



DevOps & Docker - Jour 3 Matin

Master 1 Développement Fullstack



Rappel Jour 2

Docker Compose et architecture multi-services

Ce que vous avez appris hier

Docker Compose

- Orchestration locale de services
- Fichier `docker-compose.yml` déclaratif
- Gestion des dépendances (`depends_on`, health checks)
- Variables d'environnement

Multi-stage builds

- Optimisation de la taille des images
- Séparation build/runtime
- Builder stage vs production stage

Stack complète

- Frontend + Backend + Base de données
- Réseaux customs
- Volumes persistants
- Configuration par environnement

TP de fin de journée

- Stack complète avec plusieurs services
- Hot-reload en développement
- Variables d'environnement
- Debugging avec `docker compose logs`

Le projet : Task Manager

Structure de votre projet `demo-projet-devops` :

```
demo-projet-devops/
├── frontend/                      # Interface React/Vue
├── tasks-service/                 # API de gestion des tâches (TypeScript + Express)
│   ├── src/
│   │   └── domain/
│   │       └── task.test.ts    # Tests Vitest
│   └── Dockerfile
└── stats-service/                 # API de statistiques (TypeScript + Express)
    ├── src/
    │   └── domain/
    │       └── stats.test.ts  # Tests Vitest
    └── Dockerfile
docker-compose.yml      # Orchestration
```

Le problème d'aujourd'hui

Comment automatiser le build, les tests et le déploiement ?

Comment s'assurer que les images Docker sont correctement construites ?

Comment stocker les images Docker pour les réutiliser ?

Comment créer une pipeline CI qui fonctionne localement et en production ?

Aujourd'hui : CI/CD locale

CI = Continuous Integration : automatiser le build et les tests

CD = Continuous Deployment : automatiser le déploiement

Objectif : une pipeline CI complète qui tourne **sur votre machine**



CI/CD : Introduction

De la théorie à la pratique locale

Qu'est-ce que la CI/CD ?

Continuous Integration (CI)

- Automatiser la construction de l'application
- Tester automatiquement le code
- Déetecter les problèmes tôt
- Valider chaque changement

Continuous Deployment (CD)

- Automatiser le déploiement
- Livrer rapidement en production
- Réduire le risque d'erreur humaine
- Accélérer le feedback utilisateur

Approche traditionnelle

```
# Build manuel
cd tasks-service
npm install
npm run build

cd ../stats-service
npm install
npm run build
# Tests manuels
cd ../tasks-service
npm test

cd ../stats-service
npm test
# Build Docker
docker build -t tasks-service .
docker build -t stats-service .
# Push manuel
docker push ...
```

Problèmes :

- Long et répétitif
- Risque d'oubli
- Pas reproductible

Approche CI/CD

```
# Une seule commande  
task ci
```

Avantages :

- Automatisé
- Reproductible
- Rapide
- Documenté

Pourquoi commencer en local ?

Feedback rapide : tester la pipeline sans attendre le cloud

Pas de dépendance : pas besoin de GitHub Actions / GitLab CI pour commencer

Debugging facile : comprendre les erreurs avant de passer en prod

Principe DevOps : "Si ça ne marche pas localement, ça ne marchera pas en prod"

L'outil : Task (GoTask)

Un task runner moderne pour automatiser vos workflows



GoTask (Task)

Task runner moderne pour automatiser vos workflows

Qu'est-ce que Task ?

Task est un task runner écrit en Go, alternative moderne à make

Caractéristiques

- Fichier de configuration en YAML (`Taskfile.yml`)
- Syntaxe simple et lisible
- Cross-platform (Linux, macOS, Windows)
- Gestion des dépendances entre tâches
- Variables et templates
- Détection automatique des changements

Site officiel

 taskfile.dev

Pourquoi Task plutôt que Make ?

Critère	Make	Task
Syntaxe	Complexe (tabs!)	Simple (YAML)
Variables	Limitées	Riches (env, shell, templating)
Cross-platform	Limité	Natif
Documentation	Dense	Claire et moderne
Communauté	Ancienne	Active et croissante

Make est excellent pour le build C/C++, Task est parfait pour les workflows modernes.

Anatomie d'un Taskfile.yml

```
version: '3'

# Variables globales
vars:
  DOCKER_REGISTRY: localhost:5000

# Tâches
tasks:
  hello:
    desc: Dit bonjour
    cmds:
      - echo "Hello, DevOps!"

  build:
    desc: Build l'application
    deps: [install]    # Dépendance : exécute 'install' avant
    cmds:
      - npm run build
```

Les concepts clés

Task (tâche)

- Une unité de travail (commandes à exécuter)
- Peut avoir des dépendances
- Peut avoir des sources/cibles pour le cache

Variables

- Globales (`vars`)
- D'environnement (`env`)
- Dynamiques (shell, templates)

Dépendances

- Entre tâches (`deps`)
- Exécution parallèle ou séquentielle

Exemple simple

```
version: '3'

vars:
  APP_NAME: my-app
  VERSION: 1.0.0

tasks:
  install:
    desc: Installe les dépendances
    cmds:
      - npm install

  test:
    desc: Lance les tests
    deps: [install]
    cmds:
      - npm test
```

```
build:
  desc: Build l'app
  deps: [test]
  cmds:
    - npm run build
    - echo "{{ .APP_NAME }} v{{ .VERSION }}
```

Utilisation

```
# Lister les tâches
task --list

# Exécuter une tâche
task install

# Exécuter plusieurs tâches
task test build

# Voir la documentation
task --summary build
```

Résultat :

```
task: [install] npm install
✓ Dependencies installed

task: [test] npm test
✓ All tests passed

task: [build] npm run build
task: [build] echo "my-app v1.0.0"
my-app v1.0.0
```

Avantages de Task

Lisibilité

YAML simple et clair

Documentation intégrée

`desc` pour chaque tâche

Portabilité

Fonctionne partout (Linux, Mac, Windows)

Performance

Écrit en Go, très rapide

Intelligence

Cache et détection de changements

Communauté

Active et documentation excellente



Installation de Task

Mettre en place l'outil

Installation sur différentes plateformes

<https://taskfile.dev/docs/installation>

macOS (Homebrew)

```
brew install go-task
```

Vérification de l'installation

```
# Vérifier la version
task --version

# Devrait afficher quelque chose comme :
# v3.45.5
```

Aide et commandes utiles

```
# Afficher l'aide
task --help

# Lister les tâches disponibles
task --list

# Voir le détail d'une tâche
task --summary nom-de-la-tache

# Mode verbose (debug)
task --verbose nom-de-la-tache
```

Initialisation d'un projet

```
# Crée un Taskfile.yml de base  
task --init
```

Crée un fichier `Taskfile.yml` avec la structure minimale :

```
version: '3'  
  
tasks:  
  default:  
    cmds:  
      - echo "Hello Task!"  
    silent: true
```

Tester

```
task default
# Affiche : Hello Task!
```

À vous de jouer !

Atelier : Premier Taskfile



Atelier 1

Créer un Taskfile pour le projet

Objectif de l'atelier

Créer un `Taskfile.yml` à la racine du projet `demo-projet-devops` avec :

- 1 Une tâche pour lister les services
- 2 Une tâche pour installer les dépendances
- 3 Une tâche pour lancer les tests
- 4 Une tâche pour tout nettoyer

Étape 1 : Créer le Taskfile.yml

```
cd demo-projet-devops
task --init
```

Modifier le fichier créé

```
version: '3'

vars:
  SERVICES: tasks-service stats-service frontend

tasks:
  default:
    desc: Affiche les tâches disponibles
    cmds:
      - task --list
    silent: true
```

Tester :

task

Étape 2 : Ajouter l'installation des dépendances

```
tasks:  
  install:  
    desc: Installe les dépendances de tous les services  
    cmd:  
      - echo "📦 Installation des dépendances..."  
      - cd tasks-service && npm install  
      - cd stats-service && npm install  
      - cd frontend && npm install  
      - echo "✅ Dépendances installées"
```

Tester :

```
task install
```

Question : Comment améliorer cette tâche pour éviter la répétition ?

Étape 3 : Utiliser les boucles et variables

```
version: '3'

vars:
  SERVICES: tasks-service stats-service frontend
  NODE_SERVICES: tasks-service stats-service
  # Le frontend sera géré séparément

tasks:
  install:
    desc: Installe les dépendances de tous les services
    cmd:
      - for: { var: NODE_SERVICES, split: ' ' }
        cmd: |
          echo "📦 Installation de {{.ITEM}}..."
          cd {{.ITEM}} && npm install
      - echo "✅ Toutes les dépendances sont installées"
```

Étape 4 : Ajouter les tests

```
tasks:  
  test:  
    desc: Lance les tests de tous les services  
    deps: [install]  
    cmds:  
      - echo "📝 Lancement des tests..."  
      - cd tasks-service && npm test  
      - cd stats-service && npm test  
      - echo "✅ Tous les tests sont passés"
```

Tester :

```
task test
```

Note : `deps: [install]` s'assure que les dépendances sont installées avant de lancer les tests.

Étape 5 : Ajouter le nettoyage

```
tasks:  
  clean:  
    desc: Nettoie les dossiers node_modules et dist  
    cmds:  
      - echo "👉 Nettoyage..."  
      - rm -rf tasks-service/node_modules tasks-service/dist  
      - rm -rf stats-service/node_modules stats-service/dist  
      - rm -rf frontend/node_modules frontend/dist  
      - echo "✅ Nettoyage terminé"
```

Tester :

```
task clean
```

Vérification

```
# Lister les tâches
task --list

# Devrait afficher :
task: Available tasks for this project:
* default:    Affiche les tâches disponibles
* clean:       Nettoie le projet
* install:    Installe les dépendances
* test:        Lance les tests
```

Points clés

- Structure claire et maintenable
- Utilisation de variables pour éviter la répétition
- Gestion des dépendances entre tâches
- Documentation intégrée avec `desc`



Docker Build

docker build **vs** docker compose build

Deux approches pour builder des images

docker build

- Commande Docker **native**
- Build une image à partir d'un Dockerfile
- **Un service à la fois**
- Contrôle fin sur le build

docker compose build

- Commande Docker Compose
- Build **tous les services** définis dans docker-compose.yml
- **Ou un service spécifique** avec un nom
- Pratique pour les stacks multi-services

docker build

```
# Build une image
docker build -t tasks-service:latest ./tasks-service

# Build avec un Dockerfile spécifique
docker build -f Dockerfile.dev -t tasks-service:dev ./tasks-service

# Build multi-stage avec target
docker build --target production -t tasks-service:prod ./tasks-service

# Build avec build args
docker build --build-arg NODE_VERSION=20 -t tasks-service:latest ./tasks-service
```

docker compose build

```
# Build tous les services
docker compose build

# Build un service spécifique
docker compose build tasks-service

# Build sans cache
docker compose build --no-cache

# Build en parallèle
docker compose build --parallel

# Build avec arguments
docker compose build \
  --build-arg NODE_VERSION=20 \
  tasks-service
```

Rappel : docker-compose.yml avec build

```
services:  
  tasks-service:  
    build:  
      context: ./tasks-service  
      dockerfile: Dockerfile  
      target: production  
      args:  
        NODE_VERSION: 20  
    image: tasks-service:latest  
  
  stats-service:  
    build:  
      context: ./stats-service  
      dockerfile: Dockerfile  
      target: production  
    image: stats-service:latest
```

Quand utiliser quoi ?

Utiliser docker build quand :

- Vous voulez builder **une seule image**
- Vous avez besoin de **contrôle fin** (build args, target, etc.)
- Vous créez un **script CI/CD** personnalisé
- Vous travaillez sur **un service isolé**

Utiliser docker compose build quand :

- Vous voulez builder **tous les services** d'une stack
- Vous travaillez en **développement local**
- Vous voulez **simplicité** et **cohérence** avec docker-compose.yml
- Vous voulez builder **en parallèle**

Exemple pratique

Projet demo-projet-devops

```
demo-projet-devops/
├── tasks-service/
│   └── Dockerfile
├── stats-service/
│   └── Dockerfile
└── frontend/
    └── Dockerfile
docker-compose.yml
```

Approche 1: docker build

```
docker build -t tasks-service:v1 \
./tasks-service

docker build -t stats-service:v1 \
./stats-service

docker build -t frontend:v1 \
./frontend
```

3 commandes, contrôle total

Approche 2 : docker compose build

```
docker compose build
```

1 commande, plus simple

Cohérence avec les deux approches

Dans votre Taskfile.yml

```
tasks:  
  build:docker:  
    desc: Build les images avec docker build  
    cmds:  
      - docker build -t tasks-service:latest ./tasks-service  
      - docker build -t stats-service:latest ./stats-service  
      - docker build -t frontend:latest ./frontend  
  
  build:compose:  
    desc: Build les images avec docker compose  
    cmds:  
      - docker compose build  
  
  build:  
    desc: Build les images (méthode par défaut)  
    cmds:  
      - task: build:compose
```

Bonnes pratiques

Tag cohérent

Utilisez toujours les mêmes tags

Build context

Gardez-le minimal avec `.dockerignore`

Multi-stage builds

Optimisez la taille des images

Cache

Utilisez `--no-cache` quand nécessaire

Parallélisation

```
docker compose build --parallel
```

CI/CD



Atelier 2

Builder les services avec Task

Objectif de l'atelier

Ajouter au `Taskfile.yml` :

- 1 Une tâche pour builder avec `docker build`
- 2 Une tâche pour builder avec `docker compose build`
- 3 Une tâche pour builder un service spécifique
- 4 Tester les deux approches

Durée : 20 minutes

Étape 1 : Ajouter les variables

```
version: '3'

vars:
  NODE_SERVICES: tasks-service stats-service
  ALL_SERVICES: tasks-service stats-service frontend
  DOCKER_TAG: latest

tasks:
  # ... tâches existantes
```

Étape 2 : Build avec docker build

```
tasks:  
  build:docker:  
    desc: Build les images avec docker build  
    cmd:  
      - echo "↙ Build des images Docker..."  
      - for: { var: ALL_SERVICES, split: ' ' }  
        cmd: |  
          echo "Building {{.ITEM}}..."  
          docker build -t {{.ITEM}}:{{.DOCKER_TAG}} ./{{.ITEM}}  
      - echo "✓ Build terminé"
```

Tester :

```
task build:docker
```

Étape 3 : Build avec docker compose

```
tasks:  
  build:compose:  
    desc: Build les images avec docker compose  
    cmd:  
      - echo "↙ Build avec Docker Compose..."  
      - docker compose build  
      - echo "✅ Build terminé"  
  
  build:compose:service:  
    desc: Build un service spécifique avec docker compose  
    cmd:  
      - echo "↙ Build de {{.CLI_ARGS}}..."  
      - docker compose build {{.CLI_ARGS}}
```

Tester :

```
task build:compose  
task build:compose:service -- tasks-service
```

Étape 4 : Build par défaut

```
tasks:  
  build:  
    desc: Build les images (méthode par défaut = compose)  
    cmd:  
      - task: build:compose
```

Tester :

```
task build
```

Alternative : Build avec tests

```
tasks:  
  build:all:  
    desc: Tests puis build  
    cmds:  
      - task: test  
      - task: build
```

Étape 5 : Build sans cache

```
tasks:  
  build:no-cache:  
    desc: Build sans cache  
    cmds:  
      - echo "↗ Build sans cache..."  
      - docker compose build --no-cache  
  
  rebuild:  
    desc: Clean, install et rebuild  
    cmds:  
      - task: clean  
      - task: install  
      - task: build:no-cache
```

Tester :

```
task build:no-cache  
task rebuild
```

Étape 6 : Vérification des images

```
tasks:  
  images:  
    desc: Liste les images Docker  
    cmd:  
      - docker images | grep -E "(tasks-service|stats-service|frontend|REPOSITORY)"
```

Tester :

```
task images
```

Résultat attendu :

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
tasks-service	latest	abc123...	2 minutes ago	150MB

Points clés

Flexibilité

Deux approches disponibles

Simplicité

`task build` suffit

Cohérence

Variables centralisées

Contrôle

Build spécifique si besoin



Tests avec Vitest

Intégrer les tests dans la pipeline CI

Vitest : Test runner moderne

Vitest est un framework de test pour JavaScript/TypeScript

Caractéristiques

- **Rapide** : utilise Vite pour la compilation
- **Compatible** avec Jest (API similaire)
- **ESM native** : support des modules ES6
- **TypeScript** : support natif
- **Watch mode** : re-exécution automatique

🔗 vitest.dev

Tests dans le projet

```
demo-projet-devops/
└── tasks-service/
    ├── src/
    │   └── domain/
    │       └── task.test.ts      # ✓ Tests Vitest
    │
    └── package.json            # "test": "vitest run"

└── stats-service/
    ├── src/
    │   └── domain/
    │       └── stats.test.ts    # ✓ Tests Vitest
    │
    └── package.json            # "test": "vitest run"
```

Lancer les tests

En local (npm)

```
cd tasks-service  
npm test  
  
cd ../stats-service  
npm test
```

Avec watch mode

```
npm run test:watch
```

Avec coverage

```
npm run test:coverage
```

Tests avec Docker Compose

Configuration multi-fichiers

```
# Lancer les tests de tasks-service
docker compose -f docker-compose.yml \
    -f docker-compose.override.test.yml \
    up --build --no-deps tasks-service

# Lancer les tests de stats-service
docker compose -f docker-compose.yml \
    -f docker-compose.override.test.yml \
    up --build --no-deps stats-service
```

Avantages :

- Tests dans un environnement isolé
- Pas besoin d'installer les dépendances localement
- Flag `--no-deps` pour ignorer les dépendances

Tests dans la CI

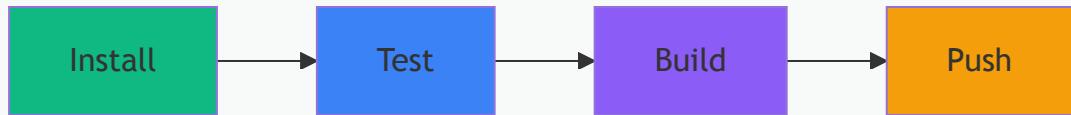
Pourquoi tester dans la CI ?

Détection précoce des régressions

Validation avant le build Docker

Confiance dans le code déployé

Pipeline CI idéale



- 1 **Install** : Installer les dépendances
- 2 **Test** : Lancer les tests unitaires
- 3 **Build** : Builder les images Docker
- 4 **Push** : Pousser vers un registry

Principe : Si les tests échouent, on ne build pas !

Gestion des erreurs

Test qui échoue

```
$ npm test

FAIL  src/domain/task.test.ts
  Task
    ✓ should create a task with valid data
    ✗ should mark task as completed

      Expected: true
      Received: false

  Tests:  1 failed, 1 passed, 2 total
```

⚠ La CI doit s'arrêter et ne pas continuer le build

À vous de jouer !

Atelier : Intégrer les tests dans Task



Atelier 3

Intégrer les tests dans le Taskfile

Objectif de l'atelier

Améliorer le `Taskfile.yml` pour :

- 1 Lancer les tests avant le build
- 2 Gérer les erreurs de tests
- 3 Ajouter un mode watch pour le développement
- 4 Créer une tâche CI complète

Durée : 15 minutes

Étape 1 : Améliorer la tâche test

```
tasks:  
  test:  
    desc: Lance tous les tests  
    deps: [install]  
    cmds:  
      - echo "📝 Lancement des tests..."  
      - for: { var: NODE_SERVICES, split: ' ' }  
        cmd: |  
          echo "Testing {{.ITEM}}..."  
          cd {{.ITEM}} && npm test  
      - echo "✅ Tous les tests sont passés"
```

Tester :

```
task test
```

Note : Si un test échoue, Task s'arrête automatiquement ✨

Étape 2 : Test en mode watch

```
tasks:  
  test:watch:  
    desc: Lance les tests en mode watch  
    cmds:  
      - echo "⌚ Mode watch activé (Ctrl+C pour quitter)"  
      - echo "Choisir le service : tasks-service ou stats-service"  
      - cd {{.CLI_ARGS}} && npm run test:watch
```

Tester :

```
task test:watch -- tasks-service
```

Pratique pour le développement local !

Étape 3 : Test d'un service spécifique

```
tasks:  
  test:service:  
    desc: Test un service spécifique  
    cmd:  
      - echo "📝 Test de {{CLI_ARGS}}..."  
      - cd {{CLI_ARGS}} && npm test
```

Tester :

```
task test:service -- tasks-service  
task test:service -- stats-service
```

Étape 4 : Pipeline CI complète

```
tasks:  
  ci:  
    desc: Pipeline CI complète (install → test → build)  
    cmds:  
      - echo "🚀 Démarrage de la pipeline CI..."  
      - task: install  
      - task: test  
      - task: build  
      - echo "✅ Pipeline CI terminée avec succès"
```

Tester :

```
task ci
```

Résultat :

- ✓ Install les dépendances
- ✓ Lance les tests
- ✓ Build les images Docker
- ⚠ S'arrête dès qu'une étape échoue

Étape 5 : Coverage (bonus)

```
tasks:  
  test:coverage:  
    desc: Lance les tests avec coverage  
    deps: [install]  
    cmds:  
      - echo "📊 Tests avec coverage..."  
      - for: { var: NODE_SERVICES, split: ' ' }  
        cmd: |  
          echo "Coverage de {{.ITEM}}..."  
          cd {{.ITEM}} && npm run test:coverage
```

Tester :

```
task test:coverage
```

Gestion des erreurs : Test pratique

Créer un test qui échoue

Dans `tasks-service/src/domain/task.test.ts`, ajouter :

```
it('should fail on purpose', () => {
  expect(1 + 1).toBe(3) // ✗ Va échouer !
})
```

Lancer la CI

```
task ci
```

Résultat attendu :

```
Démarrage de la pipeline CI...
 Installation...
 Dépendances installées
 Lancement des tests...
 Testing tasks-service...
 FAIL src/domain/task.test.ts
Error: task: Failed to run task "test": exit status 1
```

 La pipeline s'arrête, le build n'est pas lancé !

Points clés

Automatisation

Une commande = toute la CI

Sécurité

Arrêt dès qu'un test échoue

Flexibilité

Test global ou service spécifique

Productivité

Mode watch pour le dev



Récapitulatif

Ce que vous avez appris

- ✓ **GoTask** : automatiser les workflows
- ✓ **Taskfile.yml** : définir des tâches réutilisables
- ✓ **docker build vs docker compose build** : deux approches
- ✓ **Tests Vitest** : intégrer les tests dans la CI
- ✓ **Pipeline CI locale** : install → test → build



Prochaine étape

Container Registry local

Stocker et partager vos images Docker localement



Container Registry

Stocker et partager vos images Docker

Qu'est-ce qu'un Container Registry ?

Un **Container Registry** est un dépôt pour stocker et distribuer des images Docker

Analogie

- **npm** pour les packages Node.js
- **PyPI** pour les packages Python
- **Docker Registry** pour les images Docker

Fonctionnalités

- **Stockage** d'images Docker
- **Versioning** (tags)
- **Partage** entre développeurs/serveurs
- **Sécurité** (authentification, scan de vulnérabilités)

Registries publics vs privés

Registries publics

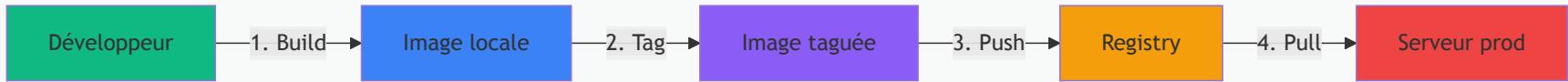
Service	Description	Gratuit ?
Docker Hub	Registry officiel Docker	<input checked="" type="checkbox"/> Oui (limité)
GitHub Container Registry	ghcr.io	<input checked="" type="checkbox"/> Oui
GitLab Container Registry	intégré GitLab	<input checked="" type="checkbox"/> Oui
Quay.io	Red Hat	<input checked="" type="checkbox"/> Oui (limité)

Registries publics vs privés

Registries privés

- **Self-hosted** (Docker Registry, Harbor)
- **Cloud** (AWS ECR, Google GCR, Azure ACR)
- **Enterprise** (JFrog Artifactory, Nexus)

Workflow typique avec un registry



- 1 **Build** : Créer l'image localement
- 2 **Tag** : Donner un nom cohérent (avec version)
- 3 **Push** : Envoyer vers le registry
- 4 **Pull** : Récupérer depuis n'importe où

Tags et versioning

Conventions de nommage

```
<registry>/<namespace>/<image>:<tag>  
  
# Exemples  
docker.io/library/nginx:1.25.3  
ghcr.io/mycompany/myapp:v1.0.0  
localhost:5000/tasks-service:latest
```

Bonnes pratiques

- ✓ `v1.0.0` : version sémantique
- ✓ `main-abc123f` : branche + commit SHA
- ✓ `2024-01-15` : date
- ⚠ `latest` : à éviter en production (pas reproductible)

Pourquoi un registry local ?

Développement : tester la pipeline complète localement

Pas de dépendance externe : pas besoin de Docker Hub, GitHub, etc.

Rapide : push/pull en local est instantané

Formation : comprendre le fonctionnement d'un registry

Solution : Docker Registry 3.0

Registry officiel Docker, auto-hébergé



Docker Registry Local

registry:3.0.0

Qu'est-ce que Docker Registry ?

Docker Registry est l'implémentation officielle d'un registry Docker

Caractéristiques

- **Open-source** : projet officiel de Docker
- **Léger** : image Docker de ~10 MB
- **Simple** : pas de base de données externe
- **Stockage** : local ou S3/Azure/GCS
- **API REST** : compatible avec tous les clients Docker

🔗 github.com/distribution/distribution

Version 3.0.0

Version actuelle : 3.0.0 (stable)

Nouveautés 3.x

- Support OCI (Open Container Initiative)
- Performance améliorée
- Meilleures APIs
- Support des multi-arch images

Aujourd'hui : on utilise le registry officiel (plus simple)

Alternative : Harbor

- **Harbor** : registry entreprise (plus complet)
- Interface web, RBAC, scan de vulnérabilités
- Plus lourd mais plus de fonctionnalités

Lancer le registry local

```
docker run -d \
-p 5000:5000 \
--name registry \
registry:3.0.0
```

Vérifier que ça fonctionne

```
# Vérifier que le conteneur tourne
docker ps | grep registry

# Tester l'API
curl http://localhost:5000/v2/
# Devrait retourner : {}

# Lister les images (vide au début)
curl http://localhost:5000/v2/_catalog
# Devrait retourner : {"repositories":[]}
```

Utiliser le registry

1. Builder une image

```
docker build -t tasks-service:v1 ./tasks-service
```

2. Taguer pour le registry local

```
docker tag tasks-service:v1 localhost:5000/tasks-service:v1
```

3. Pousser vers le registry

```
docker push localhost:5000/tasks-service:v1
```

4. Vérifier

```
curl http://localhost:5000/v2/_catalog
# {"repositories":["tasks-service"]}

curl http://localhost:5000/v2/tasks-service/tags/list
# {"name":"tasks-service","tags":["v1"]}
```

Pull depuis le registry

Supprimer l'image locale

```
docker rmi tasks-service:v1  
docker rmi localhost:5000/tasks-service:v1
```

Pull depuis le registry

```
docker pull localhost:5000/tasks-service:v1
```

Lancer le conteneur

```
docker run -p 3001:3001 localhost:5000/tasks-service:v1
```

Persistance des données

Problème : données éphémères

Si le conteneur est supprimé, les images sont perdues !

Solution : utiliser un volume

```
docker run -d \
-p 5000:5000 \
--name registry \
-v registry-data:/var/lib/registry \
registry:3.0.0
```

Les images sont maintenant persistées dans le volume `registry-data`

Configuration avancée

Fichier de configuration

```
version: 0.1
storage:
  filesystem:
    rootdirectory: /var/lib/registry
  delete:
    enabled: true
http:
  addr: :5000
```

Lancer avec la config

```
docker run -d -p 5000:5000 --name registry \
-v $(pwd)/config.yml:/etc/docker/registry/config.yml \
-v registry-data:/var/lib/registry registry:3.0.0
```

Gestion dans docker-compose.yml

```
services:  
  registry:  
    image: registry:3.0.0  
    container_name: local-registry  
    ports:  
      - "5000:5000"  
    volumes:  
      - registry-data:/var/lib/registry  
    restart: unless-stopped  
  
volumes:  
  registry-data:
```

```
docker compose up -d registry
```

Sécurité (aperçu)

En production

⚠ Le registry local **n'est pas sécurisé** par défaut

Pour sécuriser :

- **HTTPS** : certificat TLS obligatoire
- **Authentification** : basic auth, token, OAuth
- **Firewall** : restreindre l'accès
- **Reverse proxy** : Nginx, Traefik

Exemple avec authentification

```
htpasswd -Bbn admin password > htpasswd

docker run -d \
-p 5000:5000 \
-v $(pwd)/htpasswd:/auth/htpasswd \
-e REGISTRY_AUTH=htpasswd \
-e REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm" \
-e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
registry:3.0.0
```

Aujourd'hui : on reste en mode simple (pas d'auth)

À vous de jouer !

Atelier : Configurer et utiliser le registry local



Atelier 4

Configurer et utiliser le registry local

Objectif de l'atelier

- 1 Lancer un registry Docker local
- 2 Pousser des images vers le registry
- 3 Vérifier les images avec l'API
- 4 Intégrer le push dans le Taskfile.yml

Durée : 25 minutes

Étape 1 : Lancer le registry

```
# Lancer le registry
docker run -d \
-p 5000:5000 \
--name registry \
-v registry-data:/var/lib/registry \
registry:3.0.0

# Vérifier
docker ps | grep registry

# Tester l'API
curl http://localhost:5000/v2/
```

Résultat attendu : { }

Étape 2 : Push manuel d'une image

```
# Build l'image
docker build -t tasks-service:v1 ./tasks-service

# Tag pour le registry local
docker tag tasks-service:v1 localhost:5000/tasks-service:v1

# Push vers le registry
docker push localhost:5000/tasks-service:v1
```

Vérifier :

```
curl http://localhost:5000/v2/_catalog
# {"repositories":["tasks-service"]}

curl http://localhost:5000/v2/tasks-service/tags/list
# {"name":"tasks-service","tags":["v1"]}
```

Étape 3 : Ajouter le registry dans docker-compose.yml

```
services:  
  registry:  
    image: registry:3.0.0  
    container_name: local-registry  
    ports:  
      - "5000:5000"  
    volumes:  
      - registry-data:/var/lib/registry  
    restart: unless-stopped  
  
    # ... autres services  
  
volumes:  
  registry-data:
```

Étape 4 : Ajouter variables dans Taskfile.yml

```
vars:  
  ALL_SERVICES: tasks-service stats-service frontend  
  DOCKER_REGISTRY: localhost:5000  
  DOCKER_TAG: latest  
  VERSION: v1.0.0
```

Étape 5 : Créer la tâche tag

Étape 6 : Créer la tâche push

Étape 7 : Vérifier les images

Étape 8 : Pull depuis le registry

Étape 9 : Tâche pull dans Taskfile

Étape 10 : Pipeline complète

```
tasks:  
  ci:  
    desc: Pipeline CI (install → test → build → tag → push)  
    cmds:  
      - echo "🚀 Pipeline CI complète..."  
      - task: install  
      - task: test  
      - task: build  
      - task: tag  
      - task: push  
      - echo "✅ Pipeline CI terminée"  
      - task: registry:list
```

Tester :

```
task ci
```

Nettoyage

```
tasks:  
  registry:clean:  
    desc: Supprime toutes les images du registry  
    cmds:  
      - echo "⚠️ Suppression des images du registry..."  
      - docker compose down registry  
      - docker volume rm registry-data || true  
      - docker compose up -d registry  
      - echo "✅ Registry nettoyé"
```

Tester :

```
task registry:clean
```

Points clés

Registry local

Fonctionne comme en prod

Pipeline complète

Build → Test → Push

Versioning

Tags `latest` et `v1.0.0`

API REST

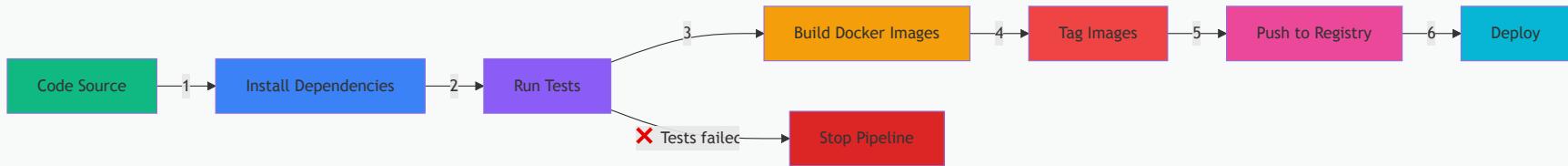
Inspection facile avec curl



Pipeline CI Complète

Vue d'ensemble et bonnes pratiques

Architecture de la pipeline



Les étapes de la pipeline

1. Install Dependencies

- Installer les dépendances npm
- Cache pour accélérer

2. Run Tests

- Tests unitaires avec Vitest
- ⚠ Stop si échec

3. Build Docker Images

- Build avec docker build ou docker compose
- Multi-stage builds

4. Tag Images

- Tag avec version et latest
- Convention de nommage cohérente

5. Push to Registry

- Push vers le registry local
- Ou Docker Hub, GHCR, etc.

Utilisation en pratique

Développement local

```
# Installer et tester rapidement
task test

# Builder une image spécifique
task build:compose:service -- tasks-service

# Watch mode pour les tests
task test:watch -- tasks-service
```

CI/CD local

```
# Pipeline complète
task ci

# Vérifier les images dans le registry
task registry:list
```

Bonnes pratiques

1. Tests d'abord : Toujours tester avant de builder

2. Cache intelligent : `npm ci` au lieu de `npm install`

3. Versioning clair : Utiliser des tags sémantiques (v1.0.0)

4. Fail fast : Arrêter dès qu'une étape échoue

Variables d'environnement

```
version: '3'

dotenv:
  - .env
  - .env.local

env:
  NODE_ENV: production
  DOCKER_BUILDKIT: 1
  COMPOSE_DOCKER_CLI_BUILD: 1

tasks:
  ci:
    env:
      CI: true
    cmds:
      - task: test
      - task: build
```

- `.env` : variables partagées
- `env:` : variables globales
- Task-specific env : variables par tâche

Optimisations

1. Parallélisation

```
tasks:  
  test:parallel:  
    desc: Teste tous les services en parallèle  
    deps:  
      - task: test:service  
        vars: { SERVICE: tasks-service }  
      - task: test:service  
        vars: { SERVICE: stats-service }
```

2. Cache Docker

```
tasks:  
  build:  
    cmd:  
      - docker compose build --parallel
```

3. Layers Docker optimisés

```
# Copier package.json d'abord (cache)  
COPY package*.json ./  
RUN npm ci
```

```
# Copier le code ensuite  
COPY . .
```

Gestion des erreurs

Arrêt automatique

```
tasks:  
  ci:  
    cmd:  
      - task: test      # ❌ Échoue → pipeline s'arrête  
      - task: build    # ⏸ N'est pas exécuté  
      - task: push     # ⏸ N'est pas exécuté
```

Continuer malgré les erreurs (à éviter !)

```
tasks:  
  lint:  
    ignore_error: true  # ⚠️ À éviter !  
    cmd:  
      - npm run lint
```

Comparaison avec d'autres outils

Feature	Task	Make	npm scripts	GitHub Actions
Syntaxe	YAML	Makefile	JSON	YAML
Cross-platform	✓	⚠	✓	Cloud
Variables	✓	⚠	⚠	✓
Dépendances	✓	✓	⚠	✓
Local	✓	✓	✓	✗
Documentation	✓	⚠	⚠	✓

Task : Parfait pour CI/CD locale

GitHub Actions : Pour CI/CD cloud

Combinaison : Task localement, GitHub Actions en prod



Vous avez maintenant une pipeline CI locale complète !

- ✓ Automatisation avec **Task**
- ✓ Tests avec **Vitest**
- ✓ Build avec **Docker**
- ✓ Push vers **Registry local**



Synthèse Jour 3 Matin

CI/CD locale avec Task

Ce que vous avez appris

1. CI/CD locale

- Pourquoi commencer en local
- Principe : install → test → build → push
- Feedback rapide et debugging facile

2. Task (GoTask)

- Task runner moderne en YAML
- Alternative à Make et npm scripts
- Gestion des dépendances entre tâches
- Variables et templating

3. Docker build vs docker compose build

- Deux approches complémentaires
- `docker build` : contrôle fin
- `docker compose build` : simplicité

4. Tests Vitest

- Framework de test moderne
- Intégration dans la pipeline CI
- Fail fast : arrêt si échec

Ce que vous avez appris (suite)

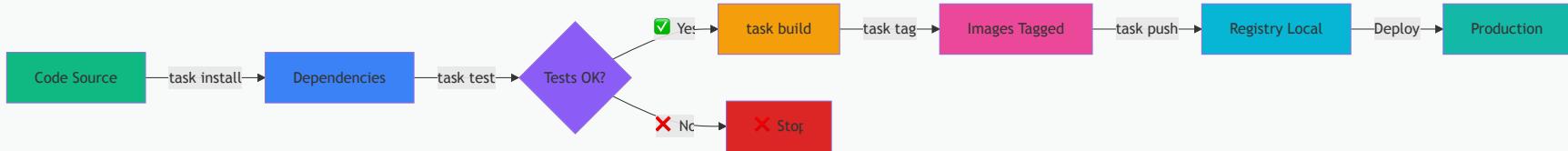
5. Docker Registry local

- Stockage d'images Docker
- registry:3.0.0
- Push/Pull en local
- API REST pour inspection

6. Pipeline CI complète

- Taskfile.yml complet
- Automatisation de bout en bout
- Versioning et tagging

Architecture finale



Commandes clés

```
# Lister les tâches
task --list

# Pipeline complète
task ci

# Étapes individuelles
task install
task test
task build
task push

# Inspection
task registry:list
task images

# Développement
task test:watch -- tasks-service
task build:compose:service -- tasks-service
```

Bénéfices

Automatisation

Plus de commandes manuelles

Rapidité

Feedback instantané

Fiabilité

Tests avant build

Portabilité

Fonctionne partout

Documentation

Taskfile = documentation

Reproductibilité

Même résultat à chaque fois

Questions ?

Posez vos questions avant la pause