

Jour 3 — Docker Compose & intégration continue

DevOps — Titre professionnel DWWM
ForEach Academy

Formateur : Fabrice Claeys

Programme de la journée

Horaire	Contenu
9h00-9h10	Accueil, où en est-on ?
9h10-10h05	Bloc 3.5 — Docker Compose + démo <code>demos/demo-compose/</code>
10h05-10h25	Bloc 3.6 — Base de données conteneurisée
10h25-10h35	Pause
10h35-12h15	TP fil rouge — Étape 2 : environnement complet avec Compose
13h15-14h00	Bloc 4.1 — GitHub Actions : concepts + démo <code>demos/demo-actions-hello/</code>
14h00-14h45	Bloc 4.2 — Pipeline de vérification
14h45-15h15	Bloc 4.3 — Secrets, variables, build d'images
15h15-15h25	Pause
15h25-16h50	TP fil rouge — Étape 3 : pipeline d'intégration continue
16h50-17h00	Récap, quiz, glossaire

Où en est-on sur la boucle DevOps ?



- **Hier** : vous avez écrit les `Dockerfile` de l'API et du front, lancé Postgres à la main
- **Ce matin** : tout TaskFlow démarre avec **une seule commande**
- **Cet après-midi** : chaque push est **vérifié automatiquement**, les images sont **publiées**

“ Objectif du soir : `docker compose up` lance TaskFlow, et chaque Pull Request (PR, demande de fusion) affiche un pipeline vert ou rouge. ”

Bloc 3.5 — Docker Compose

« J'ai oublié une option dans un des dix `docker run` ... »

Le problème : hier soir, pour lancer TaskFlow

```
docker network create taskflow
docker volume create pgdata
docker run -d --name db --network taskflow -v pgdata:/var/lib/postgresql/data \
  -e POSTGRES_USER=taskflow -e POSTGRES_PASSWORD=taskflow postgres:16-alpine
docker build -t taskflow-api ./api
docker run -d --name api --network taskflow \
  -e DATABASE_URL=postgres://taskflow:taskflow@db:5432/taskflow taskflow-api
docker build -t taskflow-front ./front
docker run -d --name front --network taskflow -p 8080:80 taskflow-front
```

- 8 commandes à taper **dans le bon ordre**, avec les bonnes options
- Un nouvel arrivant dans l'équipe ? Il faut lui envoyer... un message Slack ?
- On oublie `--network` : l'API ne trouve plus la base. On oublie `-v` : les données disparaissent



Ce qui n'est pas écrit dans un fichier finit par être oublié.



Compose : la partition et le chef d'orchestre

La partition : `compose.yaml`

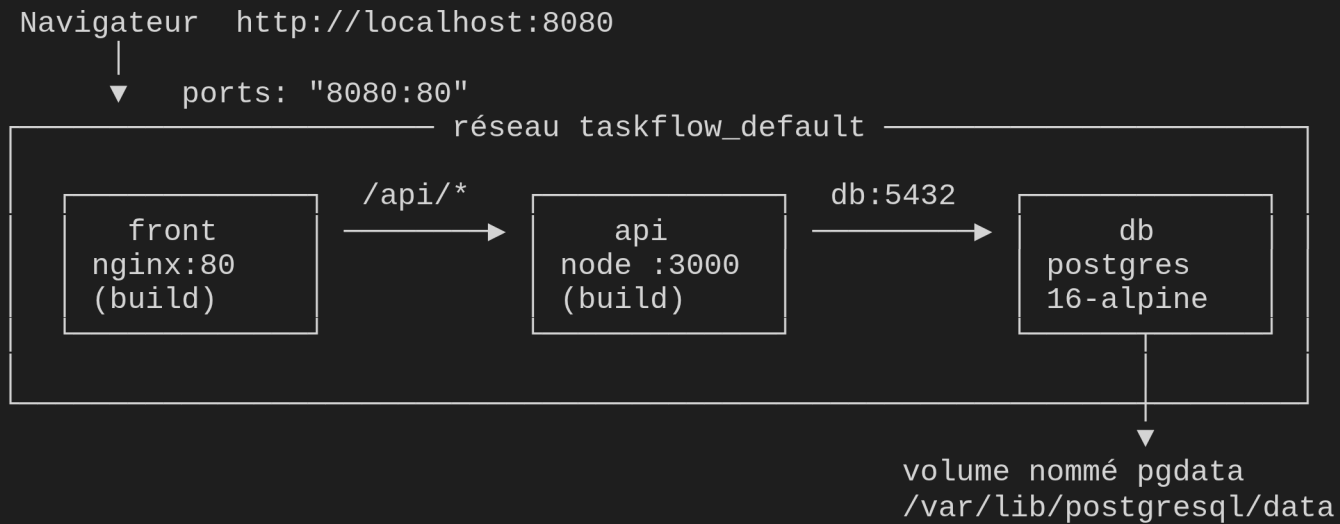
- Décrit **tous les musiciens** (services) : db, api, front
- Pour chacun : quelle image, quels ports, quelles variables, quels volumes
- Versionnée dans Git, avec le code

Le chef d'orchestre : `docker compose`

- Lit la partition
- Crée le réseau, les volumes, les conteneurs
- Fait entrer chaque musicien **au bon moment** (la base avant l'API)

“ Docker Compose v2 est intégré à Docker : la commande s'écrit `docker compose` (avec un espace).
L'ancien `docker-compose` (avec un tiret) est obsolète. ”

L'architecture que nous allons décrire



- **Un seul port** ouvert sur votre poste : 8080. L'API (interface de programmation, le back-end) et la base ne sont joignables que **dans le réseau**
- Les services se trouvent **par leur nom** : `db`, `api`, `front`

compose.yaml (1/4) — le service db

```
services:                                # la liste des conteneurs à lancer
  db:                                    # nom du service = nom DNS sur le réseau
    image: postgres:16-alpine           # image officielle, version figée
    environment:                         # variables lues au 1er démarrage de Postgres
      POSTGRES_USER: taskflow
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-taskflow} # lu dans .env, sinon taskflow
      POSTGRES_DB: taskflow
    volumes:
      - pgdata:/var/lib/postgresql/data # les données survivent au conteneur
```

- `image:` : on **utilise** une image existante, on ne construit rien
- `${POSTGRES_PASSWORD:-taskflow}` : valeur prise dans `.env` ; à défaut, `taskflow` (pratique en dev, interdit en prod)

compose.yaml (2/4) — le healthcheck de db

```
healthcheck:                                # « es-tu prêt ? » posé régulièrement
  test: ["CMD-SHELL", "pg_isready -U taskflow -d taskflow"]
  interval: 5s                               # toutes les 5 secondes
  timeout: 3s                                # une réponse lente = échec
  retries: 10                                 # 10 échecs d'affilée = unhealthy
  start_period: 10s                           # tolérance au démarrage
```

État affiché par docker compose ps

Signification

Up (health: starting)	Conteneur lancé, test pas encore réussi
Up (healthy)	pg_isready répond : Postgres accepte des connexions
Up (unhealthy)	Le test échoue en boucle : regarder les logs

“ Un conteneur « Up » n'est pas forcément « prêt » : Postgres met quelques secondes à s'initialiser. ”

compose.yaml (3/4) — le service api

```
api:
  build: ./api                                # construit l'image depuis api/Dockerfile
  environment:
    PORT: 3000
    DATABASE_URL: postgres://taskflow:${POSTGRES_PASSWORD:-taskflow}@db:5432/taskflow
    APP_ENV: local
  depends_on:
    db:
      condition: service_healthy              # attend que db soit healthy
  restart: unless-stopped                     # relance si le processus plante
```

- `build:` au lieu de `image:` : Compose construit l'image à **partir de votre code**
- Dans `DATABASE_URL`, l'hôte est `db` (le nom du service), pas `localhost`
- Pas de `ports:` : l'API n'est **pas exposée** sur votre poste, seul le front l'appelle
- Pas de `healthcheck:` ici : il est déjà défini dans `api/Dockerfile` (`HEALTHCHECK` sur `/health`)

`compose.yaml` (4/4) — le service `front` et les volumes

```
front:
  build: ./front                # multi-stage : build Vite → Nginx
  ports:
    - "8080:80"                # port du poste : port du conteneur
  depends_on:
    api:
      condition: service_healthy # attend que /health réponde

volumes:
  pgdata:                      # déclaration des volumes nommés
                                # créé et géré par Docker
```

- `"8080:80"` : toujours **entre guillemets** (YAML peut mal interpréter `xx:yy`)
- Le `nginx.conf` du `front` (écrit hier) relaie `/api/` et `/health` vers `http://api:3000`

“ Fichier complet : `tp/fil-rouge-taskflow/solution/compose.yaml` (à consulter **après** le TP). ”

`depends_on` : pourquoi la condition compte

Sans condition	Avec condition: <code>service_healthy</code>
t=0s db démarre	t=0s db démarre
t=0s api démarre	t=0s ... api attend ...
t=1s api : connexion à db	t=6s db : <code>pg_isready</code> OK → healthy
ECONNREFUSED → crash	t=6s api démarre
t=8s db enfin prête (trop tard)	t=7s api : connexion OK, migration

- `depends_on: [db]` seul : attend que le conteneur soit **lancé**, pas qu'il soit **prêt**
- `condition: service_healthy` : attend le **healthcheck vert**
- `restart: unless-stopped` en filet de sécurité : si l'API plante quand même, Docker la relance

Le réseau Compose et les noms de services

- Compose crée automatiquement un réseau `<dossier>_default` (ici `taskflow_default`)
- Chaque service y est enregistré **sous son nom** : un DNS (annuaire nom → adresse) interne

Depuis...	Pour joindre Postgres, j'écris...	Pourquoi
Le conteneur <code>api</code>	<code>db:5432</code>	<code>db</code> est résolu par le DNS du réseau Compose
Le conteneur <code>api</code>	<code>localhost:5432</code>	<code>localhost</code> = le conteneur <code>api</code> lui-même
Votre poste (hors Docker)	<code>localhost:5432</code>	seulement si <code>db</code> publie <code>ports: ["5432:5432"]</code>

“ Erreur n° 1 du jour : `ECONNREFUSED 127.0.0.1:5432` dans les logs de l'API = `localhost` au lieu de `db`. ”

Variables d'environnement : trois mécanismes

```
environment:                                # 1. valeurs écrites dans compose.yaml
  APP_ENV: local
env_file: .env                               # 2. toutes les lignes du fichier → dans le conteneur
image: postgres:${PG_VERSION}               # 3. interpolation : remplacée AVANT le lancement
```

```
# .env (à la racine, à côté de compose.yaml – JAMAIS commité)
POSTGRES_PASSWORD=un-mot-de-passe-local
# .env.example (commité : la liste des variables attendues, sans vraies valeurs)
POSTGRES_PASSWORD=changez-moi
```

- Compose lit **automatiquement** `.env` pour l'interpolation `${...}`
- `.env` dans `.gitignore` ; `.env.example` dans Git → le nouvel arrivant fait `cp .env.example .env`

Les commandes essentielles

Commande	Effet
<code>docker compose up -d</code>	Crée réseau, volumes, conteneurs et les lance en arrière-plan (<code>-d</code> = détaché)
<code>docker compose up -d --build</code>	Idem, en reconstruisant les images (après une modification du code)
<code>docker compose ps</code>	État et santé de chaque service
<code>docker compose logs -f api</code>	Suit les logs du service <code>api</code> (<code>Ctrl+C</code> pour quitter)
<code>docker compose exec db sh</code>	Ouvre un terminal dans le conteneur <code>db</code>
<code>docker compose stop</code>	Arrête les conteneurs sans les supprimer
<code>docker compose down</code>	Supprime conteneurs et réseau — le volume est conservé
<code>docker compose down -v</code>	Supprime aussi les volumes → données de la base perdues

“ `down -v` : à utiliser en développement pour repartir de zéro. Jamais sur un serveur sans sauvegarde. ”

Démo — demos/demo-compose/

1. Lire ensemble le `compose.yaml` de la démo : `web` (Node), `db` (Postgres 16), `adminer` (interface web de la base)
2. `cp .env.example .env` puis `docker compose up -d --build` ; `docker compose ps` : `health: starting` → `healthy`
3. `http://localhost:3002` : chaque visite est enregistrée dans Postgres, le compteur monte ;
`http://localhost:8081` : Adminer montre la table `visits`
4. `docker compose down` puis `up -d` : le compteur **est conservé** (volume)
5. `docker compose down -v` puis `up -d` : le compteur **repart à zéro**
6. Casser volontairement le nom d'hôte dans `environment:` → lire l'erreur avec `logs -f`

“ Pendant la démo, notez sur papier : quelle commande, quel résultat attendu. Ce sera votre antisèche pour le TP. ”

Si ça ne marche pas — pièges fréquents

Symptôme	Cause probable	Correction
<code>port is already allocated</code>	Le port 8080 est déjà pris sur votre poste	<code>"8081:80"</code> ou arrêter l'autre programme
API : <code>ECONNREFUSED 127.0.0.1:5432</code>	<code>localhost</code> dans <code>DATABASE_URL</code>	Remplacer par <code>db</code>
API qui plante au démarrage puis repart	Pas de <code>condition: service_healthy</code>	Ajouter le healthcheck et la condition
Modification du code non visible	Image pas reconstruite	<code>docker compose up -d --build</code>
<code>variable is not set. Defaulting to a blank string</code>	Fichier <code>.env</code> absent	<code>cp .env.example .env</code>
Mot de passe changé, connexion refusée	Volume déjà initialisé avec l'ancien	En dev : <code>down -v</code> puis <code>up -d</code>
<code>yaml: mapping values are not allowed</code>	Indentation incorrecte ou tabulation	Espaces uniquement, 2 par niveau

À retenir — Docker Compose

- `compose.yaml` = la **partition** de l'application, versionnée dans Git
- `services` → `image` ou `build`, `ports`, `environment` / `env_file`, `volumes`, `depends_on`
- Les services se joignent **par leur nom** sur le réseau créé par Compose
- `healthcheck` + `depends_on: condition: service_healthy` = démarrage dans le bon ordre
- `.env` (secret, ignoré par Git) / `.env.example` (modèle, commité)
- `up -d`, `ps`, `logs -f`, `down` ; `down -v` efface les données

Bloc 3.6 — Pourquoi mettre la base dans un conteneur ?

- **Même version pour tous** : `postgres:16-alpine` sur tous les postes, en CI et sur le serveur
- **Rien à installer** sur votre machine, rien à désinstaller après
- **Repartir de zéro** en une commande (`down -v`)

Variable	Rôle	Valeur TaskFlow
<code>POSTGRES_USER</code>	Utilisateur créé au premier démarrage	<code>taskflow</code>
<code>POSTGRES_PASSWORD</code>	Son mot de passe (obligatoire)	lu dans <code>.env</code>
<code>POSTGRES_DB</code>	Base créée au premier démarrage (défaut : nom de l'utilisateur)	<code>taskflow</code>

“ Ces variables ne sont lues **qu'au tout premier démarrage**, quand le volume est vide. Les changer ensuite n'a aucun effet. ”

Le volume est obligatoire

Sans volume

conteneur db
données ici

docker compose down
→ conteneur supprimé
→ DONNÉES PERDUES

Avec volume nommé pgdata

conteneur db

→ pgdata (sur le disque
géré par Docker)

docker compose down
→ conteneur supprimé
→ données intactes dans pgdata

- Le système de fichiers d'un conteneur est **jetable** : il disparaît avec lui
- Le chemin des données de Postgres est `/var/lib/postgresql/data` : c'est **lui** qu'on monte
- `docker volume ls` liste les volumes ; `docker volume inspect taskflow_pgdata` montre où ils sont

Script d'initialisation (optionnel)

```
db:
  volumes:
    - pgdata:/var/lib/postgresql/data
    - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql:ro # ro = lecture seule
```

```
-- db/init.sql : exécuté UNE SEULE FOIS, quand le volume est vide
INSERT INTO tasks (title) VALUES ('Découvrir Docker Compose');
```

- Tout fichier `.sql` ou `.sh` placé dans `/docker-entrypoint-initdb.d/` est exécuté au **premier** démarrage
- Utile pour des **données de démonstration** ou des réglages de la base
- Dans TaskFlow, la table est créée par l'API elle-même (`migrate()` dans `api/src/db.js`) : `init.sql` ne doit donc rien créer de concurrent

Parler à la base avec `psql`

```
docker compose exec db psql -U taskflow -d taskflow # client SQL dans le conteneur
```

```
taskflow=# \dt          -- lister les tables
taskflow=# SELECT * FROM tasks;  -- lire les tâches créées depuis le front
taskflow=# \q           -- quitter
```

```
# En une ligne, sans session interactive
docker compose exec db psql -U taskflow -d taskflow -c "SELECT count(*) FROM tasks;"
```

- `exec` lance une commande **dans un conteneur qui tourne déjà**
- Pas besoin d'installer `psql` sur votre poste : il est dans l'image

Et MySQL / MariaDB ?

	PostgreSQL	MySQL / MariaDB
Image	postgres:16-alpine	mysql:8.4 / mariadb:11
Mot de passe administrateur	POSTGRES_PASSWORD	MYSQL_ROOT_PASSWORD / MARIADB_ROOT_PASSWORD
Base et utilisateur	POSTGRES_DB , POSTGRES_USER	MYSQL_DATABASE , MYSQL_USER , MYSQL_PASSWORD
Dossier à mettre en volume	/var/lib/postgresql/data	/var/lib/mysql
Test de santé	pg_isready	mysqladmin ping / healthcheck.sh --connect
Client en ligne de commande	psql	mysql / mariadb

“ Même logique pour toutes les bases : **variables d'init + volume + healthcheck**. Seuls les noms changent. ”

TP fil rouge — Étape 2 : environnement complet (1h40)

Énoncé : `tp/fil-rouge-taskflow/etape2-compose.md`

1. Branche `feature/compose` dans votre dépôt `taskflow`
2. Créer `.env.example` (commité) et `.env` (ignoré par Git)
3. Écrire `compose.yaml` : `db` (healthcheck + volume), `api` (`build`), `front` (`build` , port 8080)
4. `docker compose up -d --build` → `docker compose ps` : tout est `Up`, `db` est `healthy`
5. Ouvrir `http://localhost:8080`, créer deux tâches, vérifier avec `psql`
6. `down` puis `up -d` : les tâches sont toujours là. `down -v` : elles ont disparu. Expliquez pourquoi

Exercices guidés si vous avez fini : `exercices/jour3/` (compose à compléter)

Étape 2 — Le test du nouvel arrivant

En binôme, l'un joue le nouveau développeur, sur **son** poste :

```
git clone https://github.com/<binome>/taskflow.git && cd taskflow
git switch feature/compose
cp .env.example .env
docker compose up -d --build
```

- Critère de réussite : TaskFlow fonctionne **sans aucune autre instruction** que le README
- Si ça bloque : c'est le README ou le `compose.yaml` qu'il faut corriger, pas le collègue
- Ensuite : **PR** vers `main`, **revue** par le binôme (relire `compose.yaml` ligne à ligne), merge

“

Validation formateur : `docker compose ps` tout vert + une tâche créée + la PR revue.

”

S4 — Intégration continue avec GitHub Actions

« Qui a cassé `main` ? » — « J'ai oublié de lancer les tests » — « Ça passe chez moi »

Intégration continue (CI, *Continuous Integration*) : à **chaque push**, une machine neutre récupère le code, installe les dépendances, lance le lint, les tests et le build. Le résultat s'affiche sur la PR.

“

*L'humain relit la **logique** ; la machine vérifie ce qui est **mécanique**.*

”

Le vocabulaire de GitHub Actions

Terme	Définition	Exemple TaskFlow
Workflow	Un fichier YAML dans <code>.github/workflows/</code> : la recette complète	<code>ci.yml</code> , <code>release.yml</code>
Événement	Ce qui déclenche le workflow	un <code>push</code> , une <code>pull_request</code>
Job	Un groupe d'étapes exécutées sur une même machine	<code>front</code> , <code>api</code>
Step (étape)	Une commande (<code>run:</code>) ou une action (<code>uses:</code>)	<code>npm test</code>
Action	Une brique réutilisable, publiée par d'autres	<code>actions/checkout@v4</code>
Runner	La machine qui exécute un job	<code>ubuntu-latest</code> (fournie par GitHub)

YAML (*YAML Ain't Markup Language*) : format texte de configuration, basé sur l'indentation. Le même que `compose.yaml`.

Anatomie d'un workflow

Événement : pull_request sur main
| déclenche



Workflow .github/workflows/ci.yml

Job « front »

```
runs-on: ubuntu-latest
step 1  checkout du code
step 2  installer Node 20
step 3  npm ci
step 4  npm run lint
```

machine neuve n° 1

Job « api »

```
runs-on: ubuntu-latest
step 1  checkout du code
step 2  installer Node 20
step 3  npm ci
step 4  npm test
```

machine neuve n° 2

- Les jobs tournent **en parallèle**, chacun sur **sa propre machine**, qui est détruite à la fin
- Les steps d'un job s'enchaînent **dans l'ordre** ; si un step échoue, les suivants sont sautés

Le runner : l'ouvrier qui exécute la recette

L'analogie

- Le **workflow** est une recette de cuisine écrite dans le dépôt
- Le **runner** est l'ouvrier qui la lit et l'exécute, étape par étape
- Il commence chaque job **sur un plan de travail propre** : rien de la fois précédente

Deux sortes de runners

- **Hébergé par GitHub** (`ubuntu-latest`) : machine virtuelle neuve à chaque job, gratuite dans des limites généreuses — **notre choix aujourd'hui**
- **Auto-hébergé** (*self-hosted*) : votre propre machine qui exécute les jobs — **demain**, pour atteindre le serveur du lab

“ Conséquence : le runner ne connaît **que** ce que le workflow lui fait installer. Si ça passe sur votre poste et pas en CI, c'est souvent une dépendance oubliée. ”

YAML en 5 règles

```
name: CI                                     # 1. clé: valeur (un espace APRÈS les deux-points)
on:
  push:                                     # 2. l'indentation (2 espaces) crée la hiérarchie
    branches: [main]                       # 3. liste sur une ligne : [a, b]
  jobs:
    test:
      steps:
        - run: npm ci                       # 4. liste sur plusieurs lignes : un tiret par élément
        - run: |                             # 5. texte sur plusieurs lignes : barre verticale
            npm run lint
            npm test
```

Erreur classique

Pourquoi

Une **tabulation** au lieu d'espaces

Interdit en YAML

`name:CI` (sans espace)

Lu comme le texte `name:CI`, pas comme une clé

Un niveau d'indentation décalé

La clé est rattachée au mauvais parent

Les déclencheurs (`on:`)

Déclencheur	Quand ?	Usage TaskFlow
<code>push</code>	À chaque push (filtrable : <code>branches: [main]</code>)	Publier les images après un merge
<code>pull_request</code>	À l'ouverture et à chaque nouveau commit d'une PR	Vérifier avant le merge
<code>workflow_dispatch</code>	Bouton <i>Run workflow</i> dans l'onglet Actions	Relancer à la main, rollback (J4)
<code>schedule</code>	Selon une expression cron (heure UTC)	Tâche nocturne : audit des dépendances

```
on:
  pull_request:          # toutes les PR
  push:                  # seulement les push sur main
    branches: [main]
  schedule:
    - cron: "0 3 * * 1"  # minute heure jour mois jour-semaine → lundi 3h00 UTC
```

“ *cron : format standard Unix pour planifier une tâche ; UTC = temps universel (1 à 2 h de moins qu'à Paris).* ”

Premier workflow : `hello.yml`

```
name: Hello                                # nom affiché dans l'onglet Actions
on:
  push:                                    # à chaque push, toutes branches
  workflow_dispatch:                       # + bouton manuel
jobs:
  dire-bonjour:                            # identifiant du job (libre)
    runs-on: ubuntu-latest                # runner hébergé par GitHub
    steps:
      - uses: actions/checkout@v4          # récupère le code du dépôt sur le runner
      - name: Saluer                       # nom lisible du step
        run: echo "Bonjour depuis ${github.repository} (${github.ref_name})"
      - run: ls -la                        # le code est bien là
```

- Fichier à créer dans `.github/workflows/hello.yml`, puis commit et push
- Sans `actions/checkout`, le runner est **vide** : votre code n'y est pas

Expressions et contextes `${{ }}`

`${{ ... }}` : GitHub remplace l'expression **avant** d'exécuter le step.

Contexte	Contient	Exemple
<code>github</code>	Infos sur l'événement et le dépôt	<code>github.sha</code> , <code>github.ref_name</code> , <code>github.actor</code>
<code>runner</code>	Infos sur la machine	<code>runner.os</code>
<code>secrets</code>	Les secrets du dépôt (masqués dans les logs)	<code>secrets.GITHUB_TOKEN</code>
<code>vars</code>	Les variables (non secrètes) du dépôt	<code>vars.APP_URL</code>
<code>env</code>	Variables définies dans le workflow	<code>env.NODE_ENV</code>
<code>needs</code>	Les sorties des jobs précédents	<code>needs.build.outputs.tag</code>
<code>inputs</code>	Les paramètres d'un <code>workflow_dispatch</code>	<code>inputs.tag</code> (J4)

`github.sha` = SHA du commit : son empreinte unique de 40 caractères (les 7 premiers suffisent à l'identifier).

L'onglet Actions — Démo `demos/demo-actions-hello/`

```
Actions
├── All workflows
│   └── Hello ────────── Run workflow ▼ (bouton du workflow_dispatch)
│       ├── • Saluer en français    main    #3    il y a 2 min    (vert = succès)
│       └── • test d'une erreur      main    #2    il y a 9 min    (rouge = échec)
│           └── Job dire-bonjour
│               ├── Set up job
│               ├── Run actions/checkout@v4
│               ├── Saluer ← cliquer pour déplier le log
│               └── Complete job
```

- Démo : pousser `hello.yml`, suivre le run en direct, le relancer (*Re-run jobs*), le déclencher à la main
- Puis : ajouter `run: exit 1` → run rouge → lire **quel step** a échoué

À retenir — GitHub Actions : concepts

- **Workflow** (fichier YAML) → déclenché par un **événement** → contient des **jobs** → faits de **steps**
- Chaque job tourne sur un **runner** neuf ; les jobs sont **parallèles** par défaut
- `uses:` = une action réutilisable ; `run:` = une commande shell
- `actions/checkout` d'abord, sinon le runner n'a pas votre code
- YAML : espaces (jamais de tabulation), un espace après `:`, un tiret par élément de liste
- `${{ }}` donne accès aux contextes : `github`, `secrets`, `vars`, `needs`, `inputs`

Bloc 4.2 — Le pipeline de vérification

Lint, tests, build : à chaque PR

Ce que le CI de TaskFlow doit vérifier, et pourquoi

Vérification	Commande	Ce qu'elle attrape
Lint (analyse statique) et formatage	<code>npm run lint</code> , <code>npm run format:check</code>	Variable inutilisée, faute de frappe, style incohérent
Tests unitaires	<code>npm test</code>	Une route qui renvoie le mauvais code HTTP
Tests d'intégration (api)	<code>npm test</code> (avec <code>DATABASE_URL</code>)	Une requête SQL (langage des bases de données) incorrecte, face à un vrai Postgres
Build (front)	<code>npm run build</code>	Un import cassé qui ne se voit qu'à la compilation

Le flux complet que nous construisons aujourd'hui et demain :

```
feature/x —push→ Pull Request → ci.yml : lint + tests + build → vert / rouge
                        | revue d'un pair + CI verte
                        v
                        merge dans main → release.yml : build & push → ghcr.io
                                      | (demain)
                                      v
                                      job deploy → « production » (taskflow-prod) → smoke test /health
```

ci.yml (1/4) — nom et déclencheurs

```
name: CI                                # nom affiché dans l'onglet Actions et sur la PR

on:
  pull_request:                         # vérifier chaque PR, avant le merge
  push:                                # revérifier main après chaque merge
    branches: [main]

jobs:                                   # api et front en parallèle (+ un 3e job, voir plus loin)
```

- Sur `pull_request`, GitHub teste le code **tel qu'il serait après le merge** : on détecte un conflit logique avec `main`
- Sur `push` de `main` : on vérifie que le résultat du merge est toujours vert (badge du README)

ci.yml (2/4) — le job front

```
front:
  runs-on: ubuntu-latest
  defaults:
    run: { working-directory: front }    # chaque `run:` s'exécute dans front/
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4      # installe Node.js sur le runner
      with: { node-version: 20, cache: npm, cache-dependency-path: front/package-lock.json }
    - run: npm ci                      # installation exacte depuis package-lock.json
    - run: npm run lint && npm run format:check  # ESLint puis Prettier (formatage)
    - run: npm test                    # Vitest, fetch mocké
    - run: npm run build                # Vite → dist/
```

- `npm ci` et non `npm install` : installe **exactement** les versions du lock, échoue s'il est désynchronisé
- `cache: npm` : garde le cache npm entre deux runs → installation plus rapide

ci.yml (3/4) — le job `api` et son service Postgres

```
api:
  runs-on: ubuntu-latest
  services:
    postgres:
      image: postgres:16-alpine
      env:
        POSTGRES_USER: taskflow
        POSTGRES_PASSWORD: taskflow
        POSTGRES_DB: taskflow
      ports: ["5432:5432"]
      options: >-
        --health-cmd "pg_isready -U taskflow" --health-interval 5s --health-retries 10
```

conteneurs lancés à côté du job
même version qu'en local et en prod
BDD jetable : pas besoin de secret
accessible via localhost:5432 sur le runner
le job attend que Postgres soit healthy

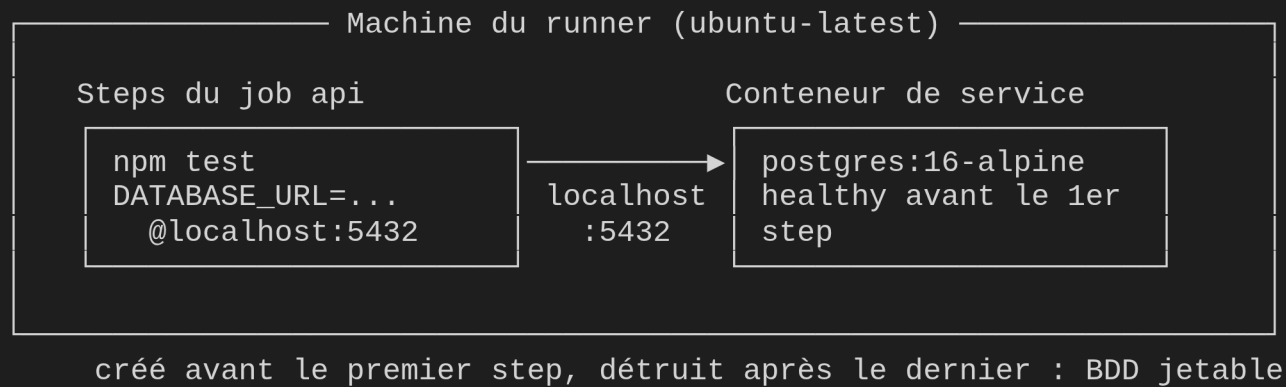
BDD : base de données. Ici elle vit le temps du job, puis elle est détruite avec le runner.

ci.yml (4/4) — les steps du job api

```
env:
  DATABASE_URL: postgres://taskflow:taskflow@localhost:5432/taskflow
defaults:
  run: { working-directory: api }
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-node@v4
    with: { node-version: 20, cache: npm, cache-dependency-path: api/package-lock.json }
  - run: npm ci
  - run: npm run lint
  - run: npm run format:check          # Prettier : formatage respecté ?
  - run: npm test                     # unitaires + intégration (contre le Postgres)
```

- env: au niveau du job : DATABASE_URL est visible par **tous** ses steps
- Rappel : db.int.test.js est **sauté** si DATABASE_URL est absent. Ici il est défini, donc npm test l'exécute aussi

Le conteneur de service, vu de l'intérieur



- Les steps tournent **directement sur la machine** → ils joignent Postgres par `localhost:5432` (grâce à `ports:`)
- Différence avec Compose : là-bas, l'API est **dans un conteneur** et utilise `db:5432`

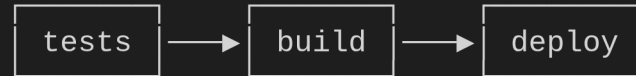
Jobs parallèles, `needs` et matrice

Par défaut : en parallèle



plus rapide, indépendants

Avec `needs` : en série



`build: needs: tests` `deploy: needs: build`

```
api:
  strategy:
    matrix:
      node: [20, 22] # le job est lancé 2 fois, une par version
  steps:
    - uses: actions/setup-node@v4
      with:
        node-version: ${ matrix.node }
```

- Dans `ci.yml`, un 3e job `docker-build` (`needs: [api, front]`) vérifie que les deux Dockerfile se construisent
- La matrice sert à tester plusieurs versions (Node, navigateurs) : pour TaskFlow, c'est un aperçu

Badge de statut et checks requis

```
<!-- README.md : badge qui affiche l'état du dernier run sur main -->
[![CI](https://github.com/<owner>/taskflow/actions/workflows/ci.yml/badge.svg)](https://github.com/<owner>/taskflow/actions)
```

Rendre le CI obligatoire avant le merge :

1. *Settings* → *Branches* (ou *Rules* → *Rulesets*) → règle de protection de `main`
2. Cocher **Require status checks to pass before merging**
3. Ajouter les checks des jobs (noms affichés : `API (lint + tests)`, `Front (lint + tests + build)`) — ils n'apparaissent qu'après un premier run
4. Résultat : bouton *Merge* **grisé** tant que le CI est rouge

“ La PR devient un contrat : revue humaine **et** CI verte. Personne ne peut casser `main` par accident. ”

Lire un log d'échec

```
Run npm test
  FAIL tests/app.test.js > POST /api/tasks > refuse un titre vide
AssertionError: expected 201 to be 400
  > tests/app.test.js:42:25
Test Files  1 failed (1)
Tests      1 failed | 11 passed (12)
Error: Process completed with exit code 1.
```

1. **Quel job, quel step ?** → job `api`, step `npm test` (en rouge dans l'onglet Actions)
2. **Quel test ?** → `POST /api/tasks > refuse un titre vide`
3. **Attendu / obtenu ?** → le test attendait `400`, l'API a répondu `201`
4. **Où ?** → `tests/app.test.js`, ligne 42
5. **Reproduire en local :** `cd api && npm test` → corriger → pousser : le CI se relance seul
Entraînement : `exercices/jour3/` (lire un log d'échec CI, YAML à corriger)

À retenir — Pipeline de vérification

- `checkout` → `setup-node` (avec cache) → `npm ci` → lint → tests → build
- `npm ci` pour une installation **reproductible**
- `services:` lance un Postgres jetable à côté du job, joignable par `localhost`
- Jobs en parallèle par défaut ; `needs:` pour imposer un ordre ; `matrix` pour plusieurs versions
- Badge dans le README ; **checks requis** sur `main` → merge impossible si rouge
- Un échec se lit : job → step → test → attendu / obtenu → ligne → reproduire en local

Bloc 4.3 — Secrets, variables, build d'images

Construire une fois, déployer partout

Pourquoi publier une image ?

- Demain, la « production » ne va **pas** reconstruire TaskFlow : elle va **télécharger** une image toute prête
- L'image construite par le CI est celle qui a été testée : **construite une fois, déployée partout**
- Un **registre** d'images = un « GitHub pour les images Docker ». Nous utilisons **ghcr.io** (GitHub Container Registry) : gratuit, lié au dépôt, pas de compte supplémentaire

```
ghcr.io/<owner>/taskflow-api:a1b2c3d    ← tag = SHA court du commit : immuable, traçable
ghcr.io/<owner>/taskflow-api:latest      ← tag mobile : « la dernière version de main »
└─ registre ─┴─ compte ─┴─ image ─┴─ tag ─┴─
```

- Le tag **SHA** répond à la question « quel commit tourne en production ? » et permet le **rollback**
- Le tag **latest** est pratique pour tester, mais il **change** à chaque merge
- `<owner>` doit être en **minuscules** (ghcr.io refuse les majuscules)

Secrets, variables, `env` : qui va où ?

	<code>secrets.NOM</code>	<code>vars.NOM</code>	<code>env:</code> dans le YAML
Où le définir ?	<i>Settings</i> → <i>Secrets and variables</i> → <i>Actions</i>	Même page, onglet <i>Variables</i>	Dans le fichier du workflow
Visible dans les logs ?	Non, remplacé par <code>***</code>	Oui, en clair	Oui, en clair
Dans Git ?	Jamais	Non (stocké par GitHub)	Oui
Exemples	<code>NOTIF_WEBHOOK</code> (adresse secrète de notification)	<code>APP_URL</code>	<code>DATABASE_URL</code> de test

`GITHUB_TOKEN` : un secret **créé automatiquement** à chaque run, qui expire à la fin du run. Ses droits sont fixés par `permissions:` — on donne le **minimum** nécessaire.

```
permissions:
  contents: read      # lire le code du dépôt
  packages: write     # pousser des images sur ghcr.io
```

release.yml (1/3) — déclencheur et permissions

```
name: Release

on:
  push:
    branches: [main]                # uniquement après un merge sur main

permissions:
  contents: read
  packages: write                  # sans cette ligne : « denied » au push

jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      tag: ${ steps.meta.outputs.tag } # exposé au job deploy de demain
```

- Pas besoin de relancer les tests ici : `main` est protégée, le code y est arrivé **avec une CI verte**
- `outputs:` : le job publie une valeur (le tag) que d'autres jobs pourront lire via `needs`

release.yml (2/3) — calculer le tag et se connecter

```
steps:
  - uses: actions/checkout@v4
  - id: meta                                     # id = pour lire ses sorties plus loin
    run: |
      echo "tag=${GITHUB_SHA:0:7}" >> "$GITHUB_OUTPUT"      # 7 premiers caractères
      echo "owner=${GITHUB_REPOSITORY_OWNER,,}" >> "$GITHUB_OUTPUT" # ,, = en minuscules
  - uses: docker/login-action@v3                 # docker login ghcr.io
    with:
      registry: ghcr.io
      username: ${GITHUB_ACTOR}                  # l'utilisateur qui a déclenché le run
      password: ${GITHUB_TOKEN}
```

- `$GITHUB_OUTPUT` : fichier spécial où un step écrit ses sorties (`nom=valeur`)
- On lit ensuite `steps.meta.outputs.tag` et `steps.meta.outputs.owner`

release.yml (3/3) — construire et pousser les images

```
- uses: docker/build-push-action@v6
  with:
    context: ./api                # dossier contenant le Dockerfile
    push: true                    # pousser vers le registre après le build
    build-args: APP_VERSION=${{ steps.meta.outputs.tag }}
    tags: |
      ghcr.io/${{ steps.meta.outputs.owner }}/taskflow-api:${{ steps.meta.outputs.tag }}
      ghcr.io/${{ steps.meta.outputs.owner }}/taskflow-api:latest
# même step avec context: ./front et l'image taskflow-front
```

- APP_VERSION : dans api/Dockerfile, ARG APP_VERSION=dev puis ENV APP_VERSION=\$APP_VERSION
- Résultat : GET /health répond "version": "a1b2c3d" → on voit quel commit tourne

Vérifier la publication, protéger les secrets

Vérifier : page du dépôt → *Packages* → `taskflow-api` et `taskflow-front`, chacun avec deux tags

```
docker pull ghcr.io/<owner>/taskflow-api:latest # depuis votre poste
```

- Par défaut un package est **privé** : pour le télécharger, `docker login ghcr.io` avec un jeton personnel (PAT, *Personal Access Token*) ayant le droit `read:packages` — ou rendre le package public dans ses réglages

Règles d'or des secrets

- Jamais de `echo ${{ secrets.X }}` : le masquage `***` n'est pas infaillible (un secret encodé n'est plus reconnu)
- Les secrets ne sont **pas** transmis aux PR venant d'un *fork* (copie du dépôt par un inconnu)
- Un secret commité par erreur est **compromis** : le révoquer et en créer un nouveau, même après suppression du commit

À retenir — Secrets, variables, images

- **Construire une fois, déployer partout** : le CI publie l'image, la production la télécharge
- ghcr.io : `ghcr.io/<owner>/<image>:<tag>`, en minuscules
- Deux tags : **SHA court** (traçabilité, rollback) + **latest** (pratique)
- `secrets.*` masqués, `vars.*` en clair, `env:` dans le YAML
- `GITHUB_TOKEN` automatique ; `permissions:` au minimum (`packages: write` pour pousser)
- `docker/login-action` + `docker/build-push-action` ; `APP_VERSION` injectée dans l'image

TP fil rouge — Étape 3 : pipeline CI (1h25)

Énoncé : `tp/fil-rouge-taskflow/etape3-ci.md`

1. Branche `feature/ci` : écrire `.github/workflows/ci.yml` (jobs `front` et `api` avec service Postgres)
2. Ouvrir une PR → suivre le run dans l'onglet Actions → **vert**
3. Ajouter le badge dans le `README.md`
4. Protection de `main` : checks des jobs `api` et `front` **requis**
5. **Casser volontairement un test** (changer un code attendu) → PR **rouge** → bouton *Merge* bloqué → corriger → vert → merge
6. Écrire `release.yml` (build & push) → merge → vérifier *Packages* : deux images, tags SHA et `latest`
7. `docker pull` de votre image depuis votre poste

Antisèche : `ressources/cheatsheet-github-actions.md`

Étape 3 — Si ça ne marche pas

Symptôme	Cause probable	Correction
Le workflow n'apparaît pas dans Actions	Mauvais dossier ou extension	<code>.github/workflows/ci.yml</code> exactement
<code>Invalid workflow file</code>	Erreur YAML (ligne indiquée)	Indentation, espaces, tirets
<code>npm ci</code> : <i>lock file out of sync</i>	<code>package-lock.json</code> absent ou pas à jour	<code>npm install</code> en local, commiter le lock
Les tests d'intégration sont <code>skipped</code>	<code>DATABASE_URL</code> non défini pour le job	Bloc <code>env:</code> au niveau du job <code>api</code>
<code>ECONNREFUSED</code> dans les tests d'intégration	Postgres pas prêt ou port non publié	<code>options: --health-cmd ...</code> et <code>ports:</code>
<code>denied: permission_denied</code> au push	<code>packages: write</code> manquant	Bloc <code>permissions:</code>
<code>repository name must be lowercase</code>	Propriétaire avec majuscules	<code>\${GITHUB_REPOSITORY_OWNER,,}</code>
Checks introuvables dans la protection	Jamais exécutés	Faire tourner le CI une fois d'abord

Récap du jour 3

- **Compose** décrit toute l'application dans `compose.yaml` : `db` + `api` + `front`, réseau, volume, healthchecks
- La **base conteneurisée** : variables `POSTGRES_*` lues au premier démarrage, volume **obligatoire**, `psql` via `exec`
- **GitHub Actions** : workflow → événement → jobs (parallèles, runners neufs) → steps
- `ci.yml` : lint, formatage, tests (dont intégration avec un Postgres de service), build — **requis** sur `main`
- `release.yml` : après merge, images publiées sur ghcr.io avec deux tags (SHA + `latest`)

Demain : on monte une « production » sur votre poste, on y déploie TaskFlow à **la main** derrière Nginx en HTTPS... puis on automatise le déploiement.

Quiz du jour 3

1. Quelle différence entre `docker compose down` et `docker compose down -v` ?
2. Dans `compose.yaml`, pourquoi l'API se connecte-t-elle à `db:5432` et pas à `localhost:5432` ?
3. À quoi sert `depends_on` avec `condition: service_healthy` ?
4. Quelle différence entre un **job** et un **step** ? Les jobs s'exécutent-ils en série ou en parallèle par défaut ?
5. Pourquoi publier chaque image avec **deux** tags (SHA et `latest`) ? Quelle permission faut-il pour pousser sur `ghcr.io` ?

Quiz du jour 3 — Réponses

1. `down` supprime conteneurs et réseau mais **garde le volume** ; `down -v` supprime **aussi le volume** : les données sont perdues
2. Dans un conteneur, `localhost` désigne **le conteneur lui-même** ; `db` est résolu par le DNS du réseau Compose vers le conteneur Postgres
3. À démarrer l'API **seulement quand Postgres est prêt** (healthcheck vert), pas simplement lancé
4. Un job = un groupe de steps sur **une machine** ; un step = **une commande ou une action**. Les jobs sont **parallèles** par défaut (`needs:` pour les ordonner)
5. Le SHA est **immuable** (traçabilité : quel commit tourne, rollback possible) ; `latest` est **pratique** mais bouge. Permission : `packages: write`

Glossaire du jour 3

Terme	Définition
Docker Compose	Outil qui lance une application multi-conteneurs décrite dans <code>compose.yaml</code>
Service	Un conteneur décrit dans <code>compose.yaml</code> (<code>db</code> , <code>api</code> , <code>front</code>) ; son nom sert d'adresse sur le réseau
Healthcheck	Test répété qui indique si un conteneur est réellement prêt (<code>healthy</code>)
Volume nommé	Espace de stockage géré par Docker, qui survit à la suppression du conteneur
DNS	<i>Domain Name System</i> : annuaire qui traduit un nom (<code>db</code>) en adresse réseau
CI	<i>Continuous Integration</i> , intégration continue : vérifier automatiquement chaque modification
Workflow / job / step	Fichier d'automatisation / groupe d'étapes sur une machine / une commande ou action
Runner	Machine qui exécute un job (hébergée par GitHub ou auto-hébergée)
YAML	Format texte de configuration fondé sur l'indentation
Service container	Conteneur (ex. Postgres) lancé à côté d'un job pour les tests
Registre / ghcr.io	Serveur qui stocke les images Docker / celui de GitHub
GITHUB_TOKEN	Jeton temporaire créé à chaque run, droits fixés par <code>permissions:</code>
SHA	Empreinte unique d'un commit (ex. <code>a1b2c3d</code> en version courte)

À demain !

Ressources du jour

`tp/fil-rouge-taskflow/etape2-compose.md` · `tp/fil-rouge-taskflow/etape3-ci.md`
`exercices/jour3/` · `ressources/cheatsheet-docker.md` · `ressources/cheatsheet-github-actions.md`
`docs.docker.com/compose` · `docs.github.com/actions`