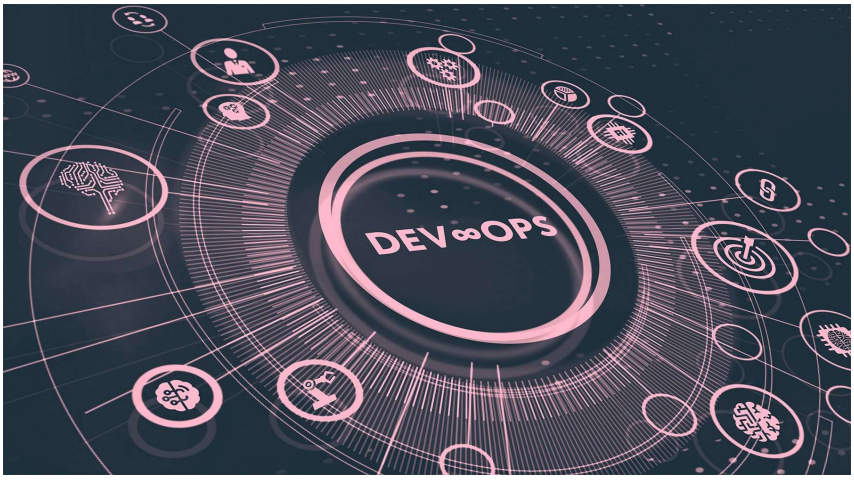


DÉPLOIEMENT D'UNE ARCHITECTURE FULLSTACK CONTENEURISÉE AVEC RABBITMQ

RESUME

Ce document présente un projet web fullstack combinant une API ASP.NET Core 8, un frontend Angular 21 et RabbitMQ pour la communication asynchrone. Il illustre la mise en œuvre d'un système distribué moderne, avec une orchestration via Docker Compose et une intégration de pipelines CI/CD.

Ousseynou Nabalma

Sommaire

1 GENERALITES

1.1 Exigences de bases

- des environnements de développement, test & production
- avec pipelines CI/CD pour continuellement merger, tester, et déployer le code
- je suis ouvert à tout, avec un petit budget, donc privilégier les outils open-source ou les tiers d'entrée
- propose moi des solutions, avec un calendrier de livraison, j'en ai besoin depuis un certain temps
- C# .net +ReactJs + Angular
- PostgreSQL + RabbitMQ + docker
- Git

1.2 Analyse des exigences et propositions

1.2.1 Gestionnaire de versions de code

Le projet sera hébergé sur GitHub. Un seul dépôt regroupera à la fois le code FrontEnd et BackEnd, afin de simplifier la gestion des workflows et de faciliter l'accès au code.

1.2.2 Workflows runners: git runners versus hosted runners

Nous privilégierons l'utilisation de GitHub runners afin de réduire les coûts, compte tenu de notre budget limité. L'option consistant à déployer nos propres machines comme runners aurait également été possible, mais elle s'avère plus onéreuse.

1.2.3 Visibilité du repos

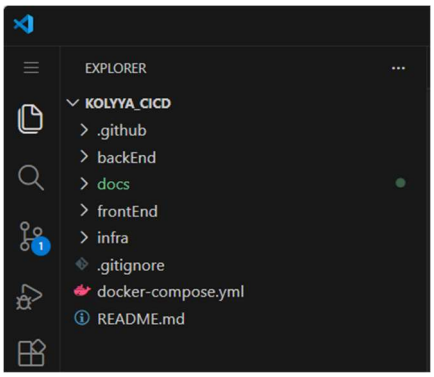
Le choix de la visibilité du dépôt a un impact direct sur le coût du projet. Pour des raisons budgétaires, nous opterons pour un dépôt public. En effet, les dépôts privés disposent d'un quota limité concernant le temps d'exécution des workflows lorsque les runners utilisés sont les GitHub runners.

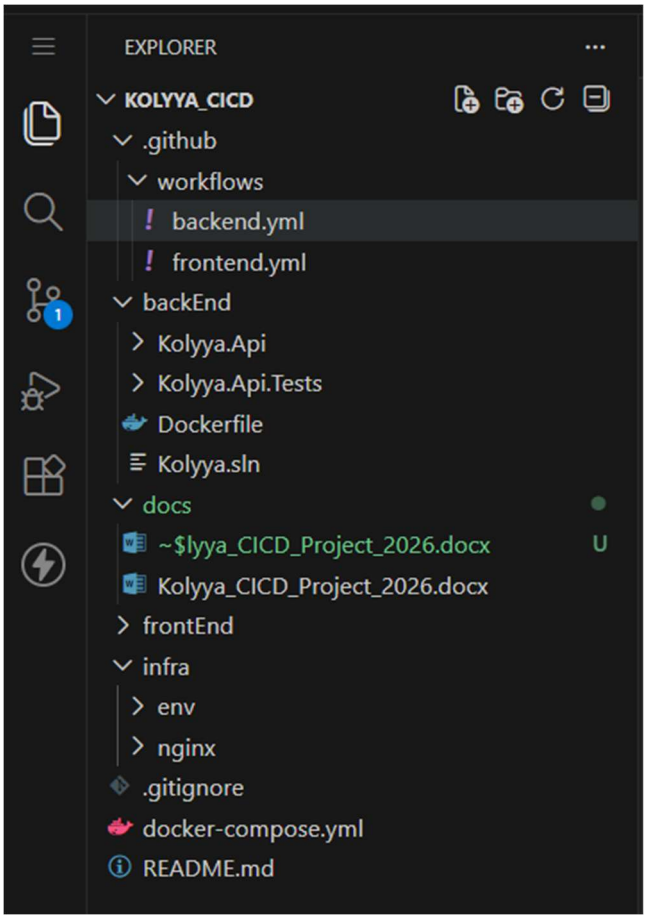
2 CONFIGURATIONS GENERALES

2.1 Structure du dépôt et organisation du code

Le dépôt GitHub est structuré de manière à bien séparer les différentes couches de l'application :

/.github/workflows → Contient les pipelines CI/CD
/backEnd → Contient l'API en .NET
/frontEnd → Contient les interfaces utilisateurs (ReactJS ou Angular)
/infra → Contient les scripts d'infrastructure (les variables d'Env, nginx si besoin, etc.)
/docs → Contient la documentation technique





Le dépôt Git principal regroupe l'ensemble des composants du projet afin d'assurer une meilleure cohérence, visibilité et efficacité pour la mise en place de l'intégration et du déploiement continus (CI/CD).

2.1.1 .github/workflows – Définition des pipelines CI/CD

Ce dossier contient les fichiers YAML définissant les workflows GitHub Actions. Ces workflows sont déclenchés à chaque modification du code dans les branches principales (main, dev).
Le fichier backend.yml configure notamment une pipeline CI qui exécute les étapes suivantes pour le backend :

- Restauration des dépendances .NET
- Vérification du style de code (dotnet format)
- Exécution des tests unitaires via xUnit
- Vérification des vulnérabilités dans les dépendances
- Build de l'image Docker du backend

À terme, un second workflow pourra être ajouté pour le front-end avec les mêmes principes.

2.1.2 backEnd – API .NET + Tests unitaires

Le dossier backEnd contient deux projets .NET :

- Kolyya.Api : l'API principale développée en ASP.NET Core
- Kolyya.Api.Tests : le projet de tests unitaires utilisant xUnit, Moq et FluentAssertions
  Ce projet référence directement Kolyya.Api.csproj pour effectuer des tests sur les classes de l'API (comme SampleItem).

Un fichier solution Kolyya.sln regroupe les deux projets, facilitant la compilation et les tests depuis la CI et les IDEs.

2.1.3 frontEnd – Application Angular

Ce dossier (prévu) contiendra l'application front-end du projet. Sa séparation claire du backend permet :

- d'organiser les builds spécifiques
- de configurer un pipeline CI/CD distinct
- de supporter facilement plusieurs frameworks JS (Angular)

2.1.4 infra – Configuration d'environnement

Le fichier dev.env présent dans ce dossier stocke les variables d'environnement nécessaires au backend, comme la connexion PostgreSQL :
POSTGRES_CONN=Host=kolyya.db;Port=5432;Username=postgres;Password=devpassword;Database=myappdb
Ce fichier est injecté dans Docker grâce à l'option --env-file de docker compose.
Des fichiers similaires pourront être créés pour les environnements test et prod.

2.1.5 docs – Documentation technique

Ce dossier contiendra toute la documentation du projet :

- guides d'installation
- architecture du système
- notes de développement
- consignes pour les développeurs

2.1.6 .gitignore – Fichiers ignorés par Git

Le fichier .gitignore est configuré pour exclure les fichiers inutiles ou sensibles du suivi Git :

- artefacts de compilation .NET (bin, obj)
- fichiers temporaires d'IDE (.vscode, .idea)
- secrets et variables d'environnement (*.env)
- dépendances front-end (node_modules)

2.1.7 docker-compose.yml – Orchestration multi-conteneurs

Le fichier docker-compose.yml permet de démarrer l'environnement de développement local avec :

- l'API backend (port 5000 → 8080 exposé)
- la base de données PostgreSQL (port 5432)

Ce fichier est essentiel pour reproduire un environnement cohérent sur chaque machine, mais aussi pour les phases de test dans les workflows CI/CD.

2.1.8 README.md – Introduction au projet

Le fichier README.md (à enrichir) permet de :

- présenter rapidement le projet
- expliquer comment lancer les services en local avec Docker
- documenter la structure des dossiers
- inclure les badges de CI/CD et des liens vers les outils externes

3 BACKEND : ASP.NET CORE 8.

3.1 Synthèse du projet

3.1.1 Framework utilisé : ASP.NET Core 8 (Minimal API)

3.1.2 Langage : C#

3.1.3 Base de données :

- PostgreSQL version 16
- Connexion configurée via la variable d'environnement ConnectionStrings__DefaultConnection
- Dépendance explicitement définie dans docker-compose.yml

3.1.4 Construction :

- Le projet est compilé avec dotnet build puis publié avec dotnet publish
- Utilisation d'un Dockerfile adapté aux projets ASP.NET Core :
  - mcr.microsoft.com/dotnet/sdk:8.0 pour la compilation
  - mcr.microsoft.com/dotnet/aspnet:8.0 pour l'image finale

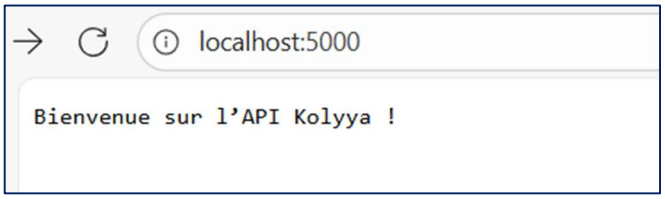
3.1.5 Image Docker :

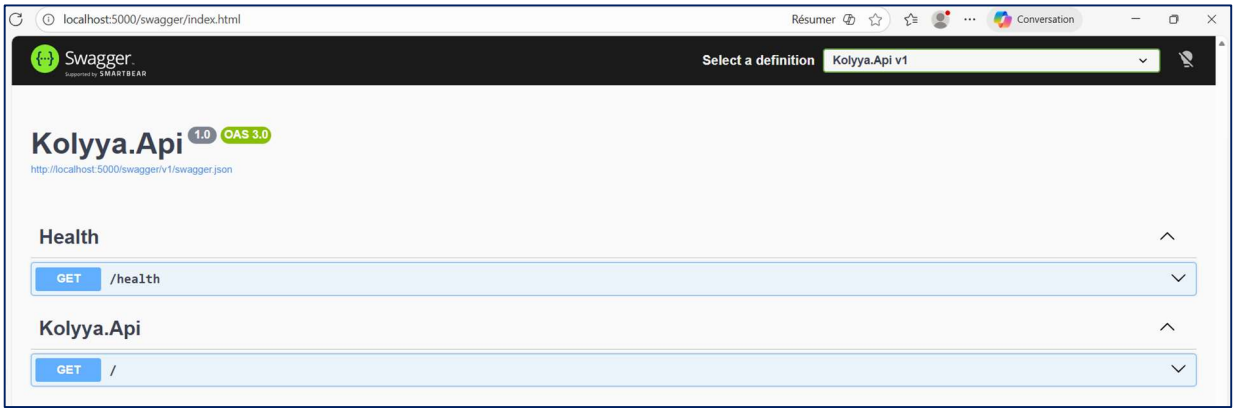
- Multi-stage build pour optimiser la taille
- L'API est exposée sur le port 8080 dans le container

3.1.6 Test local réussi :

- Lancement via docker-compose up --build
- L'API est accessible sur http://localhost:5000 (port mappé via 5000:8080)
- Connexion établie avec la base de données PostgreSQL (port 5432)

3.2 Rendu du projet





4 FRONTEND : ANGULAR 21 + DOCKER

4.1 Synthèse du projet

4.1.1 Framework utilisé : Angular version 21

4.1.2 Version Node : 20 (compatibilité Angular 21)

4.1.3 Construction :

- L'application est compilée avec ng build
- La sortie est générée dans dist/kolyya-frontend/

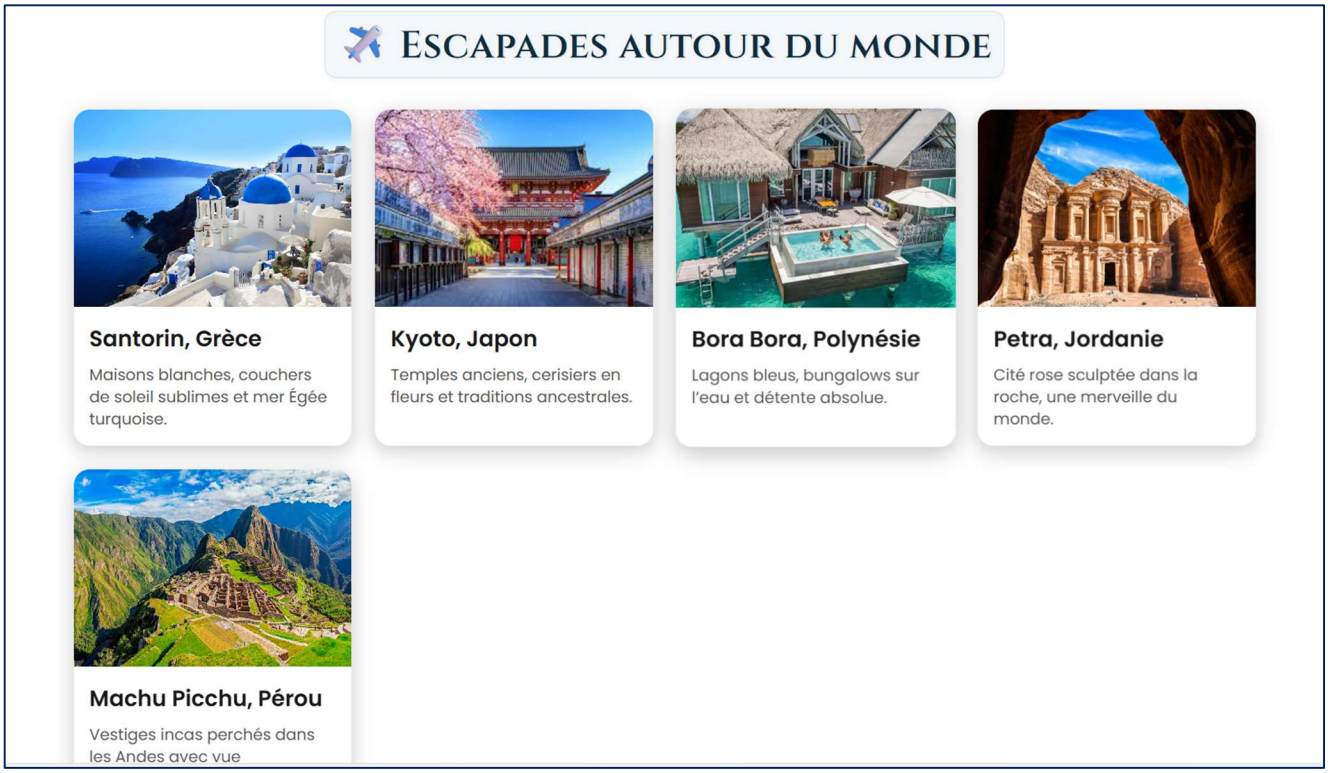
4.1.4 Image Docker :

- Construite avec un Dockerfile multi-stage (Node + Nginx)
- Le build Angular est copié dans /usr/share/nginx/html

4.1.5 Test local réussi :

- Lancement via docker run -p 3000:80 kolyya-frontend
- Résultat visible sur http://localhost:3000 (accès confirmé au frontend)

4.2 Rendu du projet



5 INTÉGRATION DE RABBITMQ

5.1 Objectifs de l'intégration

L'intégration de RabbitMQ permet d'introduire une architecture orientée message (Message-Driven Architecture), facilitant :

- Le découplage entre les services Frontend et Backend.
- La **scalabilité** future en introduisant des traitements asynchrones.
- La **fiabilité** du système via une file d'attente qui persiste les messages jusqu'à traitement.

5.2 Fonctionnement général de la messagerie

5.2.1 Schéma de communication

Voici le flux logique de communication :

1. **Frontend** : Un utilisateur clique sur un bouton "Commander".
2. **API .NET (Producer)** : Envoie un message via RabbitMQ.
3. **RabbitMQ (Broker)** : Route le message via un `Exchange` vers une `Queue`.
4. **Consumer .NET** : Récupère le message et effectue le traitement associé.

5.2.2 4.2 Détails des rôles

Élément	Rôle	Composant
Producer	Publie un message RabbitMQ	`OrdersController.cs`
Exchange (par défaut `fanout`)	Distribue les messages aux files associées	Déclaré implicitement via MassTransit
Queue `touristic-card-orders`	Contient les messages à consommer	Configurée dans `Program.cs`
Consumer	Récupère et traite le message	`TouristicCardOrderedConsumer.cs`

5.3 Structure des fichiers impliqués

Fichier	Rôle
Program.cs	Configure MassTransit et RabbitMQ
dev.env	Contient les variables RABBITMQ_HOST, RABBITMQ_USER, RABBITMQ_PASSWORD
OrdersController.cs	Point d'entrée des requêtes POST /api/orders, joue le rôle de producer
TouristicCardOrdered.cs	Message envoyé dans RabbitMQ (classe DTO)
TouristicCardOrderedConsumer.cs	Classe chargée de consommer le message
touristic-card.component.ts	Frontend Angular avec appel fetch() POST

5.4 Tests réalisés

5.4.1 Depuis Postman

- Envoi d'un POST sur `http://localhost:5000/api/orders` avec payload JSON.

- Résultat attendu : message loggé dans le backend, création de la queue `touristic-card-orders`.

5.4.2 Depuis l'interface RabbitMQ

- Vérification de la présence de la **queue** et des **messages consommés**.
- Monitoring via `http://localhost:15672` avec login `guest/guest`.

5.4.3 Depuis l'application Angular

- Interaction utilisateur sur le bouton "Order".
- Message POST généré via `fetch()` vers `/api/orders`.

5.5 Configuration et variables d'environnement

Les variables sont chargées depuis le fichier `.env` via `docker-compose` :
Elles sont injectées dans l'application backend via :
`Environment.GetEnvironmentVariable("RABBITMQ_HOST")`

5.6 Déploiement dans Docker Compose

Le service `backend` dépend explicitement de RabbitMQ dans `docker-compose.yml`, garantissant que le broker est prêt avant le démarrage de l'API :

```
depends_on:
  - rabbitmq
```

5.7 Résumé et recommandations

L'implémentation RabbitMQ dans le projet Kolyya respecte une architecture asynchrone propre, modulaire et évolutive. Elle permet :

- Un découplage clair Front ↔ Back.
- Une infrastructure prête à accueillir de nouveaux consommateurs ou files spécialisées.
- Une bonne observabilité avec RabbitMQ Management UI.

6 INTEGRATION CONTINUE (CI)

6.1 Workflow Backend CI

6.1.1 Objectif

Ce workflow est conçu pour automatiser les tâches de build, test, linting et analyse de code du backend .NET dès qu'un changement est détecté.

6.1.2 Déclencheurs

Push ou pull request sur les branches `main` ou `dev`
Si des fichiers changent dans :
`backEnd/**` (code backend)
`infra/**` (infrastructure)
`docker-compose.yml`, `.env`, ou ce fichier workflow

6.1.3 Jobs

`build-and-test`

S'exécute sur `ubuntu-latest`

Définit des variables d'environnement :

```
DOTNET_VERSION: "8.0.x"
IMAGE_NAME: kolyya-backend
```

6.1.4 Étapes :

- Checkout du dépôt (`actions/checkout`)
- Setup .NET SDK (`actions/setup-dotnet`)
- Restauration des dépendances avec `dotnet restore`
- Linting (vérification du style de code) avec `dotnet format`
- Tests unitaires avec `dotnet test`

6.2 Workflow Frontend CI

6.2.1 Objectif

Ce workflow GitHub Actions vise à automatiser le processus de **CI (intégration continue)** pour l'application **frontend Angular**. Il permet de :

- Vérifier la qualité du code avec `ng lint`.
- Exécuter les **tests unitaires avec Vitest**.
- Construire l'application Angular.
- Construire l'image Docker correspondante.
- (Optionnellement) préparer l'image à être poussée sur Docker Hub, avec un **tag dynamique** basé sur le commit.
-

6.2.2 Déclencheurs

Le workflow est déclenché automatiquement :

Lors de **push** ou **pull request** sur les branches `main` ou `dev`.

Si ces changements affectent l'un des fichiers ou dossiers suivants :

```
frontEnd/**
infra/**
docker-compose.yml
.env
.github/workflows/frontend.yml
```

6.2.3 Jobs

Le workflow contient un **unique job** nommé `build-test-dockerize`, qui s'exécute sur un environnement Linux (`ubuntu-latest`).

6.2.4 Étapes

- **Checkout du code source** :

Utilise l'action officielle `actions/checkout@v3` pour récupérer le dépôt.

- **Configuration de Node.js 20** :

Utilise `actions/setup-node@v3` pour installer Node.js version 20, compatible avec Angular 21 et Vitest.

- **Installation des dépendances** :

Exécute `npm ci` pour une installation propre basée sur le `package-lock.json`.

- **Linting (analyse statique)** :

Exécute `npm run lint` pour vérifier la qualité du code selon les règles Angular définies dans le projet.

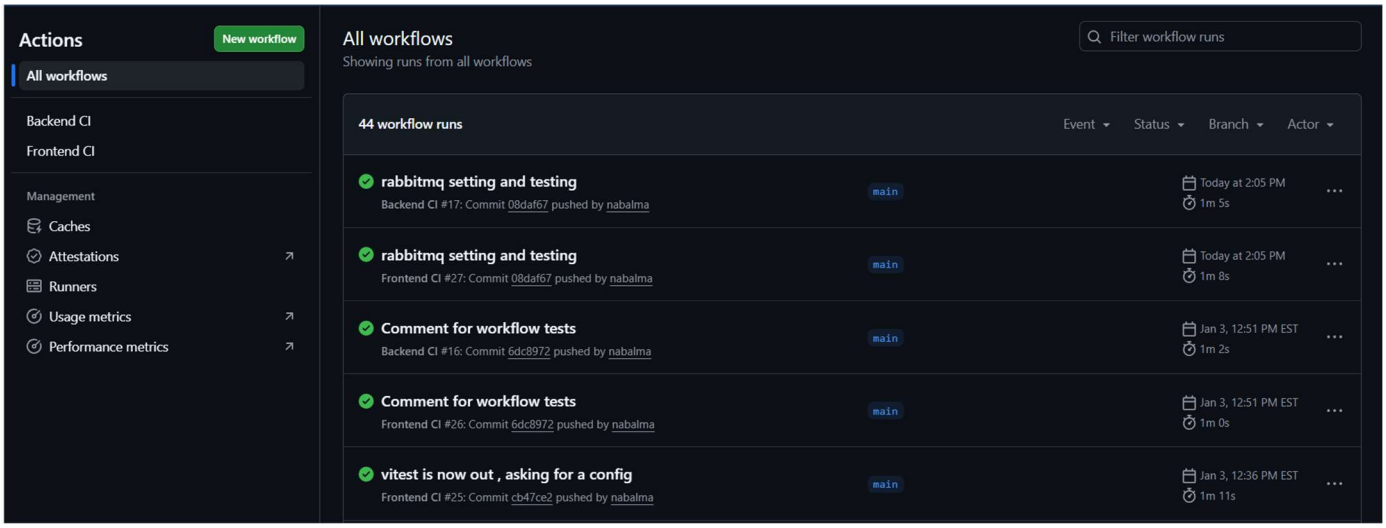
- **Build Angular** :

Compile l'application en mode production avec `npm run build`.

- **Build Docker image** :

Construit une image Docker locale avec un tag dynamique basé sur le `GITHUB_SHA` (commit) :
`docker build -t kolyya-frontend:${GITHUB_SHA::7} .`

6.3 Exécutions automatique des Workflows sur github.

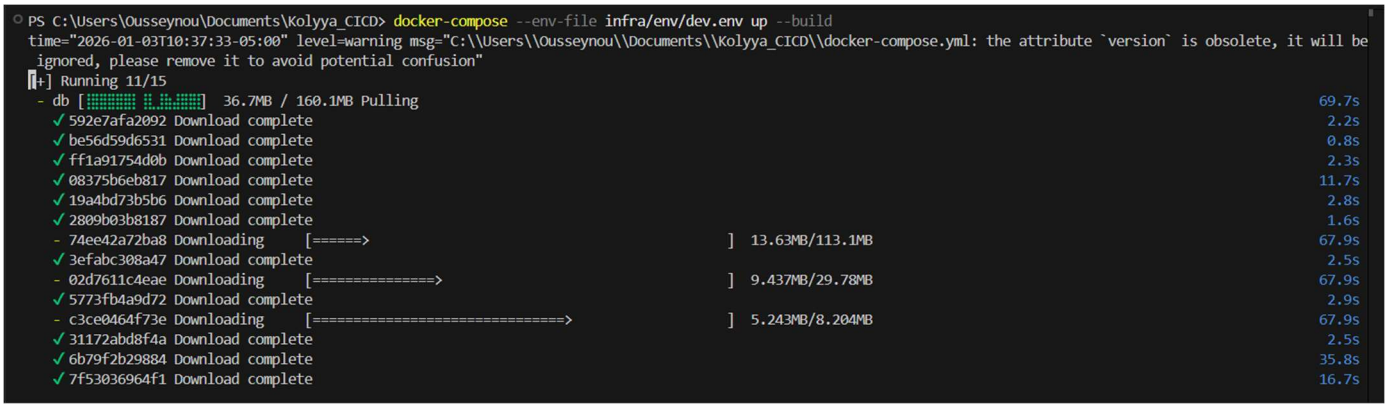


7 LANCEMENT ET GESTION DU PROJET

7.1 Démarrer les services (Frontend, Backend, Base de données, RabbitMQ)

Pour lancer l'ensemble du projet en environnement de développement, utilisez la commande suivante à la racine du projet :
docker-compose --env-file infra/env/dev.env up --build
Cette commande :

- Reconstruit les images Docker si nécessaire (--build)
- Utilise les variables d'environnement définies dans dev.env
- Lance tous les services définis dans docker-compose.yml :
  - backend (API .NET)
  - frontend (Angular)
  - db (PostgreSQL)
  - rabbitmq (Broker de messages)



7.2 Arrêter proprement l'ensemble des services

Pour arrêter tous les conteneurs et libérer les volumes de données persistants, exécutez :
docker-compose down -v
Cette commande :

- Arrête tous les conteneurs en cours d'exécution
- Supprime les réseaux, volumes nommés et autres ressources créées automatiquement