Rails API + Next.js + PostgreSQL + Nginx

## Lancement

Voir les instructions dans ce README une fois le projet terminé.
```

6. Premier commit

```
git add .
git commit -m "Initial setup: project structure"
git push
```

Validation

- Repository créé et accessible par tous
- Structure de dossiers présente
- Premier commit réalisé

Phase 2 : Database Layer

Objectif

Créer le service PostgreSQL avec :

- Secret Docker pour le mot de passe
- Volume persistant
- Healthcheck fonctionnel
- Script d'initialisation avec seed data

Tâches

1. Créer le Dockerfile PostgreSQL

Dans `database/Dockerfile`, créer une image basée sur `postgres:16-alpine` qui :

- Installe postgresql-contrib si nécessaire
- Copie le script init.sql
- Définit un healthcheck avec `pg_isready`

2. Créer le script init.sql

Dans `database/init.sql`, créer :

- Une table `users` (id, name, email, created_at, updated_at)
- Une table `posts` (id, title, content, user_id, created_at, updated_at)
- 3 users de test
- 10 posts de test

3. Créer le secret

```
echo "votre_mot_de_passe" > secrets/db_password.txt
chmod 600 secrets/db_password.txt
```

4. Créer docker-compose.yml (base)

Définir :

- Service `db` avec build depuis `./database`
- Variables d'environnement (POSTGRES_DB, POSTGRES_USER, POSTGRES_PASSWORD_FILE)
- Secret `db_password` pointant vers le fichier
- Volume nommé `postgres_data`
- Network `database` avec `internal: true`
- Healthcheck
- Resource limits (0.5 CPU, 512M RAM)

5. Créer docker-compose.dev.yml

Ajouter :

- Service `adminer` avec profile `dev`
- Port 8080 exposé
- Connecté au network `database`

6. Tester

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up db adminer
```

Ouvrir http://localhost:8080 et se connecter :

- System: PostgreSQL
- Server: db
- Username: postgres
- Password: (celui dans secrets/db_password.txt)
- Database: rails_app_development

Validation

- Connexion réussie via Adminer
- Tables users et posts visibles
- 3 users et 10 posts présents
- Healthcheck "healthy" dans `docker compose ps`

Questions checkpoint

- Pourquoi utiliser `internal: true` pour le network database ?
- Que se passe-t-il si on supprime le volume `postgres_data` ?
- Où est stocké le secret dans le conteneur ?

Phase 3 : Backend Rails API

Objectif

Créer une API Rails minimale avec :

- Multi-stage Dockerfile (builder + production + development)
- Models User et Post
- Controllers avec endpoints REST
- Endpoint /health pour healthcheck
- CORS configuré
- Entrypoint avec wait-for-database

Tâches

1. Créer le Gemfile

Définir les gems nécessaires :

- rails (~> 7.1.0)
- pg (~> 1.5)
- puma (~> 6.0)
- rack-cors

2. Créer le Dockerfile multi-stage

Trois stages :

- `builder` : FROM ruby:3.2-alpine, installe build-base et postgresql-dev, bundle install
- `production` : FROM ruby:3.2-alpine, copie les gems depuis builder, COPY le code
- `development` : FROM ruby:3.2-alpine, bundle install complet, volumes pour hot reload

3. Créer l'entrypoint.sh

Script qui :

- Attend que PostgreSQL soit ready (boucle avec pg_isready)
- Lit le secret depuis POSTGRES_PASSWORD_FILE
- Supprime server.pid si existant
- Exécute la commande fournie

4. Créer la structure Rails minimale

Dans `backend/config/` :

- `application.rb` : config Rails en mode API, config CORS
- `database.yml` : connexion PostgreSQL avec lecture du secret
- `routes.rb` : routes pour /health et /api/users, /api/posts
- `environment.rb` et `boot.rb`

5. Créer les models

- `app/models/application_record.rb`
- `app/models/user.rb` : has_many :posts
- `app/models/post.rb` : belongs_to :user

6. Créer les controllers

- `app/controllers/application_controller.rb`
- `app/controllers/health_controller.rb` : GET /health retourne { status: 'ok' }
- `app/controllers/api/users_controller.rb` : index et show
- `app/controllers/api/posts_controller.rb` : index et show avec include user

7. Ajouter au docker-compose.yml

Service `backend` :

- Build depuis ./backend, target production
- Depends_on database avec condition service_healthy
- Networks: backend, database
- Secrets: db_password
- Healthcheck sur /health
- Resource limits (0.5 CPU, 512M RAM)
- Port 3000 exposé (pour test direct)

8. Ajouter au docker-compose.dev.yml

Override `backend` :

- Target: development
- Volume mount ./backend:/app pour hot reload
- Profile: dev

9. Tester

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up

# Dans un autre terminal
curl http://localhost:3000/health
curl http://localhost:3000/api/posts
curl http://localhost:3000/api/users
```

Validation

- `/health` retourne `{"status":"ok"}`
- `/api/posts` retourne les 10 posts avec leurs users
- `/api/users` retourne les 3 users
- Healthcheck "healthy"
- Hot reload fonctionne (modifier un fichier, voir le changement)

Questions checkpoint

- Pourquoi utiliser `pg_isready` dans l'entrypoint ?
- Quelle est la différence entre les targets production et development ?
- Comment Rails lit-il le mot de passe de la base de données ?

Phase 4 : Frontend Next.js

Objectif

Créer un frontend Next.js avec :

- Multi-stage Dockerfile optimisé
- 2 pages : liste des posts et détail d'un post
- Appels API vers le backend Rails
- Gestion des erreurs
- Healthcheck

Tâches

1. Créer le package.json

Dépendances :

- next (14.0.4)
- react (^18.2.0)
- react-dom (^18.2.0)
- typescript, @types/node, @types/react, @types/react-dom

Scripts : dev, build, start, lint

2. Créer le Dockerfile multi-stage

Quatre stages :

- `deps` : Copie package files, npm ci
- `builder` : Copie depuis deps, copie code, npm run build
- `production` : Image alpine, copie standalone depuis builder
- `development` : Image standard, npm install, hot reload

3. Créer next.config.js

Config :

- output: 'standalone'
- rewrites pour proxy /api vers backend (optionnel, peut passer par nginx)

4. Créer tsconfig.json

Configuration TypeScript standard pour Next.js.

5. Créer app/layout.tsx

Layout de base avec :

- Metadata (title, description)
- HTML structure

- Header simple

6. Créer app/page.tsx

Page d'accueil qui :

- Fetch GET /api/posts depuis le backend (via variable d'environnement)
- Affiche la liste des posts
- Liens vers /posts/[id]
- Gestion d'erreur si API down

7. Créer app/posts/[id]/page.tsx

Page détail qui :

- Fetch GET /api/posts/:id
- Affiche le post complet avec user
- Lien retour vers home
- Gestion erreur 404

8. Ajouter au docker-compose.yml

Service `frontend` :

- Build depuis ./frontend, target production
- Depends_on backend avec condition service_healthy
- Networks: public
- Environment: NEXT_PUBLIC_API_URL
- Healthcheck sur /
- Resource limits (0.5 CPU, 512M RAM)
- Port 3001 exposé (pour test direct)

9. Ajouter network public

```
networks:
  public:
    driver: bridge
```

10. Ajouter au docker-compose.dev.yml

Override `frontend` :

- Target: development
- Volume mount pour hot reload
- Profile: dev

11. Tester

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up

# Ouvrir navigateur
http://localhost:3001
```

Validation

- Page d'accueil affiche les 10 posts
- Click sur un post ouvre la page détail
- Retour à la liste fonctionne
- Si backend est down, message d'erreur approprié
- Healthcheck "healthy"

Questions checkpoint

- Pourquoi séparer les stages deps et builder ?
- Que fait `output: 'standalone'` ?
- Comment Next.js connaît-il l'URL de l'API backend ?

Phase 5 : Nginx Reverse Proxy

Objectif

Configurer Nginx comme point d'entrée unique sur le port 80 :

- Route /api vers Rails backend
- Route / vers Next.js frontend
- Healthcheck sur /health
- Logs centralisés

Tâches

1. Créer nginx/Dockerfile

Image basée sur `nginx:1.25-alpine` qui :

- Copie nginx.conf
- Définit healthcheck sur /health
- Expose port 80

2. Créer nginx/nginx.conf

Configuration avec :

- Upstream `rails_backend` pointant vers backend:3000
- Upstream `nextjs_frontend` pointant vers frontend:3000
- Server bloc listen 80

- Location /api/ → proxy_pass vers rails_backend
- Location / → proxy_pass vers nextjs_frontend
- Location /health → return 200 "healthy"
- Proxy headers (X-Real-IP, X-Forwarded-For, Host)
- GZIP compression
- Logs dans /var/log/nginx/

3. Ajouter au docker-compose.yml

Service `nginx` :

- Build depuis ./nginx
- Depends_on frontend et backend avec service_healthy
- Networks: public, backend
- Ports: "80:80"
- Volume pour logs: nginx_logs:/var/log/nginx
- Healthcheck sur /health
- Resource limits (0.25 CPU, 256M RAM)

4. Modifier les services backend et frontend

Retirer l'exposition des ports 3000 et 3001 (utiliser `expose` au lieu de `ports`).

Tous les accès passent maintenant par Nginx.

5. Tester

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up

# Tester
curl http://localhost/health
curl http://localhost/api/posts
# Ouvrir navigateur
http://localhost
```

Validation

- http://localhost affiche le frontend Next.js
- http://localhost/api/posts retourne JSON depuis Rails
- http://localhost/health retourne "healthy"
- Tous les healthchecks "healthy"
- Impossible d'accéder directement à backend:3000 ou frontend:3001 depuis l'hôte

Questions checkpoint

- Pourquoi utiliser `expose` au lieu de `ports` pour backend et frontend ?
- Que font les headers X-Real-IP et X-Forwarded-For ?
- Comment Nginx sait-il où router /api vs / ?

Phase 6 : Concepts Avancés J3

Objectif

Appliquer tous les concepts Docker avancés :

- Profiles dev/prod fonctionnels
- Resource limits sur tous les services
- Restart policies appropriées
- Logging avec rotation en prod
- Networks isolés
- Secrets sécurisés

Tâches

1. Vérifier les resource limits

Tous les services doivent avoir des limits définis dans docker-compose.yml.

2. Ajouter restart policies en prod

Dans docker-compose.prod.yml, ajouter `restart: on-failure:3` sur backend, nginx, database.

3. Configurer logging en prod

Dans docker-compose.prod.yml, ajouter logging avec rotation (max-size: 10m, max-file: 3) sur tous les services.

4. Créer .env.example

Fichier template avec :

- POSTGRES_DB=rails_app_development
- POSTGRES_USER=postgres
- Note: Password dans secrets/db_password.txt
- RAILS_ENV, NODE_ENV, etc.

5. Créer .env.dev et .env.prod

Valeurs spécifiques par environnement.

6. Vérifier l'isolation réseau

Networks :

- `public` : nginx, frontend

- `backend` : nginx, backend, frontend (pour appels API)
- `database` (internal: true) : backend, database, adminer

7. Tester les deux profils

```
# Dev
docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up

# Prod
docker compose -f docker-compose.yml -f docker-compose.prod.yml --profile prod up -d
```

Validation

- Profile dev : Adminer accessible, hot reload fonctionne
- Profile prod : Adminer absent, restart policies actives, logging avec rotation
- `docker stats` montre les limits appliquées
- Frontend ne peut pas ping database directement

Questions checkpoint

- Quelle est la différence entre un network normal et internal: true ?
- Pourquoi on-failure:3 au lieu de always ?
- Comment vérifier que le logging avec rotation fonctionne ?

Phase 7 : Scripts d'Automation

Objectif

Créer des scripts pour simplifier les commandes Docker.

Tâches

1. Créer scripts/setup.sh

Script qui :

- Copie .env.example vers .env si n'existe pas
- Génère secrets/db_password.txt si n'existe pas
- Crée les volumes Docker (postgres_data, rails_bundle, nginx_logs)
- Affiche message de succès

2. Créer scripts/dev.sh

Script qui :

- Charge .env.dev
- Lance `docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up`

3. Créer scripts/prod.sh

Script qui :

- Charge .env.prod
- Lance `docker compose -f docker-compose.yml -f docker-compose.prod.yml --profile prod up -d`
- Affiche les URLs d'accès

4. Créer Makefile

Targets :

- `help` : Affiche les commandes disponibles
- `setup` : Exécute scripts/setup.sh
- `dev` : Exécute scripts/dev.sh
- `prod` : Exécute scripts/prod.sh
- `stop` : docker compose down
- `clean` : docker compose down -v
- `logs` : docker compose logs -f
- `ps` : docker compose ps

5. Rendre les scripts exécutables

```
chmod +x scripts/*.sh
```

6. Tester

```
make setup
make dev
```

Validation

- `make help` affiche toutes les commandes
- `make setup` crée les secrets et volumes
- `make dev` démarre l'environnement de développement
- `make stop` arrête tous les conteneurs

Questions checkpoint

- Pourquoi créer un Makefile en plus des scripts ?
- Que fait `docker compose down -v` ?
- Comment ajouter une nouvelle commande au Makefile ?

Phase 8 : Validation Finale

Objectif

Vérifier que tout fonctionne correctement et que le projet respecte toutes les contraintes.

Checklist de validation

Setup et lancement :

```
# Tout nettoyer
make clean

# Setup initial
make setup

# Lancer en dev
make dev
```

Tests fonctionnels :

- [ ] Adminer accessible sur http://localhost:8080
- [ ] Frontend accessible sur http://localhost
- [ ] Liste des posts affichée
- [ ] Détail d'un post fonctionne
- [ ] API accessible sur http://localhost/api/posts
- [ ] Healthcheck Nginx : `curl http://localhost/health`

Tests de persistance :

```
# Créer des données (optionnel si déjà des posts)
# Arrêter tout
docker compose down

# Relancer
docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up -d

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

- [ ] Les posts existent toujours après redémarrage

Tests des healthchecks :

```
docker compose ps
```

- [ ] Tous les services affichent (healthy)

Tests des resource limits :

```
docker stats
```

- [ ] Limits visibles pour chaque service
- [ ] Aucun service ne dépasse ses limits

Tests de sécurité :

- [ ] `secrets/` dans .gitignore
- [ ] Secrets jamais committés
- [ ] Network database avec internal: true
- [ ] Frontend ne peut pas atteindre database directement

Tests des profiles :

```
# Arrêter dev
docker compose down

# Lancer prod
make prod

# Vérifier
docker compose ps
```

- [ ] Adminer absent en prod
- [ ] Restart policies actives
- [ ] Logging configuré

Documentation :

- [ ] README.md avec noms des membres
- [ ] Instructions de lancement claires
- [ ] Architecture documentée

Test entre groupes

Échangez avec un autre groupe :

- Clonez leur repository
- Suivez leur README
- Lancez leur projet
- Donnez du feedback

Questions finales

- Quel a été le concept le plus difficile aujourd'hui ?
- Quelle optimisation pourrait encore être faite ?
- Comment feriez-vous pour déployer ce projet sur un serveur ?

Conclusion

Vous avez maintenant une stack Docker complète et production-ready qui applique tous les concepts avancés vus aujourd'hui.

Ce que vous avez appris :

- Sécuriser les credentials avec Docker Secrets
- Gérer plusieurs environnements avec Compose Profiles
- Contrôler les ressources pour éviter la surcharge
- Configurer des restart policies pour la résilience
- Isoler les services avec des networks dédiés
- Optimiser les images avec multi-stage builds
- Garantir la disponibilité avec healthchecks
- Gérer les logs avec rotation

Prochaines étapes :

- Branch bonus : Traefik, Redis, Prometheus/Grafana
- CI/CD avec GitHub Actions
- Déploiement sur un serveur réel
- Kubernetes pour l'orchestration à grande échelle

Ressources :

- Docker Compose : https://docs.docker.com/compose/
- Docker Secrets : https://docs.docker.com/engine/swarm/secrets/
- Multi-stage Builds : https://docs.docker.com/build/building/multi-stage/
- Healthchecks : https://docs.docker.com/engine/reference/builder/#healthcheck