



# DevOps & Docker - Jour 2 PM

Master 1 Développement Fullstack



# Chapitre 9

Multi-stage builds

# Le problème : Images trop lourdes

```
FROM node:20
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
CMD ["npm", "start"]
```

## Problèmes :

- Toutes les dépendances de build restent dans l'image finale
- `node_modules` complet (dev + prod)
- Outils de build (TypeScript, Webpack, etc.)
- Taille de l'image : 1.2 GB 😱

# Analyse d'une image classique

```
docker images
# REPOSITORY      TAG      SIZE
# myapp           latest   1.2GB
```

Ce qui prend de la place :

- Base image Node.js : 300 MB
- Dependencies dev ( `node_modules` ) : 600 MB
- Build tools (TypeScript, Webpack) : 200 MB
- Application buildée : 50 MB
- Code source : 50 MB

On n'a besoin QUE de l'application buildée en production !

# Solution : Multi-stage builds

Principe :

- 1 **Stage 1 (builder)** : Installer les dépendances et builder l'application
- 2 **Stage 2 (runtime)** : Copier uniquement le build final

Avantages :

- Image finale légère
- Pas de dépendances de dev en production
- Pas d'outils de build en production
- Meilleure sécurité (surface d'attaque réduite)

# Exemple : Application Node.js

```
# ✓ Multi-stage build (200 MB)
# Stage 1: Builder
FROM node:20 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Runtime
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY package*.json ./
RUN npm install --production
CMD ["node", "dist/index.js"]
```

# Décortiquons le multi-stage build

```
# Stage 1: Builder
FROM node:20 AS builder          # ← Nommer le stage "builder"
WORKDIR /app
COPY package*.json ./
RUN npm install                   # ← Dépendances dev + prod
COPY . .
RUN npm run build                # ← Build l'application

# Stage 2: Runtime
FROM node:20-alpine              # ← Image plus légère
WORKDIR /app
COPY --from=builder /app/dist ./dist # ← Copier DEPUIS le stage builder
COPY package*.json ./
RUN npm install --production      # ← Seulement les dépendances prod
CMD ["node", "dist/index.js"]
```

# Optimisation : Image de base

Choisir la bonne image de base :

Image	Taille	Usage
node:20	~900 MB	Développement, build
node:20-slim	~200 MB	Production simple
node:20-alpine	~120 MB	Production optimisée

Pour le multi-stage :

- Builder : node:20 (complet, outils de build)
- Runtime : node:20-alpine (léger, production)

# Exemple : Application TypeScript

```
# Stage 1: Build TypeScript
FROM node:20 AS builder
WORKDIR /app
COPY package*.json tsconfig.json ./
RUN npm install
COPY src ./src
RUN npm run build

# Stage 2: Runtime
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package*.json ./
RUN npm install --production
USER node
CMD ["node", "dist/index.js"]
```

Taille finale : ~150 MB au lieu de 1.2 GB !

# Exemple : Application Go

```
# Stage 1: Build
FROM golang:1.21 AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o app .

# Stage 2: Runtime
FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/app .
CMD ["./app"]
```

Taille finale : ~15 MB (Go produit un binaire statique) 🚀

# Exemple : Frontend React

```
# Stage 1: Build
FROM node:20 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Nginx
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Taille finale : ~50 MB (juste Nginx + fichiers statiques)

# Multi-stage : Plusieurs builders

```
# Stage 1: Build dependencies
FROM node:20 AS deps
WORKDIR /app
COPY package*.json ./
RUN npm install

# Stage 2: Build application
FROM node:20 AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN npm run build

# Stage 3: Build assets
FROM node:20 AS assets
WORKDIR /app
COPY --from=builder /app/public ./public
RUN npm run optimize-images
```

```
# Stage 4: Runtime
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=assets /app/public ./public
CMD ["node", "dist/index.js"]
```

# Concepts avancés : Dépendances entre stages

Comment Docker résout les dépendances :

Quand vous buildez une image multi-stage, Docker analyse les instructions pour déterminer quels stages exécuter.

Deux types de dépendances :

- 1 `FROM <stage>` : Héritage du système de fichiers
- 2 `COPY --from=<stage>` : Copie de fichiers depuis un stage

Résolution intelligente :

- Docker exécute **uniquement les stages nécessaires** pour le stage final
- Les stages non référencés sont ignorés

# Exemple : Quels stages sont exécutés ?

```
FROM node:20 AS deps          # Stage 1
RUN npm install

FROM node:20 AS builder        # Stage 2
COPY --from=deps /app .
RUN npm run build

FROM node:20 AS tester         # Stage 3
COPY --from=builder /app .
RUN npm test

FROM node:20-alpine           # Stage 4 (final)
COPY --from=builder /app/dist .
```

## Stages exécutés pour le build final :

- 1  `deps` (référencé par `builder`)
- 2  `builder` (référencé par le stage final)
- 3  `tester` (non référencé, ignoré)
- 4  Stage final (dernier FROM)

# Builder un stage spécifique avec Docker

Nous verrons ensuite comment builder avec Compose :

```
docker compose build
```

Mais vous pouvez builder directement avec Docker :

```
# Build l'image complète (dernier stage)
docker build -t myapp:latest .

# Build un stage spécifique avec --target
docker build -t myapp:builder --target builder .
docker build -t myapp:tester --target tester .
docker build -t myapp:dev --target development .
```

## Utilité du `--target` :

- Builder une image de dev (avec hot-reload)
- Créer une image de test (avec outils de test)
- Débugger un stage intermédiaire

# Exemple pratique : --target

```
FROM node:20 AS base
WORKDIR /app
COPY package*.json ./

FROM base AS development          # Stage pour dev
RUN npm install
COPY . .
CMD ["npm", "run", "dev"]

FROM base AS builder              # Stage pour build
RUN npm install
COPY . .
RUN npm run build

FROM node:20-alpine AS production # Stage pour prod
COPY --from=builder /app/dist ./dist
RUN npm install --production
CMD ["node", "dist/index.js"]
```

```
# Development : créer une image avec hot-reload
docker build -t myapp:dev --target development .
docker run -v $(pwd)/src:/app/src myapp:dev

# Tests : créer une image avec les tests
docker build -t myapp:test --target builder .
docker run myapp:test npm test

# Production : image finale optimisée
docker build -t myapp:prod --target production .
# Ou simplement (production est le dernier stage) :
docker build -t myapp:prod .
```

# Cache multi-stage : Comment ça fonctionne ?

Principe du cache par layer :

- Docker met en cache chaque instruction ( `FROM` , `RUN` , `COPY` )
- Si l'instruction et son contexte n'ont pas changé → réutilisation du cache
- Le cache est **partagé entre les stages**

## Exemple :

```
FROM node:20 AS builder
COPY package*.json ./
RUN npm install           # ← Mis en cache si package.json inchangé
COPY . .
RUN npm run build

FROM node:20-alpine
COPY --from=builder /app/dist ./dist
RUN npm install --production # ← Cache indépendant du stage builder
```

# Cache multi-stage : Optimisations

```
# ✅ Bon : Cache optimisé
FROM node:20 AS builder
WORKDIR /app
COPY package*.json ./          # Change rarement
RUN npm install                 # Cache préservé
COPY . .                         # Change souvent
RUN npm run build                # Cache si code inchangé

FROM node:20-alpine
COPY --from=builder /app/dist ./dist
```

# Cache : Visualiser les layers

```
# Build avec affichage du cache
docker build -t myapp .

# Output :
# [builder 1/5] FROM docker.io/library/node:20
# [builder 2/5] COPY package*.json ./
# [builder 3/5] RUN npm install
# => CACHED # ← Layer mis en cache !
# [builder 4/5] COPY . .
# [builder 5/5] RUN npm run build
```

## Tips pour maximiser le cache :

- Copier `package.json` avant le code source
- Grouper les `RUN` qui changent ensemble
- Utiliser `.dockerignore` pour exclure les fichiers temporaires
- Ne pas copier `node_modules` ou `dist` depuis le host

# Récapitulatif : Concepts avancés

## Dépendances entre stages :

- Docker exécute uniquement les stages nécessaires
- `FROM <stage>` et `COPY --from=<stage>` créent des dépendances
- Les stages non référencés sont ignorés

## Builder avec `--target` :

- `docker build --target <stage>` pour builder un stage spécifique
- Utile pour dev, test, debug
- Alternative à Docker Compose pour certains cas

## Cache multi-stage :

- Cache partagé entre stages
- Ordre des instructions crucial
- Copier les fichiers stables d'abord (`package.json`)

# Bonnes pratiques multi-stage

## ✓ Faire

- Utiliser Alpine pour le runtime
- Copier uniquement ce qui est nécessaire
- Installer seulement les deps prod
- Utiliser `USER node` (non-root)
- Nommer les stages explicitement

## ✗ Éviter

- Copier tout le `node_modules`
- Installer des deps dev en prod
- Utiliser root en production
- Oublier le `.dockerignore`
- Images de base trop lourdes

# .dockerignore : Essentiel

```
# .dockerignore
node_modules
npm-debug.log
.git
.gitignore
.env
.env.local
dist
build
coverage
*.md
.vscode
.idea
```

## Impact :

- Accélère le build (moins de fichiers à copier)
- Réduit la taille du contexte de build
- Évite de copier des secrets accidentellement

# Build avec Docker Compose

```
services:  
  frontend:  
    build:  
      context: ./frontend  
      dockerfile: Dockerfile  
      target: production  # ← Choisir un stage spécifique  
    ports:  
      - "80:80"  
  
  backend:  
    build:  
      context: ./backend  
      dockerfile: Dockerfile  
      target: development  # ← Stage différent pour dev  
    ports:  
      - "3000:3000"
```

# Multi-stage : Mode dev vs prod

```
# Stage 1: Base
FROM node:20 AS base
WORKDIR /app
COPY package*.json ./

# Stage 2: Development
FROM base AS development
RUN npm install
COPY . .
CMD ["npm", "run", "dev"]

# Stage 3: Builder
FROM base AS builder
RUN npm install
COPY . .
RUN npm run build
```

```
# Stage 4: Production
FROM node:20-alpine AS production
WORKDIR /app
COPY --from=builder /app/dist ./dist
RUN npm install --production
CMD ["node", "dist/index.js"]
```

# Utiliser les stages en Compose

```
# docker-compose.yml (prod)
services:
  backend:
    build:
      context: ./backend
      target: production
```

```
# docker-compose.dev.yml
services:
  backend:
    build:
      context: ./backend
      target: development
volumes:
  - ./backend/src:/app/src
```

```
# Development
docker compose -f docker-compose.yml -f docker-compose.dev.yml up

# Production
docker compose up
```

# Comparaison : Avant / Après

## Sans multi-stage

- Taille : 1.2 GB
- Dépendances dev incluses
- Outils de build en prod
- Surface d'attaque élevée
- Temps de pull : ~5 min

## Avec multi-stage

- Taille : 150 MB
- Seulement deps prod
- Pas d'outils de build
- Surface d'attaque réduite
- Temps de pull : ~30 sec

# Cache des layers : Optimisation

```
# ❌ Mauvais ordre (rebuild complet à chaque changement de code)
FROM node:20 AS builder
WORKDIR /app
COPY . .                      # ← Copie tout (change souvent)
RUN npm install                 # ← Rebuild à chaque fois
RUN npm run build
```

```
# ✅ Bon ordre (cache npm install)
FROM node:20 AS builder
WORKDIR /app
COPY package*.json ./          # ← Copie d'abord les deps (change rarement)
RUN npm install                 # ← Mis en cache
COPY . .                      # ← Copie le code (change souvent)
RUN npm run build
```

# Exercice : Optimiser cette image

```
FROM node:20
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build
RUN npm install --production
CMD ["node", "dist/index.js"]
```

## Problèmes :

- Pas de multi-stage
- Mauvais ordre de COPY
- Dépendances dev restent
- Image lourde

# Correction : Image optimisée

```
# Stage 1: Builder
FROM node:20 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Production
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY package*.json ./
RUN npm install --production
USER node
CMD ["node", "dist/index.js"]
```

Gains : 1.2 GB → 150 MB (87% de réduction) 🎉

# Points clés à retenir

## Multi-stage builds :

- Images plus légères
- Meilleure sécurité
- Séparation build / runtime
- Optimisation du cache

## Bonnes pratiques :

- Utiliser Alpine pour le runtime
- Copier uniquement le nécessaire
- Ordre des COPY pour le cache
- `.dockerignore` bien configuré

**En production :** Multi-stage = obligatoire !