LAB 4 : Version Control, Build Systems, and Automated Testing

Explication détaillée des étapes

Étape 1 — Git local : “Branches, commits et merge”

Objectif de l’étape :
Mettre en place un mini-repo Git en local pour maîtriser les actions de base attendues en DevOps : initialiser un dépôt, travailler sur une branche, committer proprement, puis fusionner et vérifier l’historique.

Ce que j’ai fait :

1. Création du dépôt Git local
    - J’ai créé un dossier de travail (lab4/git-playground) puis initialisé un dépôt avec :
    - git init
    - J’ai ajouté un fichier README.md, puis je l’ai indexé et commité :
    - git add README.md
    - git commit -m "init: add README"
2. Travail sur une branche dédiée
    - J’ai créé une branche testing pour isoler les changements (bonne pratique : ne pas travailler directement sur la branche principale) :
    - git checkout -b testing
    - J’ai modifié README.md, puis j’ai enregistré ces changements dans l’historique :
    - git add README.md
    - git commit -m "test: update README on testing"
3. Fusion (merge) vers la branche principale
    - Je suis revenu sur la branche principale :
    - git checkout master
    - J’ai fusionné testing dans master :
    - git merge testing
    - Dans mon cas, Git a fait un fast-forward (ça veut dire que master n’avait pas eu de nouveaux commits depuis la création de testing, donc Git avance simplement le pointeur de branche sans créer de commit de merge).
4. Vérification finale (preuve)
    - J’ai vérifié l’historique et l’état des branches avec :
    - git log --oneline --decorate --graph --all

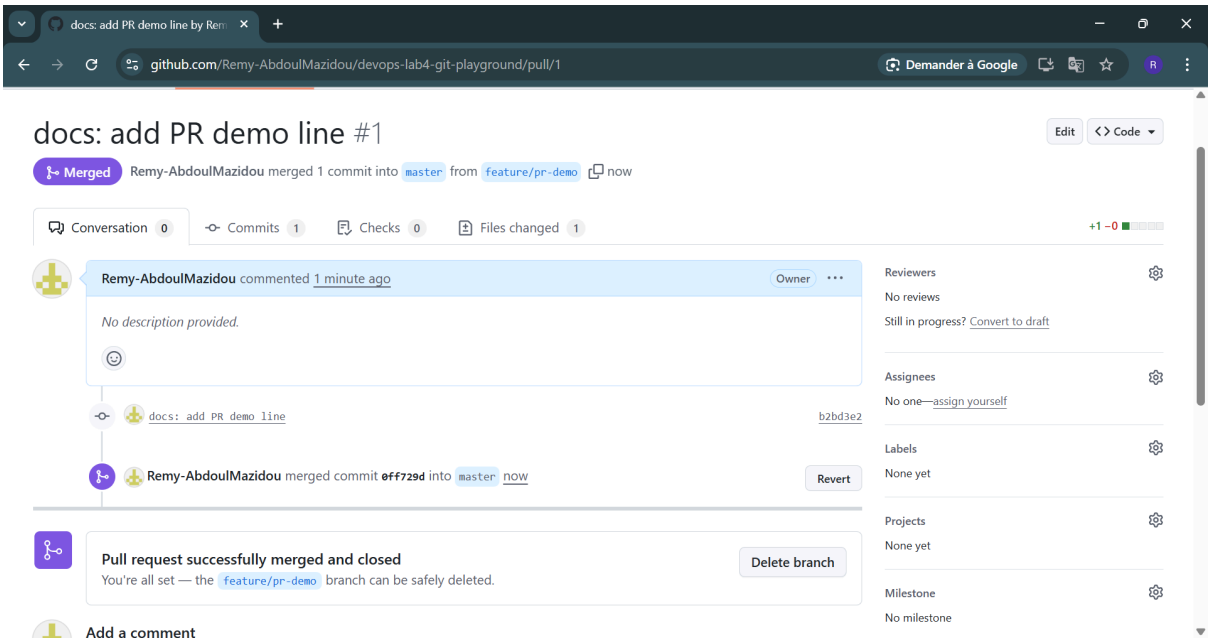
- On voit bien les 2 commits et le fait que master et testing pointent sur le dernier commit après fusion.

Ce que j'ai appris :

- Une branche sert à développer une fonctionnalité sans impacter directement la branche principale.
- Un commit est une sauvegarde versionnée (un "checkpoint") avec un message clair.
- Un merge intègre le travail d'une branche dans une autre ; le fast-forward arrive quand l'historique est linéaire (pas de divergence).
- git log --graph est une preuve rapide que je sais lire l'historique d'un projet.

Étape 2 — GitHub : Remote, Push et Pull Request (workflow d'équipe)

Lien du repository github :
https://github.com/Remy-AbdoulMazidou/devops-lab4-git-playground



Description du screen :
J'ai créé une branche dédiée (feature/pr-demo), poussé mes changements sur GitHub, ouvert une Pull Request vers master, puis je l'ai fusionnée après vérification qu'il n'y avait pas de conflits. Ça reproduit un workflow d'équipe (PR avant d'intégrer sur la branche principale).
Après le merge de la PR sur GitHub, j'ai fait git pull sur master pour synchroniser mon dépôt local avec origin/master

Objectif de l'étape:
Passer d'un dépôt Git local à un vrai workflow "comme en entreprise" : collaborer via GitHub, pousser le code sur un remote, puis intégrer les changements via Pull Request au lieu de modifier directement la branche principale.

Ce que j'ai fait :

1. Création d'un repository GitHub vide
- J'ai créé un repo GitHub sans README/.gitignore/license pour éviter les conflits avec le repo local.
2. Connexion du dépôt local au remote
- J'ai lié mon repo local à GitHub via :
- git remote add origin https://github.com/<user>/<repo>.git
3. Premier push de la branche principale
- J'ai envoyé ma branche principale sur GitHub et mis en place le tracking :
- git push -u origin master
- Résultat : master local suit origin/master.
4. Création d'une branche feature + push
- J'ai créé une branche dédiée pour simuler un changement en équipe :
- git checkout -b feature/pr-demo
- J'ai modifié README.md, puis commit :
- git add README.md
- git commit -m "docs: add PR demo line"
- J'ai poussé la branche sur GitHub :
- git push -u origin feature/pr-demo
5. Création et merge d'une Pull Request
- Sur GitHub, j'ai ouvert une PR : feature/pr-demo → master.
- GitHub a indiqué No conflicts (merge automatique possible).
- J'ai mergé la PR (statut Merged) puis j'aurais pu supprimer la branche.
6. Synchronisation du dépôt local après merge
- J'ai récupéré les changements mergés sur GitHub dans mon master local :
- git checkout master
- git pull
- Ici, Git a fait un fast-forward, donc mon master local a avancé sur le commit mergé.

Pourquoi on fait ça :

- Travailler en équipe : on évite les modifications directes sur la branche principale.
- Sécuriser la qualité : la PR sert à relire/valider les changements avant intégration.
- Historique clair : on garde des changements "atomiques" (1 feature = 1 branche/PR).
- Traçabilité : GitHub conserve le contexte (discussion, commits, diff, statut merged).

Pièges / erreurs fréquentes :

- Comparer master…master sur GitHub : ça affiche “no changes” (c’était ton cas au début).
  → Il faut comparer base: master avec compare: feature/pr-demo.
- Créer le repo GitHub avec un README auto : ça peut provoquer des conflits au premier push/pull.
  → Repo vide recommandé quand on a déjà un repo local.
- Oublier de faire git pull après un merge GitHub : le local n’est plus aligné avec le remote.
  → Toujours git pull sur la branche principale après un merge.

Étape 3 — Sample App Node + NPM start (mise en place d’une base exécutable)

Objectif de l’étape:
Créer une application minimale qui tourne localement et mettre en place un point d’entrée standard (npm start). L’idée, côté DevOps, c’est de ne pas lancer “à la main” des commandes différentes selon les gens, mais d’avoir une commande unique et reproductible.

Ce que j’ai fait :

1. Initialisation du projet Node
- J’ai créé un nouveau dossier sample-app et initialisé un projet NPM :
- npm init -y
  → Cela génère un package.json (nom, version, scripts, etc.)
2. Création d’un serveur HTTP minimal
- J’ai créé server.js qui démarre un serveur sur le port 3000 et renvoie Hello, World!.
3. Test de l’application en local
- J’ai lancé le serveur directement avec :
- node server.js
  → attendu : Server running on http://localhost:3000
- Dans un autre terminal, j’ai vérifié que l’endpoint répond :
- curl http://localhost:3000
  → attendu : StatusCode 200 et Hello, World! dans le contenu
- Puis j’ai arrêté le serveur avec Ctrl + C.
4. Ajout d’un script NPM standard
- J’ai ajouté un script de démarrage dans package.json :
- npm pkg set scripts.start="node server.js"
- J’ai vérifié le démarrage “standard” avec :
- npm start
  → attendu : Server running on http://localhost:3000

- Puis arrêt avec Ctrl + C.

Pourquoi on fait ça :

- Reproductibilité : npm start est une commande standard que tout le monde utilise (dev, CI, doc, etc.).
- Automatisation : ça prépare la suite du lab (Docker + tests + CI/CD) car l'exécution devient prévisible.
- Moins de friction en équipe : on évite "chez moi ça marche mais pas chez toi" à cause de commandes différentes.

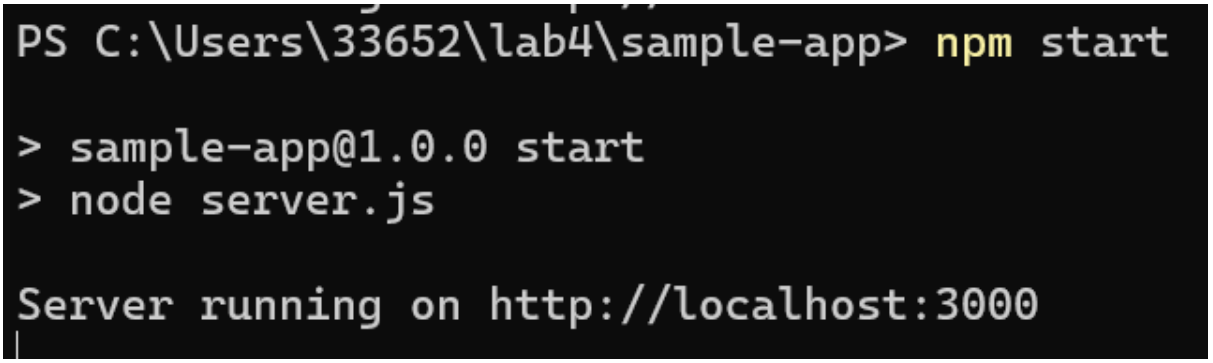
Pièges / erreurs fréquentes :

- Port déjà utilisé (3000) : si un autre programme occupe le port, le serveur ne démarre pas.
  → Solution : arrêter l'autre process ou changer de port.
- Oublier de sauvegarder server.js : node server.js ne trouve pas le fichier ou exécute une vieille version.
- Lancer curl dans le même terminal que le serveur : impossible car le terminal est occupé par node server.js.
  → Solution : ouvrir un 2e terminal.
- Sur PowerShell, curl affiche un objet détaillé (pas juste la chaîne).
  → Normal : l'info importante est StatusCode=200 et Content=Hello, World!.

Si tu veux, pour cette étape, tu peux aussi garder 1 preuve (screenshot) :

- sortie npm start + un curl http://localhost:3000 qui montre StatusCode 200 et Content : Hello, World!.

Sortie npm start :



Curl sur http://localhost:3000

Étape 4 — Dockeriser l'app : build + run

Objectif de l'étape :
Conteneurisé l'application pour qu'elle puisse tourner de façon reproductible sur n'importe quelle machine. En DevOps, Docker permet de figer l'environnement d'exécution (version Node, dépendances, démarrage) et d'éviter le "ça marche sur mon PC".

Ce que j'ai fait :

1. J'ai créé un Dockerfile minimal basé sur une image Node stable :

- FROM node:22-alpine
- WORKDIR /app
- COPY server.js ./
- EXPOSE 3000
- CMD ["node","server.js"]

2. J'ai construit l'image Docker :

- docker build -t sample-app:1.0.0 .

3. J'ai lancé un conteneur en exposant le port :

- docker run --rm -p 3000:3000 sample-app:1.0.0

4. J'ai vérifié que l'endpoint répond bien :

- curl http://localhost:3000 → StatusCode 200 et Hello, World!

Pourquoi on fait ça :

- Même environnement partout (même Node, même commande de démarrage).
- Déploiement plus simple : une fois l'image buildée, il suffit de "run".
- Base nécessaire pour la suite (tests, CI/CD, déploiements).

Pièges rencontrés / classiques :

- Docker Desktop pas démarré → erreur de connexion au moteur Docker (pipe dockerDesktopLinuxEngine introuvable).
  → Solution : lancer Docker Desktop et vérifier avec docker version (Client + Server).

- Port déjà utilisé (3000) → bind: address already in use.
  → Solution : arrêter l'ancien process/conteneur ou changer le mapping de ports.

- Vérification : sur PowerShell, curl affiche un objet, mais il faut vérifier StatusCode et Content.

Étape 5 — Passer à Express + gérer les dépendances + debug d'un "piège" (port) + mise à jour Docker

Piège typique DevOps :

- "Mon code est bon mais je ne vois pas le bon résultat" → souvent un problème de port / process (conflit, proxy, container, service déjà en écoute).

- Ici, netstat -ano | findstr :3000 a montré que Docker Desktop écoutait sur 3000 (PID 13000), donc tes requêtes ne touchaient pas ton serveur.

J'avais une réponse incohérente sur /name/:name alors que le code était correct. J'ai diagnostiqué un conflit de port : netstat -ano | findstr :3000 montrait que le port 3000 était déjà écouté (PID 13000 = Docker). J'ai temporairement déplacé l'app sur 3001 pour valider le comportement."

Dans server.js :

```
app.listen(3001, () => {
  console.log("Server running on http://localhost:3001");
});
```

Changer aussi dans le dockerfile

Objectif de l'étape :
Transformer l'application Node minimale en une vraie petite API avec Express (routes), apprendre à gérer les dépendances NPM, et rendre le tout compatible avec Docker (installation des deps dans l'image). Cette étape prépare directement la suite : tests (Jest/Supertest) et CI/CD.

5.1 Ajout d'Express et création des routes

Ce que j'ai fait :

1. Installation d'Express
- npm install express
  → Express apparaît dans dependencies et NPM génère/actualise package-lock.json.
2. Refonte de server.js avec Express
- J'ai remplacé le serveur HTTP simple par une app Express avec deux routes :
  - GET / → Hello, World!
  - GET /name/:name → Hello, <name>!
- J'ai lancé l'app avec npm start et j'ai validé les endpoints avec curl.

Pourquoi c'est important :

- Express est un standard pour exposer des endpoints HTTP (API).
- Ça force à comprendre la notion de route et de paramètre d'URL (:name).
- On prépare les tests : Supertest testera précisément ces routes.

5.2 Piège rencontré : incohérence / mauvais résultat malgré un code correct (conflit de port)

Le symptôme :

- Même si /name/Remy était implémenté, l'endpoint renvoyait encore Hello, World!.

Diagnostic (méthode DevOps) :

- J'ai vérifié le type de réponse HTTP :
  - Express renvoie souvent Content-Type: text/html; charset=utf-8
  - Mon endpoint renvoyait text/plain → indice que ce n'était pas le bon serveur qui répondait.
- J'ai cherché qui écoutait sur le port :
  - netstat -ano | findstr :3000
- J'ai identifié le process :
  - tasklist /FI "PID eq <pid>"
- Résultat : le port 3000 était pris par Docker Desktop (com.docker.backend.exe), donc mes requêtes n'arrivaient pas sur mon serveur Express.

Correction :

- Pour ne pas perdre de temps, j'ai changé le port de l'app en 3001 et retesté :
  - curl http://localhost:3001/name/Remy → Hello, Remy! (OK)

Ce que je retiens / ce que je peux expliquer :

- Un bug apparent peut venir d'un problème de contexte (mauvais process, mauvais port), pas du code.
- La bonne démarche : observer, isoler, identifier le process, corriger, puis re-tester.

5.3 Mise à jour de Docker pour Express + dépendances

Problème avec l'ancien Dockerfile :

- Il copiait seulement server.js et n'installait pas express → l'image ne peut pas tourner correctement en conteneur.

Ce que j'ai fait :

1. Dockerfile mis à jour
- J'ai copié package.json + package-lock.json puis installé les dépendances avec :
    - RUN npm ci --omit=dev
- Puis j'ai copié server.js, exposé le port 3001, et gardé le démarrage node server.js.

2. Build et run
- Build :
    - docker build -t sample-app:1.0.1 .
- Run :
    - docker run --rm -p 3001:3001 sample-app:1.0.1
- Vérification :
    - curl http://localhost:3001/name/Remy → Hello, Remy!

Pourquoi npm ci --omit=dev ?

- npm ci est reproductible (basé sur le lockfile).
- --omit=dev évite d'embarquer les outils de dev/test dans une image "prod" (bonne pratique).
  (Les tests seront plutôt faits côté dev/CI.)

Pièges fréquents de cette étape :

- Oublier de lancer curl dans un 2e terminal (le serveur bloque le terminal).
- Ne pas avoir package-lock.json → npm ci ou COPY package-lock.json peut échouer.
- Conflit de ports (3000/3001) : un process Docker/Node peut déjà être en écoute.
- Sur PowerShell, curl = Invoke-WebRequest et peut afficher un objet : il faut regarder StatusCode et Content.

Étape 6 — Tests applicatifs : Jest + Supertest + refactor app/server + “bug volontaire”

Objectif de l’étape :
Mettre en place des tests automatisés pour valider le comportement de l’application et détecter les régressions. Le lab insiste sur le fait qu’un test doit être utile : il doit échouer quand on casse le comportement, puis repasser au vert après correction.

6.1 Installation et configuration des tests

- J’ai installé les outils de test en devDependencies :
    - npm install --save-dev jest supertest
- J’ai remplacé le script test par une commande standard :
    - npm pkg set scripts.test="jest --verbose"
- J’ai validé que npm test exécute bien Jest.

Pourquoi devDependencies ?
Parce que les tests sont nécessaires en développement/CI, pas forcément dans une image “production”.

6.2 Refactor : séparer app.js et server.js

Pour tester proprement avec Supertest, il faut pouvoir importer l’app Express sans démarrer un serveur.

- app.js contient l’app Express et exporte app :
    - routes / et /name/:name
    - module.exports = app
- server.js est uniquement le lanceur :
    - const app = require("./app");
    - app.listen(PORT, ...)

Pourquoi c’est important ?
Supertest peut faire des requêtes directement sur app sans écouter un vrai port, donc les tests sont plus simples, plus rapides et plus fiables.

6.3 Écriture des tests (Supertest)

J’ai créé app.test.js et écrit 2 tests :

- GET / doit renvoyer :
    - status 200
    - texte Hello, World!
- GET /name/Remy doit renvoyer :

    - status 200
    - texte Hello, Remy!

Commande de validation :

- npm test → tests PASS

6.4 Piège / démonstration : bug volontaire

Le lab demande d'introduire un bug pour prouver que les tests servent réellement.

- J'ai remplacé Hello, World! par Hello, Mars! dans la route /.
- npm test est passé en FAIL avec un diff clair :
    - Expected: Hello, World!
    - Received: Hello, Mars!

Ensuite, j'ai corrigé le bug (remis Hello, World!) :

- npm test repasse PASS

Ce que ça prouve :
Les tests détectent une régression et protègent le comportement attendu.

Pièges fréquents :

- Oublier module.exports = app → Supertest ne peut pas importer l'app.
- Mettre listen() dans app.js → tests plus compliqués (serveur réel, ports, conflits).
- Comparaison stricte des textes (World vs Mars) : c'est volontaire pour démontrer les régressions.

Étape 7 — Docker "final" après refactor (app.js + server.js) + gestion d'un conflit de port

Objectif de l'étape :
Mettre à jour la conteneurisation après le refactor (app.js + server.js) afin que l'application fonctionne dans Docker exactement comme en local. L'idée DevOps : une image Docker doit contenir tout ce qui est nécessaire (code + dépendances) et démarrer de manière reproductible.

7.1 Mise à jour du Dockerfile

Après la séparation app.js / server.js, l'ancien Dockerfile n'était plus adapté (il ne copiait pas forcément tous les fichiers nécessaires).
J'ai donc modifié le Dockerfile pour :

- Copier package.json + package-lock.json
- Installer les dépendances de production
- Copier app.js et server.js
- Exposer le bon port (3001)
- Démarrer avec node server.js

Dockerfile utilisé :

```
FROM node:22-alpine

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm ci --omit=dev

COPY app.js server.js ./

EXPOSE 3001

CMD ["node", "server.js"]
```

Pourquoi npm ci --omit=dev ?

- npm ci est reproductible (basé sur le lockfile).
- --omit=dev permet de ne pas embarquer les dépendances de dev (tests) dans une image "prod".

7.2 Build et exécution

J'ai construit une nouvelle version d'image :

- docker build -t sample-app:1.0.2 .

Puis j'ai tenté de lancer le conteneur :

- docker run --rm -p 3001:3001 sample-app:1.0.2

7.3 Piège rencontré : “port is already allocated”

J’ai eu l’erreur :

- Bind for 0.0.0.0:3001 failed: port is already allocated

Interprétation :
Le port 3001 était déjà utilisé (par un serveur Node lancé en local, ou un autre conteneur Docker encore actif).

Résolution :

- J’ai libéré le port (en arrêtant le process/containeur qui l’utilisait) puis j’ai relancé la commande.
- Le conteneur a ensuite démarré correctement et l’application était accessible.

Ce que je retiens :

- Un problème de conteneurisation peut venir du réseau/ports, pas du code.
- Le diagnostic typique se fait avec des outils comme netstat (pour voir qui écoute sur un port) et en arrêtant le process/containeur concerné.

7.4 Validation

J’ai validé le bon fonctionnement en appelant l’API depuis l’hôte :

- curl http://localhost:3001/name/Remy → Hello, Remy!

Pièges fréquents de cette étape :

- Oublier de copier app.js dans l’image → crash au démarrage.
- Oublier package-lock.json → npm ci échoue.
- Port déjà pris (3001 ou 3000) → erreur “port is already allocated”.
- Confondre “tests” vs “runtime” : les tests (Jest/Supertest) restent côté dev/CI, pas nécessairement dans l’image “prod” si on fait --omit=dev.

Étape 8

- dependencies : ce qui est nécessaire pour exécuter l'app en production (ici express).
- devDependencies : outils uniquement utiles en dev/CI (ici jest, supertest).
- Dans Docker on a utilisé npm ci --omit=dev pour installer uniquement les dependencies, donc l'image est plus légère et plus "prod-like".

Express est en dependencies car il est nécessaire à l'exécution. Jest/Supertest sont en devDependencies car ils servent au test. Dans l'image Docker, npm ci --omit=dev installe seulement les dependencies, donc les outils de test ne sont pas embarqués.

Étape 9 — Mise au propre "livrable Lab 4" : repo final GitHub + versioning propre

Création d'un nouveau repo :
https://github.com/Remy-AbdoulMazidou/devops-lab4

Objectif de l'étape :
Avoir un rendu propre et professionnel : un repository GitHub clair, avec le code du Lab 4 versionné correctement (notamment sample-app, Docker, tests). L'idée DevOps : un projet doit être reproductible et partageable sans fichiers inutiles (ex : node_modules) et avec un historique lisible.

9.1 Création d'un repo GitHub final dédié

- J'ai créé un nouveau repository GitHub vide (sans README / .gitignore / license) :
  - devops-lab4
- Pourquoi vide ? Pour éviter les conflits dès le premier push (pas de README auto généré côté GitHub).

9.2 Création d'un dossier "propre" en local et copie du code

Pour éviter un mélange avec les essais précédents (git-playground), j'ai créé un dossier "clean" :

- lab4-final

Puis j’ai copié mon projet sample-app dedans :

- Copy-Item -Recurse -Force sample-app lab4-final\

But :

Avoir une structure propre du rendu avant d’initialiser Git.

9.3 Initialisation Git + .gitignore

1. J’ai initialisé un nouveau dépôt Git dans lab4-final :
- git init
2. J’ai créé un .gitignore pour ne pas versionner des fichiers inutiles / lourds :
- node_modules/ (à ne jamais push)
- .env (potentiellement sensible)
- logs / fichiers OS

Sur PowerShell, j’ai eu un petit piège de syntaxe avec le “bloc multi-lignes”, donc je suis passé sur une commande simple :

- Création du .gitignore :
    - "node_modules/n.envn.DS_Storennpm-debug.log*" | Out-File -Encoding utf8 .gitignore`
- Vérification :
    - type .gitignore

Piège rencontré :

La version “heredoc” @" ... "@ a été mal interprétée à cause du copier-coller, donc la version “une ligne” est plus robuste sur Windows.

9.4 Ajout des fichiers, commit initial et push sur GitHub

1. J’ai ajouté les fichiers au staging :
- git add .

J’ai eu des warnings Windows :

- LF will be replaced by CRLF
  → Ce sont des avertissements de fin de ligne (normal sur Windows). Ce n’est pas bloquant.
2. J’ai créé un commit initial clair :
- git commit -m "init: lab4 sample-app with docker and tests"
3. J’ai lié le dépôt local au repo GitHub final :
- git remote add origin https://github.com/Remy-AbdoulMazidou/devops-lab4.git
4. J’ai poussé la branche principale et mis en place le tracking :
- git push -u origin master

Ce que je peux expliquer :

- Pourquoi un repo dédié “clean” : rendu plus lisible, pas de pollution (meilleur pour l’évaluation).
- Pourquoi .gitignore est obligatoire : éviter node_modules, fichiers sensibles (.env) et parasites.
- Pourquoi le commit message est propre : trace claire de ce qui est livré.
- Pourquoi git push -u : relie master local à origin/master pour simplifier les pulls/push suivants.

Commandes utiles :

- git init
- git add .
- git commit -m "..."
- git remote add origin <url>
- git push -u origin master
- type .gitignore (vérif rapide)
- (diagnostic windows) warnings LF/CRLF = normal

Étape 10 — Documentation “prof-proof” dans le repo (/docs) + versioning

Objectif de l’étape :
Créer une documentation courte et claire directement dans le repository pour :

- garder une trace écrite de ce qui a été fait dans le Lab 4,
- pouvoir se relire avant l’évaluation,
- faciliter la transmission de connaissances à l’équipe,
- montrer au professeur une démarche DevOps : “je code + je documente + je versione”.

10.1 Création du dossier de documentation

- J’ai ajouté un dossier dédié :
    - mkdir docs

Pourquoi ?
Séparer la doc du code rend le repo plus lisible : on sait où trouver les notes rapidement.

10.2 Création d'un fichier de notes Lab 4

- J'ai créé un fichier docs/LAB4_NOTES.md contenant :
    - les objectifs du Lab 4 (Git/GitHub, Node/Express, Docker, tests),
    - les commandes clés (npm start, npm test, docker build, docker run),
    - les pièges rencontrés et comment les diagnostiquer (ports, Docker Desktop, etc.).

Commande utilisée (PowerShell) :

- création du fichier via Out-File (texte multi-ligne avec \n`).

Puis j'ai vérifié le contenu :

- type docs\LAB4_NOTES.md

Pourquoi cette étape est utile ?

- Une doc simple + concrète, avec commandes et erreurs fréquentes, permet de gagner du temps quand on reprend le projet.
- En équipe, c'est exactement ce qu'on veut : que quelqu'un d'autre puisse comprendre rapidement ce qui a été fait.

10.3 Commit et push de la documentation

Pour que la doc fasse partie du livrable, je l'ai versionnée :

1. Ajout au staging :
- git add docs\LAB4_NOTES.md
2. Commit clair :
- git commit -m "docs: add lab4 notes and pitfalls"
3. Envoi sur GitHub :
- git push

Piège rencontré (normal sur Windows) :

- Warning LF will be replaced by CRLF : c'est lié aux fins de ligne sur Windows.
  → Ce n'est pas bloquant, ça n'empêche pas de versionner ni de rendre le projet.

Ce que je peux expliquer au prof :

- "J'ai versionné la doc comme le code : add → commit → push."
- "La doc contient surtout ce qui est utile : objectifs, commandes, pièges et comment diagnostiquer."

Commandes clés (récap) :

- mkdir docs
- type docs\LAB4_NOTES.md
- git add docs\LAB4_NOTES.md
- git commit -m "docs: ..."
- git push

Étape 11 — 8 questions + réponses courtes

1. Pourquoi PR au lieu de push direct sur master ?
   "Pour forcer la relecture et éviter d'intégrer des changements non validés. La PR garde une trace (diff, discussion, commits) et sécurise la branche principale."

2. Pourquoi séparer app.js et server.js ?
   "Pour tester l'app Express sans lancer un serveur. app.js exporte l'app, server.js fait juste le listen. Supertest peut importer app directement."

3. Pourquoi Jest + Supertest ?
   "Jest exécute les tests, Supertest simule des requêtes HTTP sur l'app Express et vérifie status + body. Ça valide le comportement API."

4. Pourquoi le "bug volontaire" Hello Mars ?
   "Pour démontrer que les tests sont utiles : ils deviennent rouges si on casse le comportement attendu, puis repassent verts après correction."

5. dependencies vs devDependencies ?
   "dependencies = runtime (Express). devDependencies = outils dev/CI (Jest, Supertest)."

6. Pourquoi npm ci au lieu de npm install dans Docker ?
   "npm ci est reproductible car basé sur package-lock.json. C'est la commande standard en CI."

7. Pourquoi --omit=dev dans l'image Docker ?
   "Pour ne pas embarquer les outils de test dans une image prod, réduire la taille et la surface d'attaque."

8. Pièges rencontrés / debug
   "Conflit de ports : diagnostic netstat -ano | findstr :3000/3001. Docker Desktop pas lancé : vérifier avec docker version. Port already allocated : arrêter process/containers."

Étape 12–13 — Bonus "Infra Tests" avec OpenTofu (tofu test) : installation + démo + pièges

But global :

Montrer que je sais faire autre chose que "déployer au hasard".
En DevOps, on veut automatiser la validation de l'infrastructure, comme on automatise les tests applicatifs.
C'est exactement le rôle de tofu test : exécuter des tests d'infra avec des *assertions* (status code, contenu, etc.) et donner un résultat PASS/FAIL exploitable en CI.

12 - Mise en place d'OpenTofu (pourquoi, où, comment)

12.1 À quoi sert OpenTofu ?

OpenTofu est un outil "Infrastructure as Code" (IaC), très proche de Terraform :

- on décrit une infra avec des fichiers .tf (ressources, modules, providers),
- on peut appliquer (apply) pour créer/configurer,
- et surtout, OpenTofu propose tofu test pour tester l'infra de façon automatisée.

- "IaC = infra versionnée + reproductible"
- "tofu test = tests d'infra automatisés, comme des tests unitaires mais côté infra"

12.2 Pourquoi on n'a pas fait ça sur Windows directement ?

Quand j'ai essayé :

- tofu version → commande non reconnue
  Donc OpenTofu n'était pas installé sur Windows.

Plutôt que de perdre du temps avec des installs/variables d'environnement sur Windows, j'ai choisi une solution fiable DevOps :
→ Utiliser WSL Ubuntu (Linux) qui est l'environnement standard pour bash, outils DevOps et scripts.

12.3 Vérification WSL

Dans PowerShell :

- wsl -l -v
  → confirme que Ubuntu (WSL2) est disponible.

Puis ouverture d'un shell Ubuntu directement dans le dossier du projet :

- wsl -d Ubuntu --cd "/mnt/c/Users/33652/lab4/lab4-final"

12.4 Installation d'OpenTofu sur WSL

Dans Ubuntu :

- apt update
- tentative apt install -y opentofu → package introuvable (piège classique : pas dans les repos apt par défaut)

Solution : installer via snap (simple et stable) :

- snap install opentofu --classic

Vérification :

- tofu version
  → OpenTofu installé et fonctionnel.

Pièges de cette étape :

- Confondre "install ok" avec "outil utilisable" : toujours vérifier avec tofu version.
- Certains packages ne sont pas dispo via apt selon Ubuntu → snap est une alternative.

13 - Mise en place d’un test infra “comme dans le lab” (démo)

13.1 Pourquoi a-t-on créé un dossier infra-test-demo ?

Dans un Lab complet, la partie infra (ex : lambda-sample) existe déjà et expose souvent un output du type api_endpoint.

Ici, dans mon repo final Lab 4, je n’avais pas encore d’infra AWS/Lambda prête (normal, ça vient souvent dans d’autres labs).
Mais je voulais comprendre et démontrer le mécanisme tofu test sans dépendre d’AWS.

Donc j’ai créé un mini projet autonome infra-test-demo pour :

- reproduire la logique du lab (module test-endpoint + asserts),
- avoir un exemple concret “PASS/FAIL”,
- pouvoir l’expliquer au prof et à mon équipe.

Commandes :

- mkdir -p infra-test-demo
- cd infra-test-demo

13.2 Création du module test-endpoint (le cœur du test)

Objectif :

Créer un module réutilisable qui “teste un endpoint” en récupérant :

- le status code
- le body (contenu)

Structure :

- modules/test-endpoint/main.tf

Création du dossier :

- mkdir -p modules/test-endpoint

Contenu clé (principe) :

- on déclare un provider HTTP
- on passe une variable url
- on fait un data "http" (appel HTTP)
- on expose des outputs status_code et response_body

Pourquoi un module ?

- en DevOps, on réutilise : ce module pourrait servir à tester n’importe quel endpoint (API, Lambda, etc.)

- dans le lab, l'idée est similaire : un module "test-endpoint" qu'on branche sur la vraie infra

13.3 Création d'un projet qui appelle le module

À la racine de infra-test-demo, j'ai créé main.tf :

- module "endpoint" { source="./modules/test-endpoint"; url="https://example.com" }

Pourquoi https://example.com ?

- c'est stable, simple, et ça permet de prouver que tofu test fonctionne
- ensuite, dans le vrai lab, on remplace par un output api_endpoint

13.4 Écriture du test deploy.tftest.hcl

C'est ici qu'on fait le "PASS/FAIL".

Fichier :

- deploy.tftest.hcl

Principe :

- un run "apply" : applique la config
- un run "assert_endpoint" : vérifie avec un assert que output.status_code == 200

Pourquoi des runs ?

- tofu test exécute des scénarios :
  - tu peux déployer
  - tester
  - et (selon le cas) détruire
- ça simule ce que ferait une CI : "déployer temporairement → tester → valider"

13.5 Piège rencontré : "Module not installed"

Quand j'ai lancé directement :

- tofu test
  j'ai eu :
- Module not installed → "Run tofu init"

Pourquoi ?

- comme Terraform/OpenTofu, il faut initialiser le projet pour :
  - télécharger les providers
  - préparer les modules
  - créer le dossier .terraform

Solution :

- tofu init

Ensuite :

- tofu test → PASS

Résultat :

- Success! 2 passed, 0 failed.

Pièges classiques de cette étape :

- oublier tofu init avant tofu test
- provider pas téléchargé / bloc required_providers absent
- assertions trop strictes (ex : body exact) si l'endpoint change
- dépendance à un endpoint instable (mieux de tester un endpoint contrôlé)

Ce que je peux dire :

J'ai voulu maîtriser tofu test : l'idée est de tester l'infra comme on teste le code.
J'ai installé OpenTofu sur WSL, puis j'ai créé un module réutilisable test-endpoint basé sur le provider HTTP.
Avec un fichier deploy.tftest.hcl, j'ai mis une assertion simple (status code 200).
J'ai rencontré le piège 'module not installed', corrigé via tofu init, puis tofu test est passé en vert.
Dans le vrai lab, on branche ce module sur api_endpoint (Lambda) et on ajoute aussi un assert sur le body (Hello, World!)."

Vision globale : pourquoi ce Lab 4 existe

Le Lab 4 sert à poser les **fondations DevOps** avant d'aller vers des déploiements plus "réels" (cloud, infra, CI/CD…).
L'objectif n'est pas de faire une grosse application, mais de maîtriser les **bonnes pratiques** qui reviennent partout :

- travailler avec **Git** (historique, branches, merges)
- collaborer via **GitHub** (pull requests, review, intégration)
- rendre une app **reproductible** avec **Docker**
- sécuriser le comportement avec des **tests** (Jest/Supertest)
- (bonus) comprendre le principe des **tests d'infra** avec **OpenTofu** (tofu test)

En gros : ce lab transforme "je code sur mon PC" en "je peux livrer / partager / tester / reproduire".

Étape 1 — Git local : commits, branches, merge

Ce qu'on a fait

- Initialisation d'un repo Git local, création d'un README.md
- Premier commit
- Création d'une branche (ex : testing)
- Ajout d'un changement sur la branche
- Merge dans la branche principale
- Vérification de l'historique (git log --oneline --decorate --graph --all)

À quoi ça sert / pourquoi on le fait

- **Git** garde l'historique et permet de revenir en arrière
- Les **branches** évitent de casser la branche principale pendant qu'on expérimente
- Le **merge** est la méthode standard pour intégrer un travail validé

À quoi ça servira plus tard

Dans les labs suivants / en projet :

- tu travailleras toujours en branches (feature, fix, etc.)
- tu intégreras régulièrement dans une branche stable (main/master)
- tu devras prouver "qui a fait quoi" et avoir un historique propre

Pièges/difficultés

- confusion entre branche courante et branche distante
- ne pas comprendre “fast-forward merge” (merge sans conflit et sans commit de merge, car l’historique est linéaire)

Commandes importantes

- git init
- git status
- git add .
- git commit -m "..."
- git branch <name>
- git switch <name>
- git merge <branch>
- git log --oneline --decorate --graph --all

Étape 2 — GitHub : remote + Pull Request (PR)

Ce qu’on a fait

- Création d’un repo GitHub
- Ajout du remote origin
- Push de la branche principale
- Création d’une branche feature (feature/pr-demo)
- Push de la branche feature
- Création d’une **Pull Request** sur GitHub
- Merge de la PR dans master
- git pull en local pour récupérer le merge

Pourquoi GitHub ?

GitHub sert à la **collaboration** et à la traçabilité :

- PR = espace de review, discussion, validation
- on voit le diff, les commits, qui a mergé, quand
- ça simule le workflow en entreprise

À quoi ça servira plus tard

- la plupart des projets (cours + entreprise) imposent PR + review
- les pipelines CI/CD sont souvent déclenchés sur PR / merge

Pièges/difficultés

- Au début tu étais sur la page “compare” avec base/compare mal choisis → il fallait sélectionner la branche feature/pr-demo en “compare” et master en base.

- oublier de git pull après avoir mergé via GitHub → local pas à jour.

Commandes importantes

- git remote add origin <url>
- git push -u origin master
- git push -u origin feature/pr-demo
- git pull

Étape 3 — App Node : base exécutable + standard npm start

Ce qu'on a fait

- npm init -y → création de package.json
- création de server.js minimal
- test local avec node server.js
- test HTTP avec curl http://localhost:3000
- ajout d'un script start : npm pkg set scripts.start="node server.js"
- vérification via npm start

Pourquoi on le fait

- avoir une appli minimale mais réelle (HTTP)
- standardiser le démarrage : npm start (reproductible, utilisé en CI/CD)

À quoi ça servira plus tard

- tous les labs/projets auront besoin d'un "entrypoint" standard
- la doc + Docker + CI s'appuient sur cette standardisation

Pièges

- sur PowerShell, curl affiche un objet (Invoke-WebRequest) → l'important est StatusCode + Content.

Commandes importantes

- npm init -y
- node server.js
- npm pkg set scripts.start="node server.js"
- npm start
- curl http://localhost:<port>

Étape 4 — Docker : rendre l'app portable et reproductible

Ce qu'on a fait

- création d'un Dockerfile
- build : docker build -t sample-app:1.0.0 .
- run : docker run --rm -p 3000:3000 sample-app:1.0.0
- test via curl

Pourquoi Docker ?

- Docker "emballe" l'app + son environnement (Node, fichiers, dépendances)
- résultat : même comportement sur n'importe quelle machine

À quoi ça servira plus tard

- déploiements sur serveur/cloud
- exécutions CI (runner docker)
- composition de services (docker-compose) si plusieurs composants

Pièges rencontrés

- Docker Desktop pas lancé → erreur de connexion au moteur
    - diagnostic : docker version (il faut voir Client + Server)
- problème de ports (port déjà utilisé)

Commandes importantes

- docker version
- docker build -t <name:tag> .
- docker run --rm -p host:container <image>

Étape 5 — Express + dépendances + "piège port"

Ce qu'on a fait

- installation Express : npm install express
- ajout d'une route paramétrée : /name/:name
- test HTTP avec curl

Pourquoi Express ?

- framework standard pour exposer des routes HTTP (API)
- on apprend la logique d'API (endpoints + paramètres)

Piège majeur rencontré

Tu avais le bon code mais /name/Remy renvoyait encore Hello, World!.
C'était un **conflit de port** : ce n'était pas ton serveur qui répondait.

Diagnostic :

- netstat -ano | findstr :3000 → PID à la fin
- tasklist /FI "PID eq <pid>" → ici com.docker.backend.exe utilisait le port

Solution :

- passer l'app sur 3001 pour éviter le conflit et valider le comportement

À quoi ça servira plus tard

- en DevOps, comprendre que beaucoup de "bugs" viennent de l'environnement (ports, services, containers)
- savoir diagnostiquer rapidement

Commandes importantes

- npm install express
- netstat -ano | findstr :3000
- tasklist /FI "PID eq <pid>"

Étape 6 — Tests : Jest + Supertest + refactor app/server

Ce qu'on a fait

- installation en devDependencies :
  - npm install --save-dev jest supertest
- script test :
  - npm pkg set scripts.test="jest --verbose"
- refactor :
  - app.js exporte l'app Express
  - server.js écoute sur le port (listen)
- tests dans app.test.js :
  - GET / → Hello, World!
  - GET /name/Remy → Hello, Remy!
- démonstration bug volontaire :
  - changer World → Mars → tests FAIL
  - corriger → tests PASS

Pourquoi c'est central

- on prouve que le comportement est contrôlé

- on détecte les régressions avant de déployer
- c'est la base de la CI : un merge doit passer les tests

À quoi ça servira plus tard

- dans le projet final, tu mettras des tests avant CI/CD
- les pipelines feront npm test automatiquement

Pièges

- ne pas séparer app.js et server.js → tests plus compliqués (port, serveurs)
- assertions trop strictes / pas stables (mais ici volontaire pour la démo)

Commandes importantes

- npm test
- npm install --save-dev jest supertest

Étape 7 — Docker "final" (deps + app.js/server.js) + port déjà alloué

Ce qu'on a fait

- Dockerfile mis à jour :
  - copie package.json + package-lock.json
  - npm ci --omit=dev
  - copie app.js + server.js
- build image 1.0.2
- run sur 3001
- gestion d'erreur "port is already allocated"

Pourquoi npm ci --omit=dev ?

- npm ci : reproductible (lockfile)
- --omit=dev : image plus légère (pas de Jest/Supertest en prod)

À quoi ça servira plus tard

- pratiques "prod-like"
- builds plus rapides, images plus propres

Pièges

- oublier de copier app.js
- oublier lockfile → npm ci échoue
- port déjà pris → stopper process/conteneur

Étape 8 — Clarification : dependencies vs devDependencies

Ce qu'on a validé

- express en dependencies (runtime)
- jest/supertest en devDependencies (tests)

Pourquoi c'est important

- différence entre "ce qui fait tourner l'app" et "ce qui sert à développer/tester"
- utile pour Docker/CI et pour expliquer une image prod propre

Étape 9–10 — Repo final propre + Documentation + README

Pourquoi plusieurs repos GitHub au début ?

- devops-lab4-git-playground : sert à démontrer **branches/PR/merge** simplement (exercice GitHub)
- devops-lab4 : repo final propre "livrable" (app + docker + tests + doc + bonus)
  Ça permet d'avoir un rendu clair, sans mélange, et ça fait plus professionnel.

Ce qu'on a fait

- création d'un dossier propre lab4-final
- git init, .gitignore
- push vers devops-lab4
- création de docs/LAB4_NOTES.md
- ajout d'un README.md clair pour exécuter local / tests / docker / tofu test

À quoi ça servira plus tard

- un repo clair = un projet maintenable
- c'est ce qu'on attend dans les autres labs/projet final (doc + commandes)

Bonus — Étape 12–13 OpenTofu : tests infra avec tofu test

Pourquoi Ubuntu/Linux ?

- Les outils DevOps (bash, tofu, scripts) sont plus simples/stables en Linux.
- Sur Windows, tofu n'était pas installé → WSL est une solution standard.

Pourquoi OpenTofu ?

- c'est un outil IaC (Infrastructure as Code) : décrire/tester l'infra de façon reproductible

- tofu test permet d'écrire des tests d'infra avec des assertions (PASS/FAIL), comme pour l'app

Ce qu'on a fait

- installation sur WSL via snap
- création d'une démo infra-test-demo :
    - module test-endpoint (provider HTTP)
    - fichier deploy.tftest.hcl avec assert status_code == 200
- piège : "Module not installed" → tofu init obligatoire
- exécution : tofu test → PASS

Lien avec les autres labs / projet futur

- Dans les labs suivants, l'infra sera "réelle" (AWS / Lambda / API endpoint).
- Le principe sera identique : on récupère un output (ex : api_endpoint) et on écrit un test qui valide :
    - status code
    - body (ex : Hello, World!)
- Ça prépare la logique "déployer → tester → valider", très CI/CD.

Commandes importantes :

- tofu version
- tofu init
- tofu test

Conclusion : ce que le prof veut vérifier

Ce Lab 4 montre que je sais :

1. travailler proprement avec Git (branches/merge)
2. collaborer via GitHub (PR/merge)
3. rendre une app reproductible (npm scripts, Docker)
4. sécuriser le comportement (tests, démonstration de régression)
5. diagnostiquer des problèmes DevOps réels (ports, Docker Desktop, init tofu)
6. comprendre la logique "préparer le terrain" pour les labs suivants (infra/CI/CD/cloud)

Si je dois résumer en 1 phrase :

> Lab 4 = transformer un petit code local en un livrable DevOps partageable, testable et reproductible, prêt à être intégré dans une suite de labs orientée déploiement/infra.

Explication simple du Lab 4 :

L'idée de départ : "ça marche sur mon PC" ne suffit pas

Imagine que tu codes une petite appli sur ton ordinateur et que tout fonctionne chez toi.
En DevOps, ce n'est pas assez : le professeur (ou une équipe) doit pouvoir refaire la même chose facilement, sans dépendre de ton PC, de tes réglages, ou de tes fichiers locaux.

Le but du Lab 4, c'est donc de passer de :

- "J'ai un code qui marche sur mon PC"
  à
- "J'ai un livrable propre, reproductible, testable, et prêt pour le travail en équipe / la CI / le déploiement."

Ce que le Lab 4 nous apprends (en 4 idées claires)

1) Git : organiser ton travail et garder un historique

Un projet évolue : tu modifies, tu testes, tu reviens en arrière parfois.
Avec Git, tu peux :

- garder une trace de chaque étape (les commits),
- essayer des idées sans casser la version stable (les branches),
- revenir en arrière si tu t'es trompé.

En résumé : Git sert à travailler proprement et à ne pas "perdre" ton projet.

2) GitHub + Pull Request : travailler comme en équipe

Dans la vraie vie (et souvent dans les projets de cours), on ne modifie pas la branche principale directement.
On fait une Pull Request pour :

- montrer ce qu'on a changé,
- faire valider avant d'intégrer,
- garder une trace claire de l'intégration.

En résumé : GitHub + PR, c'est la méthode standard pour collaborer et intégrer sans casser le projet.

3) npm + Docker : rendre le projet reproductible partout

Le but est que quelqu'un d'autre puisse faire :

- installer,

- démarrer,
- tester,

sans galérer.

- Avec npm start et npm test, tu donnes des commandes simples et standard.
- Avec Docker, tu mets ton appli “dans une boîte” avec son environnement : si quelqu’un lance ton image Docker, il obtient le même résultat, même sur une autre machine.

En résumé : Docker sert à dire “ça marche partout, pas seulement chez moi”.

4) Tests automatiques : vérifier que tu n’as rien cassé

Quand tu changes du code, tu peux casser une route sans t’en rendre compte.
Les tests (Jest + Supertest) servent à :

- vérifier automatiquement que les routes répondent comme prévu (/, /name/:name),
- détecter une régression (par exemple si tu changes “World” en “Mars”, un test devient rouge).

En résumé : les tests évitent d’intégrer une version cassée et préparent la CI/CD.

Pourquoi OpenTofu ?

OpenTofu : tester l’infrastructure comme on teste le code

OpenTofu sert normalement à décrire/déployer de l’infrastructure (Infrastructure as Code).
Ici, le point important du lab, c’est l’idée suivante :

Après un déploiement, on peut vérifier automatiquement que l’endpoint répond bien (PASS/FAIL).

C’est utile pour les labs suivants (AWS, endpoints, etc.) : tu déploies, puis tu testes automatiquement que ça répond.

Pourquoi Ubuntu / Linux (WSL) ?

Les outils DevOps sont souvent plus simples à utiliser sur Linux (scripts, commandes, outils comme OpenTofu).
Sur Windows, l’outil n’était pas installé, donc on a utilisé WSL Ubuntu, ce qui est une solution courante en DevOps.

À quoi sert le repo “Lab 4” concrètement ?

Le repo sert comme un “kit prêt à l’emploi” :

- quelqu’un clone le repo,

- lit le README,
- sait lancer l'app, lancer les tests, lancer Docker,
- et peut revoir les notes/pièges rencontrés.

En résumé : c'est un livrable propre + ta documentation + un template réutilisable.

Lien avec les autres labs / le projet final

Le Lab 4 est la base méthodologique :

- Git + GitHub PR → travail en équipe pour tous les labs suivants
- Docker → exécution reproductible (utile pour déployer)
- Tests → base de la CI (automatiser la validation)
- Bonus OpenTofu → préparer les labs d'infrastructure/cloud (déployer puis tester)

En résumé : les labs suivants vont faire des choses plus grosses, mais avec les mêmes règles. Lab 4 t'apprend ces règles.

Conclusion

Le Lab 4 sert à apprendre à livrer un projet "DevOps-ready" : versionné proprement, collaboratif (PR), reproductible (Docker), validé automatiquement (tests), et prêt pour la suite (CI/CD et infra).

Présentation des dossiers du Lab 4

Dossier git-playground — Explication simple (comme un mini README)

Rôle (à quoi ça sert)

git-playground est un bac à sable Git/GitHub.
Je l'ai utilisé pour m'entraîner et démontrer le workflow :

- commits
- branches
- merge
- push vers GitHub
- Pull Request (PR) + merge sur GitHub
- git pull pour récupérer en local

Pourquoi l'avoir séparé ?

Parce que c'est plus propre :

- je peux faire des tests Git sans polluer le vrai projet,
- le prof peut voir clairement la partie "workflow GitHub PR" sur un petit exemple.

Ce qu'il y a dedans (contenu typique)

- un README.md très simple
- plusieurs commits
- plusieurs branches (ex: testing, feature/pr-demo)
- un historique lisible (git log --graph --oneline --all)

Comment l'expliquer en 1 phrase

"git-playground est mon dossier d'entraînement Git/GitHub : il montre que je sais créer des branches, faire un merge et utiliser une Pull Request sur GitHub."

Commandes clés associées

- git init, git add, git commit
- git switch -c <branch>
- git merge <branch>
- git remote add origin <url>
- git push -u origin <branch>
- (sur GitHub) Create PR → Merge
- git pull

Dossier sample-app — Explication simple (comme un mini README)

Rôle (à quoi ça sert)

sample-app est la mini application Node/Express du Lab 4.
Elle sert de support pour apprendre :

- démarrage standard avec npm (npm start)
- tests automatiques (npm test avec Jest/Supertest)
- conteneurisation Docker (docker build / docker run)

Pourquoi ce dossier est important ?

Parce que c'est la partie "technique" du lab :

- une petite API HTTP réelle (routes)
- des tests qui détectent les régressions
- Docker pour rendre l'exécution reproductible

Ce qu'il y a dedans (fichiers importants)

- app.js : l'app Express (routes / et /name/:name)
- server.js : démarre le serveur (listen sur 3001)
- app.test.js : tests Jest/Supertest des routes
- package.json : scripts (start, test) + dépendances
- Dockerfile : build et run de l'app dans un conteneur

Comment l'expliquer en 1 phrase

"sample-app est la mini API du Lab 4 : je peux la lancer en local, la tester automatiquement, et la faire tourner dans Docker de manière reproductible."

Commandes clés

Run local

- npm install
- npm start

Tests

- npm test

Docker

- docker build -t sample-app:1.0.2 .
- docker run --rm -p 3001:3001 sample-app:1.0.2

Pièges rencontrés (important à mentionner)

- Conflits de ports (3000/3001) : diagnostic netstat -ano | findstr :3001
- Docker Desktop non démarré : vérifier avec docker version
- Différence dependencies vs devDependencies (Express vs Jest/Supertest)

Dossier lab4-final — Explication simple (comme un mini README)

Rôle (à quoi ça sert)

lab4-final est la version finale propre du Lab 4 : c'est le dossier "livrable" prêt à être rendu et relu.
L'idée est que n'importe qui puisse :

- cloner le repo,
- suivre le README,
- lancer l'app,
- lancer les tests,
- lancer Docker,
- comprendre rapidement les pièges et la logique du lab.

Pourquoi l'avoir créé séparément ?

Pour avoir un rendu clair et professionnel :

- pas mélangé avec les essais/entraînements,
- structure simple,
- documentation intégrée,
- tout ce qu'il faut pour exécuter est au même endroit.

Ce qu'il y a dedans (structure)

- README.md : mode d'emploi (local, tests, docker, bonus tofu)
- sample-app/ : l'application Node/Express + tests + Dockerfile
- docs/ : notes + commandes importantes + pièges rencontrés
- infra-test-demo/ : bonus OpenTofu (démo de tofu init + tofu test)

Comment l'expliquer en 1 phrase

"lab4-final est mon livrable final du Lab 4 : une mini API testée et dockerisée, avec documentation et un bonus OpenTofu pour montrer le principe des tests d'infrastructure."

À quoi ça servira plus tard

- Template réutilisable pour les labs suivants / projet final :
    - même workflow GitHub PR
    - même logique tests avant intégration
    - même approche Docker pour la reproductibilité
    - même idée de tests infra après déploiement (OpenTofu)

Commandes clés (celles qu'on retrouve dans le README)

- Local : npm start
- Tests : npm test
- Docker : docker build ... puis docker run ...
- Bonus : tofu init puis tofu test

Résumer les 3 dossiers

J'ai séparé le Lab 4 en trois parties : git-playground pour démontrer le workflow Git/GitHub et les Pull Requests, sample-app pour la mini API Node/Express avec tests et Docker, et lab4-final qui regroupe la version propre et rendable avec le README, la doc et le bonus OpenTofu.