Mise en pratique - Après-midi

- Former groupes : obligatoire
- Repository obligatoire
  - Readme:
    Nom de chaque membre
  - Commits de chaque membre du groupe obligatoire

---

5 - Debug & Troubleshooting : Devenir un Pro du Diagnostic

CORE NOTIONS

Les commandes de debug essentielles

Pour votre information, afin que cela vous serve plus tard.

```
# Inspecter un conteneur en détail
docker inspect mon-conteneur

# Voir les logs en temps réel
docker logs -f mon-conteneur

# Voir les stats (CPU, RAM, réseau)
docker stats

# Lister les processus dans un conteneur
docker top mon-conteneur

# Rentrer dans un conteneur en cours d'exécution
docker exec -it mon-conteneur sh

# Voir l'historique des layers d'une image
docker history mon-image
```

Les erreurs classiques et leurs solutions

Erreur	Cause probable	Solution
`port already allocated`	Port déjà utilisé	Changer le port hôte
`no space left on device`	Disque plein	`docker system prune`
`connection refused`	Service pas démarré	Vérifier les logs
`unable to find image`	Typo ou image inexistante	Vérifier le nom

Exercice Pratique 3 : Débugger des Dockerfiles Cassés

Objectif : Je vous donne des Dockerfiles avec des erreurs, trouvez-les !

Dockerfile #1 : L'erreur subtile

```
FROM node:18-alpine
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD npm start
```

Questions :

1. Buildez cette image : `docker build -t debug-1 .`
2. Que se passe-t-il ?
3. Identifiez le problème (indice : regardez les logs de build)

Dockerfile #2 : L'ordre compte !

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["node", "server.js"]
```

Questions :

1. Ce Dockerfile fonctionne, mais quel est le problème de performance ?
2. Modifiez une ligne de code dans `src/`
3. Rebuilder : `docker build -t debug-2 .`
4. Que remarquez-vous ?

Dockerfile #3 : L'image géante

```
FROM node:18
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

Questions :

1. Buildez : `docker build -t debug-3 .`

2. Vérifiez la taille : `docker images debug-3`
3. Pourquoi l'image fait-elle plus de 1GB ?

CHECKPOINT 3 :

- Quelqu'un partage son écran et montre comment il a debuggé
- *"Quelle commande utilisez-vous pour voir pourquoi un conteneur crash au démarrage ?"*
- Discussion : autres erreurs rencontrées aujourd'hui ?

Exercice Pratique 4 : Débugger un docker-compose.yml

Fichier docker-compose-broken.yml :

```yaml
version: '3.8'

services:
  web:
    build: .
    ports:
      - "3000:3000"
    depends_on:
      - database
    environment:
      DB_HOST: postgres
      DB_PORT: 5432

  database:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data

volumes:
  db-data
```

Tâches :

1. Lancez : `docker compose -f docker-compose-broken.yml up`
2. L'application ne se connecte pas à la BDD. Pourquoi ?
3. Corrigez le fichier

CHECKPOINT 4 :

- *"Quelle commande pour voir les logs d'un seul service dans docker-compose ?"*
- Quelqu'un explique le concept de healthcheck

---

6 - Optimisation d'Images Docker

CORE NOTIONS

Pourquoi optimiser ?

Image	Taille	Impact
`node:18`	910 MB	Lent à pull, coûteux en stockage
`node:18-slim`	170 MB	Mieux, mais peut manquer des outils
`node:18-alpine`	110 MB	Petit, rapide, sécurisé

Les 5 techniques d'optimisation

1. Utiliser des images Alpine

```
FROM node:18-alpine  # Au lieu de node:18
```

2. Multi-stage builds

```
# Stage 1 : Build
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2 : Production (on ne garde que 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"]
```

3. .dockerignore (comme .gitignore)

```
node_modules
npm-debug.log
.git
.env
*.md
tests/
coverage/
```

4. Combiner les commandes RUN

```
# 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/*
```

5. Ordre des COPY

```
# Bon ordre (cache efficace)
COPY package*.json ./
RUN npm install
COPY . .

# Mauvais ordre (cache invalidé souvent)
COPY . .
RUN npm install
```

Exercice Pratique 5 : Challenge d'Optimisation

Objectif : Réduire au maximum la taille de votre Todo App

Dockerfile de départ (non optimisé)

```
FROM node:18
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["node", "src/server.js"]
```

Votre mission

1. **Buildez la version de départ**

   ```
   docker build -t todo-v1 .
   docker images todo-v1
   # Noter la taille
   ```

2. **Appliquez TOUTES les optimisations**
   - Image Alpine
   - Multi-stage build
   - .dockerignore
   - Ordre optimal des layers

- `npm ci --only=production`

3. **Comparez les résultats**

```
docker build -t todo-v2-optimized .
docker images | grep todo
```

4. **Calculez le gain**

```
# Exemple de résultat attendu :
# todo-v1              1.2GB
# todo-v2-optimized    150MB
# Gain : 87%
```

Compétition : Qui obtient l'image la plus petite (tout en fonctionnant) ?

CHECKPOINT 5 :

- Partage d'écran : les 3 meilleures optimisations
- *"Pourquoi `npm ci` est meilleur que `npm install` en production ?"*
- Discussion : compromis entre taille et fonctionnalités

---

7 - Mini-Projet Autonome : Stack Complète Persistante

MISSION FINALE

Créez une stack complète avec :

- API Node.js (votre Todo App ou une nouvelle)
- Base de données PostgreSQL **avec persistance**
- Redis pour le cache **avec persistance**
- Nginx comme reverse proxy
- Tout orchestré avec docker-compose
- Images optimisées

Architecture cible

```
Internet → Nginx (80) → API (3000) → Postgres (5432)
                              ↓
                         Redis (6379)
```

Contraintes

1. **Tout doit survivre à `docker compose down` puis `up`**
2. **Les images doivent être optimisées (<200MB pour l'API)**
3. **Les services doivent avoir des healthchecks**
4. **Les logs doivent être accessibles via volumes**

Structure attendue

```
mini-projet/
├── api/
│   ├── Dockerfile
│   ├── .dockerignore
│   ├── src/
│   └── package.json
├── nginx/
│   └── nginx.conf
├── docker-compose.yml
└── README.md
```

Starter : docker-compose.yml minimal

```yaml
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: ./api
    expose:
      - "3000"
    environment:
      POSTGRES_HOST: postgres
      REDIS_HOST: redis
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_started

  postgres:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
    volumes:
      # TODO: Ajouter volume persistant
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user"]
      interval: 5s
```

```
      timeout: 5s
      retries: 5

  redis:
    image: redis:7-alpine
    # TODO: Ajouter persistence et volume

volumes:
  # TODO: Déclarer les volumes
```

Nginx config starter

```
events {
    worker_connections 1024;
}

http {
    upstream api {
        server api:3000;
    }

    server {
        listen 80;

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

        location /health {
            access_log off;
            return 200 "OK";
        }
    }
}
```

Tests de validation

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

# 2. Tester l'API via Nginx
curl http://localhost/health

# 3. Créer des données
curl -X POST http://localhost/api/tasks \
  -H "Content-Type: application/json" \
  -d '{"title":"Test persistance"}'

# 4. Vérifier Redis
```

```
docker compose exec redis redis-cli KEYS '*'

# 5. Arrêter TOUT (y compris les conteneurs)
docker compose down

# 6. Redémarrer
docker compose up -d

# 7. Vérifier que les données existent toujours
curl http://localhost/api/tasks
```

Succès si :

- Nginx répond sur le port 80
- Les tâches survivent au redémarrage
- Redis a toujours ses clés
- Les images sont optimisées

Bonus challenges

Niveau 1 : Ajouter des logs dans un volume dédié
Niveau 2 : Ajouter un service Adminer (interface web pour Postgres)
Niveau 3 : Configurer Redis en mode AOF pour une meilleure persistance
Niveau 4 : Ajouter un conteneur de backup automatique de la BDD

CHECKPOINT FINAL (dernières 15 min) :

- 2-3 personnes partagent leur écran
- Démo live de leur stack
- Q&A : problèmes rencontrés et solutions
- Vote : meilleure optimisation / meilleure architecture
