

# Contents

|                                                                                       |          |
|---------------------------------------------------------------------------------------|----------|
| <b>Étape 3 : pipeline d'intégration continue</b>                                      | <b>1</b> |
| Prérequis . . . . .                                                                   | 1        |
| Étape 3.0 : lancer en local ce que le pipeline lancera (10 min) . . . . .             | 2        |
| Étape 3.1 : premier job api (15 min) . . . . .                                        | 2        |
| Étape 3.2 : une vraie base PostgreSQL pour les tests d'intégration (10 min) . . . . . | 4        |
| Étape 3.3 : jobs front et docker-build (10 min) . . . . .                             | 5        |
| Étape 3.4 : le badge et le merge (5 min) . . . . .                                    | 7        |
| Étape 3.5 : rendre les checks obligatoires (5 min) . . . . .                          | 7        |
| Étape 3.6 : casser un test volontairement (15 min) . . . . .                          | 8        |
| Étape 3.7 : release.yml, construire et publier les images (20 min) . . . . .          | 9        |
| Étape 3.8 : vérifier les images et les télécharger (10 min) . . . . .                 | 11       |
| Je valide avec le formateur . . . . .                                                 | 12       |
| Questions de réflexion . . . . .                                                      | 12       |
| Pour aller plus loin . . . . .                                                        | 12       |

## Étape 3 : pipeline d'intégration continue

**Quand** : jour 3, après-midi | **Durée** : ~1h30 **Objectif** : à chaque Pull Request et à chaque push sur main, GitHub Actions vérifie automatiquement TaskFlow (ci.yml : lint, formatage, tests dont les tests d'intégration avec un vrai PostgreSQL, build). Ces vérifications deviennent **obligatoires** pour fusionner. Après chaque merge, un second workflow (release.yml) construit les images Docker et les publie sur **ghcr.io**.

### Prérequis

- L'étape 2 est fusionnée dans main.
- Slides du jour 3, blocs 4.1 à 4.3 ; antisèche [../..ressources/cheatsheet-github-actions.md](#).
- Extension VS Code *GitHub Actions* (github.vscode-github-actions) : elle souligne les erreurs dans les workflows.

### Vocabulaire

| Terme                                                             | Signification                                                                                                |
|-------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <b>CI</b> ( <i>Continuous Integration</i> , intégration continue) | Vérifier automatiquement chaque changement, sur une machine neutre                                           |
| <b>Workflow</b>                                                   | Un fichier YAML dans .github/workflows/ ; il contient des jobs                                               |
| <b>Job</b>                                                        | Une suite de steps exécutée sur un <b>runner</b> (une machine) ; les jobs tournent en parallèle sauf needs : |
| <b>Step</b>                                                       | Une commande (run:) ou une action réutilisable (uses:)                                                       |

| Terme                                               | Signification                                                                                               |
|-----------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <b>Service container</b>                            | Un conteneur (ici PostgreSQL) que GitHub démarre à côté d'un job, comme le service db de Compose            |
| <b>Check</b>                                        | Le résultat (vert ou rouge) d'un job, affiché sur la PR                                                     |
| <b>ghcr.io</b> ( <i>GitHub Container Registry</i> ) | Le registre d'images Docker de GitHub : <code>ghcr.io/&lt;propriétaire&gt;/&lt;image&gt;:&lt;tag&gt;</code> |
| <b>SHA court</b>                                    | Les 7 premiers caractères de l'identifiant d'un commit (par exemple 4e7a1c2) : il sert de tag aux images    |
| <b>GITHUB_TOKEN</b>                                 | Un jeton créé automatiquement pour chaque exécution ; ses droits se règlent avec permissions:               |

### Étape 3.0 : lancer en local ce que le pipeline lancera (10 min)

```
cd ~/taskflow
git switch main
git pull
git switch -c feature/ci
cd api && npm run lint && npm run format:check && npm test && cd ..
cd front && npm run lint && npm run format:check && npm test && npm run build && cd ..
```

**Résultat attendu** : chaque commande se termine sans erreur : ESLint: No issues found ou aucune sortie pour le lint, All matched files use Prettier code style!, Tests 27 passed | 10 skipped (37) pour l'API, Tests 7 passed (7) et ✓ built in ... pour le front.

**Si ça ne marche pas** - `format:check` signale des fichiers : `npm run format` les reformate, puis committez le résultat. - `npm run lint` signale une erreur : lisez le fichier et la ligne indiqués ; corrigez avant d'écrire le pipeline.

**Pourquoi ?** Le pipeline ne fait **rien de magique** : il lance les mêmes commandes que vous, mais à chaque fois, sur une machine neuve, sans oubli. Si une commande échoue chez vous, elle échouera chez GitHub.

### Étape 3.1 : premier job api (15 min)

Créez le dossier et le fichier `.github/workflows/ci.yml` :

```
mkdir -p .github/workflows
code .github/workflows/ci.yml
```

Contenu de départ :

```

# ci.yml : Intégration Continue (CI)
# À chaque Pull Request et à chaque push sur main : lint, tests, build.

# Nom affiché dans l'onglet « Actions »
name: CI

# QUAND le workflow démarre
on:
  pull_request:          # à l'ouverture et à chaque mise à jour d'une PR
  push:                  # et quand du code arrive sur main
    branches: [main]

# Moindre privilège : ce workflow peut seulement LIRE le dépôt.
permissions:
  contents: read

# Deux pushes rapprochés sur la même branche : on annule le run devenu inutile.
concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true

jobs:
  # ----- Job 1 : l'API (lint + tests) -----
  api:
    name: API (lint + tests)
    runs-on: ubuntu-latest      # machine virtuelle neuve fournie par GitHub

    # Toutes les commandes « run: » de ce job s'exécutent dans le dossier api/
    defaults:
      run:
        working-directory: api

    steps:
      # Récupère le code du dépôt sur la machine du job
      - uses: actions/checkout@v4

      # Installe Node 20 et met en cache les paquets npm
      - uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: npm
          cache-dependency-path: api/package-lock.json

      - name: Installer les dépendances
        run: npm ci

      - name: Lint (ESLint)
        run: npm run lint

```

- **name:** Formatage (Prettier)  
**run:** npm run format:check
- **name:** Tests (unitaires + intégration)  
**run:** npm test

Poussez et ouvrez une PR (sans la fusionner) :

```
git add .github/workflows/ci.yml
git commit -m "ci: ajoute le job api (lint et tests)"
git push -u origin feature/ci
```

Ouvrez la PR ci: pipeline d'intégration continue sur GitHub, puis l'onglet **Actions** (ou le lien *Details* à côté du check, en bas de la PR).

**Résultat attendu** : un run **CI** démarre ; le job API (lint + tests) devient vert en une minute environ. Dans le step *Tests (unitaires + intégration)*, vous lisez Tests 27 passed | 10 skipped (37) : les tests d'intégration sont encore ignorés.

**Si ça ne marche pas** - Aucun run n'apparaît : le fichier doit être exactement .github/workflows/ci.yml (avec le point devant github). - Invalid workflow file avec un numéro de ligne : erreur d'indentation YAML (espaces uniquement, 2 par niveau). - npm ci : *lock file out of sync* : package-lock.json n'est pas à jour. npm install en local, commitez le lock. - Dependencies lock file is not found : cache-dependency-path doit être api/package-lock.json.

**Pourquoi ?** on: pull\_request fait tourner le pipeline **avant** le merge : l'erreur est attrapée sur la branche, pas sur main. actions/checkout est indispensable : la machine du job est vide au départ.

## Étape 3.2 : une vraie base PostgreSQL pour les tests d'intégration (10 min)

Dans ci.yml, dans le job api, ajoutez les blocs services: et env: **entre** runs-on: et defaults: :

```
# Un « service container » : GitHub démarre PostgreSQL à côté du job,
# exactement comme le service « db » de compose.yml.
services:
  postgres:
    image: postgres:16-alpine
    env:
      POSTGRES_USER: taskflow
      POSTGRES_PASSWORD: taskflow
      POSTGRES_DB: taskflow
    ports:
      - 5432:5432 # accessible sur localhost:5432 depuis le job
# Les steps ne démarrent qu'une fois la base prête
options: >-
  --health-cmd "pg_isready -U taskflow"
  --health-interval 5s
```

```
--health-timeout 3s
--health-retries 10
```

```
# DATABASE_URL défini : le test d'intégration (db.int.test.js) n'est plus ignoré
env:
  DATABASE_URL: postgres://taskflow:taskflow@localhost:5432/taskflow
```

```
git add .github/workflows/ci.yml
git commit -m "ci: lance les tests d'intégration avec un PostgreSQL de service"
git push
```

**Résultat attendu** : un nouveau run démarre tout seul (la PR a été mise à jour). Le job affiche d'abord un step *Initialize containers*, puis les tests passent : Tests 37 passed (37). Plus aucun test ignoré.

**Si ça ne marche pas** - Toujours 10 skipped : le bloc env: n'est pas au niveau du **job** (même indentation que services: et steps:). - ECONNREFUSED 127.0.0.1:5432 : ports: manque, ou les options de healthcheck sont absentes (les tests démarrent avant que la base soit prête).

**Pourquoi ?** Ici, la base est **jetable** et ne contient que des données de test : les tests d'intégration (qui vident la table) peuvent tourner sans risque. Dans le job, l'hôte est localhost car le port du service est publié sur la machine du job.

---

### Étape 3.3 : jobs front et docker-build (10 min)

Ajoutez, **sous** le job api (même indentation que api:, donc 2 espaces) :

```
# ----- Job 2 : le front (lint + tests + build) -----
# Pas de « needs: » : api et front tournent EN PARALLÈLE.
front:
  name: Front (lint + tests + build)
  runs-on: ubuntu-latest

  defaults:
    run:
      working-directory: front

  steps:
    - uses: actions/checkout@v4

    - uses: actions/setup-node@v4
      with:
        node-version: 20
        cache: npm
        cache-dependency-path: front/package-lock.json

    - name: Installer les dépendances
      run: npm ci
```

```

- name: Lint (ESLint)
  run: npm run lint

- name: Formatage (Prettier)
  run: npm run format:check

- name: Tests (Vitest)
  run: npm test

# Le site compile-t-il vraiment ? (erreur d'import, etc.)
- name: Build
  run: npm run build

# ----- Job 3 : les Dockerfile se construisent-ils ? -----
# On construit les images SANS les publier (push: false) : simple contrôle.
docker-build:
  name: Build des images Docker
  runs-on: ubuntu-latest
  needs: [api, front]          # attend que les deux jobs ci-dessus soient verts
  steps:
    - uses: actions/checkout@v4
    - uses: docker/setup-buildx-action@v3

- name: Build de l'image API
  uses: docker/build-push-action@v6
  with:
    context: ./api
    push: false
    cache-from: type=gha,scope=api
    cache-to: type=gha,mode=max,scope=api

- name: Build de l'image front
  uses: docker/build-push-action@v6
  with:
    context: ./front
    push: false
    cache-from: type=gha,scope=front
    cache-to: type=gha,mode=max,scope=front

```

Vérifiez votre fichier complet : il doit être identique (aux commentaires et à l'ordre des blocs près) à [solution/.github/workflows/ci.yml](https://github.com/solution/.github/workflows/ci.yml).

```

git add .github/workflows/ci.yml
git commit -m "ci: ajoute les jobs front et docker-build"
git push

```

**Résultat attendu** : dans le run, le graphe montre API (lint + tests) et Front (lint + tests + build) côte à côte, puis une flèche vers Build des images Docker. Les trois jobs sont verts.

**Si ça ne marche pas** - docker-build est gris (*skipped*) : un des deux jobs précédents a échoué ; c'est le rôle de needs. - The workflow is not valid ... Unexpected value 'front' : le job front est indenté sous api au lieu d'être à côté.

**Pourquoi ?** Les jobs sans needs tournent en même temps : le pipeline est plus rapide. Le build Docker vérifie qu'un changement de code n'a pas cassé les Dockerfile, ce que les tests ne voient pas.

---

### Étape 3.4 : le badge et le merge (5 min)

Ajoutez en **première ligne** du README.md (remplacez <vous> par votre pseudo GitHub) :

```
[![CI](https://github.com/<vous>/taskflow/actions/workflows/ci.yml/badge.svg)](https://github.com/<vous>/taskflow/actions/workflows/ci.yml)
```

```
git add README.md
git commit -m "docs: ajoute le badge de la CI"
git push
```

Quand le run est vert, fusionnez la PR (**Merge pull request**, puis **Delete branch**), et mettez main à jour en local :

```
git switch main
git pull
```

**Résultat attendu** : un run **CI** démarre aussi sur main (déclencheur push). La page d'accueil du dépôt affiche le badge **CI passing** en vert.

**Si ça ne marche pas** - Le badge affiche une image cassée : le chemin doit contenir **votre** pseudo et ci.yml. Il apparaît après le premier run sur main.

---

### Étape 3.5 : rendre les checks obligatoires (5 min)

**Settings > Branches > règle main > Edit** :

- ☒ *Require status checks to pass before merging*
- Dans le champ de recherche, ajoutez **API (lint + tests)** et **Front (lint + tests + build)** (et, si vous le souhaitez, **Build des images Docker**).
- **Save changes.**

**Résultat attendu** : les checks choisis apparaissent sous *Status checks that are required*.

**Si ça ne marche pas** - Les checks ne sont pas proposés dans la recherche : GitHub ne propose que les checks qui ont **déjà tourné** dans la dernière semaine. Vérifiez que le run de l'étape 3.4 a bien eu lieu, puis tapez le nom exact du job (le name:).

**Pourquoi ?** Un check rouge visible mais non bloquant finit toujours par être ignoré « juste cette fois ». Rendu obligatoire, il devient une **règle** : rien n'arrive sur main sans lint et tests verts.

---

### Étape 3.6 : casser un test volontairement (15 min)

On vérifie que le garde-fou fonctionne vraiment.

```
git switch -c test/casser-un-test
```

Dans `api/tests/app.test.js`, dans le test « **crée une tâche et répond 201** » (vers la ligne 36), remplacez :

```
expect(response.status).toBe(201);
```

par :

```
expect(response.status).toBe(200);
```

Vérifiez en local que le test échoue, puis poussez quand même :

```
cd api && npm test; cd ..
git commit -am "test: casse volontairement un test"
git push -u origin test/casser-un-test
```

#### Résultat attendu en local :

```
FAIL tests/app.test.js > API TaskFlow > POST /api/tasks > crée une tâche et répond
AssertionError: expected 201 to be 200 // Object.is equality
```

Ouvrez la PR. **Résultat attendu sur GitHub :**

- le check API (lint + tests) est **rouge**, Build des images Docker est sauté ;
- en bas de la PR : *Required statuses must pass before merging*, le bouton **Merge** est grisé ;
- dans le log du step *Tests*, vous retrouvez la même ligne `AssertionError: expected 201 to be 200`.

Corrigez (remettez `toBe(201)`), puis :

```
git commit -am "test: rétablit le code attendu 201"
git push
```

**Résultat attendu :** le run repasse au vert, le bouton **Merge** redevient actif. Comme la branche ne change plus rien au final, **fermez** la PR sans la fusionner (*Close pull request*) et supprimez la branche, puis `git switch main`.

**Si ça ne marche pas** - Le bouton **Merge** reste actif malgré la croix rouge : la protection de l'étape 3.5 n'est pas enregistrée, ou vous avez coché *Allow specified actors to bypass*. Vérifiez la règle. - `git commit -am` ne commite rien : vous n'avez pas enregistré le fichier dans VS Code.

**Pourquoi ?** Vous venez de voir le pipeline **refuser** un changement cassé, sans qu'aucun humain ait eu à le remarquer. C'est exactement la situation de « La Fromagerie » (TP 0), réglée.

### Étape 3.7 : release.yml, construire et publier les images (20 min)

Après chaque merge sur main, on veut des images **prêtes à déployer**, publiées sur ghcr.io avec deux tags : le SHA court du commit (pour savoir exactement quelle version tourne, et revenir en arrière) et latest.

```
git switch main
git pull
git switch -c feature/release
code .github/workflows/release.yml
```

Contenu :

```
# release.yml : Livraison = construire les images et les publier sur ghcr.io.
# (Le job de déploiement sera ajouté à l'étape 4.)
```

```
name: Release
```

```
on:
```

```
  push:
```

```
    branches: [main]
```

```
  workflow_dispatch: # bouton « Run workflow » dans l'onglet Actions
```

```
# Un seul run à la fois ; on n'interrompt jamais un run en cours.
```

```
concurrency:
```

```
  group: release
```

```
  cancel-in-progress: false
```

```
env:
```

```
  REGISTRY: ghcr.io
```

```
jobs:
```

```
  build:
```

```
    name: Construire et publier les images
```

```
    runs-on: ubuntu-latest
```

```
# Droits du GITHUB_TOKEN : lire le code + ÉCRIRE dans le registre de paquets.
```

```
permissions:
```

```
  contents: read
```

```
  packages: write
```

```
# Valeur transmise aux jobs suivants (étape 4)
```

```
outputs:
```

```
  tag: ${ steps.meta.outputs.tag }
```

```
steps:
```

```
  - uses: actions/checkout@v4
```

```
# Le tag (7 premiers caractères du SHA du commit) et le propriétaire EN MINUSCULE
```

```
# ghcr.io refuse les majuscules (${VAR,,} = passage en minuscules en bash).
```

```
  - name: Calculer le tag et le nom des images
```

```

id: meta
run: |
  echo "tag=${GITHUB_SHA::7}" >> "$GITHUB_OUTPUT"
  echo "owner=${GITHUB_REPOSITORY_OWNER,,}" >> "$GITHUB_OUTPUT"

# Connexion au registre avec le jeton automatique : aucun mot de passe à gérer
- name: Connexion à ghcr.io
  uses: docker/login-action@v3
  with:
    registry: ${ env.REGISTRY }
    username: ${ github.actor }
    password: ${ secrets.GITHUB_TOKEN }

- uses: docker/setup-buildx-action@v3

- name: Build et push de l'API
  uses: docker/build-push-action@v6
  with:
    context: ./api
    push: true
    tags: |
      ${ env.REGISTRY }/${ steps.meta.outputs.owner }/taskflow-api:${ step
      ${ env.REGISTRY }/${ steps.meta.outputs.owner }/taskflow-api:latest
    # Injecte la version dans l'image (affichée par /health)
    build-args: |
      APP_VERSION=${ steps.meta.outputs.tag }
    cache-from: type=gha,scope=api
    cache-to: type=gha,mode=max,scope=api

- name: Build et push du front
  uses: docker/build-push-action@v6
  with:
    context: ./front
    push: true
    tags: |
      ${ env.REGISTRY }/${ steps.meta.outputs.owner }/taskflow-front:${ step
      ${ env.REGISTRY }/${ steps.meta.outputs.owner }/taskflow-front:latest
    cache-from: type=gha,scope=front
    cache-to: type=gha,mode=max,scope=front

```

```

git add .github/workflows/release.yml
git commit -m "ci: publie les images sur ghcr.io après chaque merge"
git push -u origin feature/release

```

Ouvrez la PR : seul **CI** tourne (release.yml ne se déclenche que sur main). Quand il est vert, fusionnez.

**Résultat attendu** : dans l'onglet **Actions**, un run **Release** démarre sur main et devient vert en deux à trois minutes. Dans le résumé du step *Calculer le tag...*, vous pouvez retrouver le SHA court du commit de merge.

**Si ça ne marche pas** - denied: permission\_denied: write\_package ou installation not allowed to Create organization package : le bloc permissions: avec packages: write manque ou est mal indenté. - repository name must be lowercase: votre pseudo contient des majuscules et l'étape meta n'est pas utilisée dans tags:. Vérifiez `{{ steps.meta.outputs.owner }}`. - Le run **Release** ne démarre pas : il ne se déclenche qu'au **push sur main**, donc après le merge.

**Pourquoi ?** « Construire une fois, déployer partout » : l'image publiée est **celle qui a été testée**, et c'est elle que la production téléchargera demain. Le SHA court relie chaque image à un commit précis : on sait ce qui tourne, et on peut revenir à n'importe quelle version.

---

### Étape 3.8 : vérifier les images et les télécharger (10 min)

Sur GitHub, ouvrez votre profil > onglet **Packages** (ou, sur la page du dépôt, la rubrique *Packages* de la colonne de droite). **Résultat attendu** : deux paquets, taskflow-api et taskflow-front, chacun avec deux tags : le SHA court et latest.

Un paquet peut être **privé** (souvent le cas par défaut) ou **public** : regardez la mention affichée sur sa page. Choisissez **une** des deux options :

**Option A : rendre les paquets publics** (le plus simple pour la suite). Pour chacun des deux paquets : page du paquet > **Package settings** > *Danger Zone* > **Change visibility** > *Public*. Plus besoin de vous connecter pour les télécharger.

**Option B : garder les paquets privés** et se connecter avec un jeton personnel. Créez un **PAT** (*Personal Access Token*, jeton d'accès personnel) : photo de profil > **Settings** > **Developer settings** > **Personal access tokens** > **Tokens (classic)** > **Generate new token (classic)** ; nom ghcr lecture taskflow, expiration 7 jours, cochez **uniquement read:packages**. Copiez le jeton (il ne sera plus jamais affiché), puis :

```
read -rs GHCR_PAT          # collez le jeton puis Entrée (rien ne s'affiche : c'est v
echo "$GHCR_PAT" | docker login ghcr.io -u <vous> --password-stdin
```

**Résultat attendu** : Login Succeeded. Gardez ce jeton dans un gestionnaire de mots de passe : il resservira demain.

Dans les deux cas, téléchargez l'image et vérifiez la version qu'elle contient (votre pseudo **en minuscules**) :

```
docker pull ghcr.io/<vous-en-minuscules>/taskflow-api:latest
docker run --rm ghcr.io/<vous-en-minuscules>/taskflow-api:latest node -e "console.log
```

**Résultat attendu** : le pull télécharge l'image, puis la commande affiche le **SHA court** du commit de merge (par exemple 4e7a1c2), et non plus dev.

**Si ça ne marche pas** - unauthorized ou denied au pull : le paquet est privé et vous n'êtes pas connecté (option B), ou le jeton n'a pas le droit read:packages. - manifest unknown : le tag n'existe pas, ou le nom contient des majuscules. Copiez la commande docker pull affichée sur la page du paquet. - Error saving credentials lors du docker login sous WSL (le Linux intégré à Windows) : relancez Docker Desktop, ou vérifiez le fichier ~/.docker/config.json.

**Pourquoi ?** `read -rs` évite que le jeton apparaisse à l'écran et dans l'historique du terminal, et `--password-stdin` évite qu'il apparaisse dans la liste des processus. Un jeton est un mot de passe : droits minimaux, durée courte.

---

## Je valide avec le formateur

- ☐ `ci.yml` contient les jobs `api` (avec le service PostgreSQL), `front` et `docker-build` ; le run affiche 37 `passed` pour l'API.
- ☐ Le badge **CI passing** est sur la page d'accueil du dépôt.
- ☐ Les checks API (`lint + tests`) et Front (`lint + tests + build`) sont **requis** sur main.
- ☐ J'ai vu une PR **rouge** dont le bouton *Merge* était bloqué, puis repasser au vert.
- ☐ Le run **Release** est vert ; les paquets `taskflow-api` et `taskflow-front` ont les tags SHA court et latest.
- ☐ `docker run ... node -e "console.log(process.env.APP_VERSION)"` affiche le SHA court.

## Questions de réflexion

1. Pourquoi `release.yml` ne relance-t-il pas les tests ? Qu'est-ce qui garantit que le code de main les a passés ?
2. Que se passerait-il sans le bloc `permissions:` du job `build` ? Et si on donnait `packages:` `write` à `ci.yml` aussi ?
3. Pourquoi deux tags (SHA court **et** latest) ? Lequel faudra-t-il utiliser pour revenir en arrière demain ?
4. Le test d'intégration tourne contre `localhost:5432` dans le pipeline, mais contre `db:5432` dans Compose. Pourquoi ?
5. Un collègue vous propose de supprimer le step *Formatage (Prettier)* « parce qu'il échoue souvent ». Que lui répondez-vous ?

## Pour aller plus loin

- Lancez **Release** à la main : onglet **Actions** > *Release* > **Run workflow**. Quel tag obtient l'image ?
- Exercices : [../../exercices/jour3/](#) (YAML à corriger, lire un log d'échec de CI, choisir le bon déclencheur).