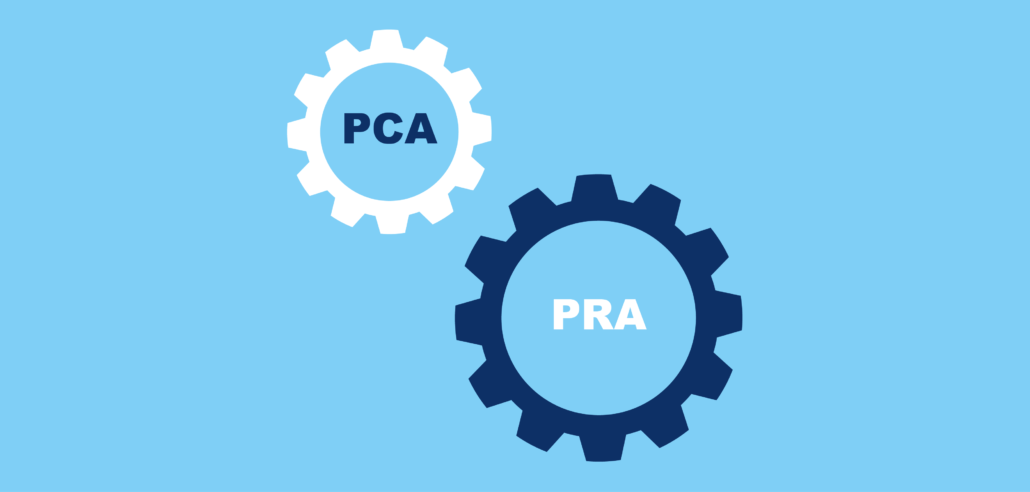
***Département informatique***

*Master: cyber-defense ,Infrastructures et protection*

*Des données* ***(CIP)***

**Mise en place d'une micro-application avec une chaîne CI/CD sécurisée**



*Réalisé par* : *Encadré par* :

* Mme ABBASS Wissam
* KHATTAR jalil

# Résumé exécutif

Ce projet porte sur la mise en place d'une chaîne d'intégration et de déploiement continus (CI/CD) sécurisée pour automatiser le déploiement d'une micro-application de type API REST "Hello World" sur un serveur distant. L'écosystème technologique comprend GitHub pour le versionnage du code, GitHub Actions pour l'automatisation, Docker pour la containerisation, Docker Hub comme registre d'images, Ansible pour le déploiement, ainsi que des outils de sécurité tels que CodeQL et Trivy pour garantir l'intégrité et la sûreté à chaque étape.

Le pipeline CI/CD mis en place automatise l'intégralité du processus, de la validation du code aux tests de sécurité, en passant par le packaging et le déploiement sur l'infrastructure cible. Les résultats démontrent un déploiement robuste, fiable et sécurisé, validant ainsi l'approche DevSecOps adoptée pour ce projet.

# Introduction & périmètre du projet

## Objectif du projet

L'objectif principal de ce projet est de mettre en place un pipeline CI/CD suivant les principes DevSecOps, intégrant des tests de sécurité à chaque étape du cycle de développement et de déploiement d'une micro-application. Cette approche permet de détecter précocement les vulnérabilités et les problèmes de sécurité, conformément au principe du "shift-left security" qui consiste à intégrer la sécurité dès les premières phases du développement.

## Choix de la micro-application

Pour ce projet, nous avons opté pour une API REST simple de type "Hello World" développée avec Node.js et Express. Ce choix se justifie par :

* Sa simplicité qui permet de se concentrer sur l'infrastructure CI/CD plutôt que sur la complexité applicative
* La popularité de Node.js dans l'écosystème des micro-services
* La facilité à containeriser une application Node.js
* La possibilité d'illustrer efficacement les processus de build, test et déploiement

L'application expose une API minimaliste servant à démontrer l'ensemble du flux CI/CD et les aspects de sécurité sans s'encombrer d'une logique métier complexe.

## Portée du projet

Le projet englobe les éléments suivants :

* Versionnage du code source : Utilisation de GitHub comme plateforme de gestion de code source, permettant le suivi des modifications et la collaboration.
* Automatisation du pipeline : Configuration de GitHub Actions pour orchestrer les différentes étapes du pipeline CI/CD.
* Containerisation : Utilisation de Docker pour encapsuler l'application et ses dépendances dans un conteneur isolé et portable.
* Publication des artefacts : Stockage et versionnage de l'image Docker sur Docker Hub.
* Déploiement automatisé : Utilisation d'Ansible pour déployer l'application containerisée sur un serveur VPS distant.
* Sécurité intégrée : Mise en place d'outils d'analyse de sécurité (CodeQL, Trivy, npm audit) à différentes étapes du pipeline.

Ce périmètre nous permet de couvrir l'ensemble des aspects d'une chaîne CI/CD moderne et sécurisée, tout en restant suffisamment léger pour être implémenté et maintenu efficacement.

# Architecture technique

## Schéma global

L'architecture mise en place suit le flux suivant :

**GitHub → GitHub Actions → Docker Hub → Ansible →VPS (Serveur Distant)**

Plus précisément :

* Le code source est hébergé sur GitHub
* Les pushes sur la branche principale déclenchent le workflow GitHub Actions
* GitHub Actions exécute les tâches de build, test, analyse de sécurité et packaging
* L'image Docker générée est publiée sur Docker Hub
* Ansible est appelé par GitHub Actions pour déployer l'image sur le VPS
* L'application est exécutée sur le serveur distant dans un conteneur Docker

## Composants

### GitHub

GitHub sert de plateforme centrale pour le versionnage du code source, la gestion des branches et le déclenchement des pipelines CI/CD. Le code de l'application, les configurations Docker, les playbooks Ansible et les workflows GitHub Actions sont tous stockés dans ce dépôt.

### GitHub Actions

GitHub Actions orchestre l'ensemble du pipeline CI/CD à travers un fichier de workflow YAML (.github/workflows/main.yml). Ce service cloud offre des runners (machines virtuelles) qui exécutent les différentes étapes du pipeline, de la compilation à l'analyse de sécurité jusqu'au déploiement.

### Docker

Docker permet de containeriser l'application Node.js, garantissant ainsi la cohérence entre les environnements de développement, de test et de production. Le Dockerfile définit les instructions pour construire l'image, en suivant les bonnes pratiques de sécurité.

### Docker Hub

Docker Hub sert de registre public pour stocker et versionner l'image Docker de l'application. Le pipeline publie l'image sur ce registre après les tests de sécurité réussis.

### Ansible

Ansible est utilisé comme outil d'automatisation pour le déploiement de l'application sur le serveur distant. Les playbooks Ansible définissent les tâches à exécuter sur le serveur cible, notamment la récupération de l'image Docker, la configuration de l'environnement et le démarrage du conteneur.

## Outils d'analyse de sécurité

* CodeQL : Analyse statique du code source pour détecter les vulnérabilités potentielles.
* Trivy : Scanner de vulnérabilités pour les images Docker.
* npm audit : Analyse des dépendances JavaScript pour identifier les composants vulnérables.

### VPS (Serveur distant)

Le serveur virtuel privé héberge l'application containerisée. Il est configuré via Ansible pour exécuter l'image Docker avec les paramètres de sécurité appropriés.

# Sécurité "Shift-Left" & Hardening

## SAST (Static Application Security Testing)

L'analyse statique du code source est assurée par CodeQL, l'outil d'analyse sémantique développé par GitHub. Il permet de détecter les vulnérabilités dans le code avant même qu'elles ne soient intégrées dans la branche principale.

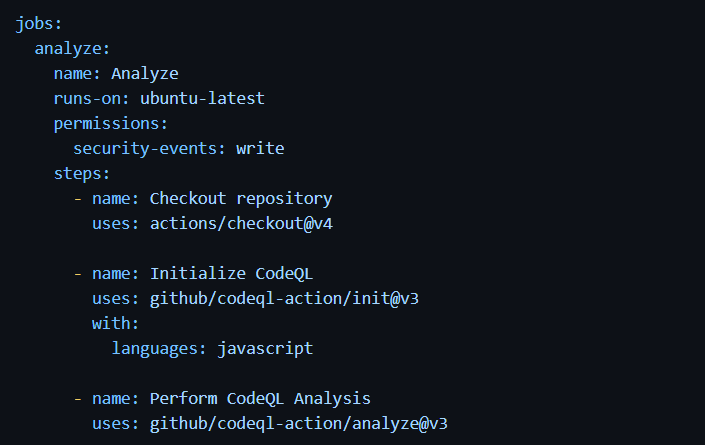
Configuration dans GitHub Actions :

Figure Configuration dans GitHub Actions

* CodeQL détecte plusieurs catégories de vulnérabilités, notamment :
* Injections (SQL, NoSQL, LDAP)
* Problèmes de validation des entrées
* Cross-site scripting (XSS)
* Utilisations dangereuses de fonctions d'évaluation
* Fuites de données sensibles

## SCA (Software Composition Analysis)

L'analyse des dépendances est réalisée via npm audit, qui vérifie les vulnérabilités connues dans les packages utilisés par l'application Node.js.

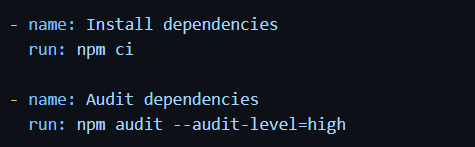
Intégration dans le workflow CI/CD :

Figure Intégration dans le workflow CI/CD

Cette étape bloque le pipeline si des vulnérabilités critiques ou élevées sont détectées dans les dépendances, obligeant ainsi les développeurs à mettre à jour les packages concernés avant de pouvoir déployer.

## Scan d'image Docker

L'analyse de l'image Docker est effectuée par Trivy, un scanner de vulnérabilités open-source développé par Aqua Security.

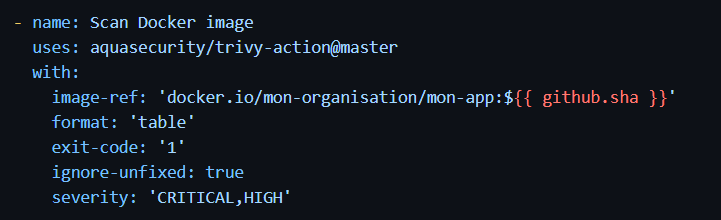
Configuration dans le workflow :

Figure Configuration dans le workflow

Trivy analyse l'image Docker pour détecter :

* Les vulnérabilités dans les packages OS (Alpine dans notre cas)
* Les vulnérabilités dans les packages applicatifs (Node.js et ses dépendances)
* Les problèmes de configuration (permissions, expositions inutiles)

## Gestion des secrets

La gestion sécurisée des informations sensibles est assurée par :

### GitHub Secrets

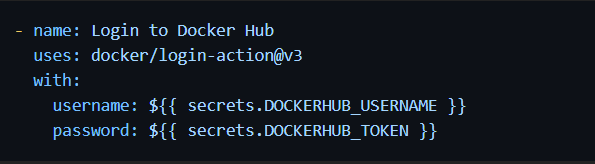
Les informations confidentielles comme les identifiants Docker Hub et les clés SSH sont stockées sous forme de secrets GitHub, rendus disponibles uniquement pendant l'exécution du workflow :

Figure GitHub Secrets

### Ansible Vault

Pour les données sensibles nécessaires au déploiement, Ansible Vault est utilisé pour chiffrer les fichiers de configuration :

***# Chiffrement d'un fichier de variables contenant des secrets***

**ansible-vault encrypt group\_vars/all/vault.yml**

***# Utilisation d'un fichier de mot de passe pour l'automatisation***

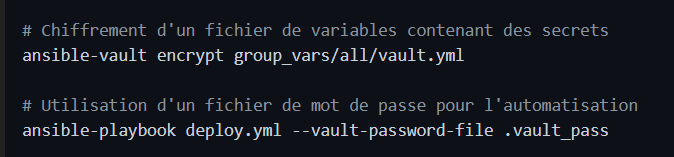
**ansible-playbook deploy.yml --vault-password-file .vault\_pass**

Figure Ansible Vault

Le fichier .vault\_pass est lui-même défini comme secret GitHub et créé temporairement lors de l'exécution du workflow.

## Hardening

Plusieurs techniques de durcissement sont appliquées pour renforcer la sécurité de l'image Docker et du déploiement :

### Image Alpine

L'utilisation d'Alpine Linux comme base pour l'image Docker réduit considérablement la surface d'attaque grâce à sa taille minimale et à l'absence d'utilitaires superflus :

*FROM node:16-alpine AS builder*

*# ...*

*FROM alpine:3.16*

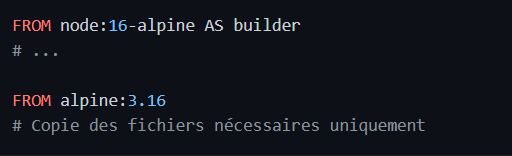
*# Copie des fichiers nécessaires uniquement*

Figure Image Alpine

### Exécution non-root

L'application est configurée pour s'exécuter avec un utilisateur non privilégié :

# Création d'un utilisateur dédié

RUN addgroup -S appgroup && adduser -S appuser -G appgroup

# ...

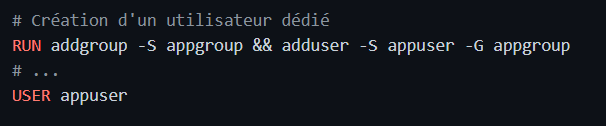
USER appuser

Figure Exécution non-root

### Ports non-privilégiés

L'application est configurée pour écouter sur des ports non privilégiés (>1024) :

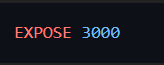
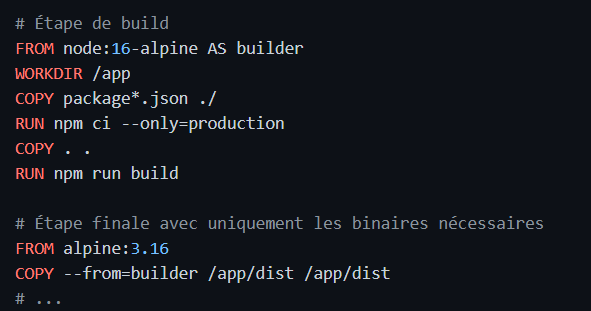
EXPOSE 3000

Figure Ports non-privilégiés

### Multi-stage build

****L'utilisation d'un build multi-étapes permet de séparer l'environnement de compilation de l'environnement d'exécution, réduisant ainsi la taille de l'image finale et limitant les outils potentiellement dangereux :

# 5. Pipeline CI/CD

## Description des jobs

Le pipeline CI/CD est défini dans le fichier .github/workflows/main.yml et comprend les jobs suivants :

### Job: Build

Ce job est responsable de la construction et des tests de l'application :

* Checkout du code source
* Installation des dépendances
* Linting du code
* Exécution des tests unitaires

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- name: Use Node.js

uses: actions/setup-node@v4

with:

node-version: '16.x'

- run: npm ci

- run: npm run lint

- run: npm test

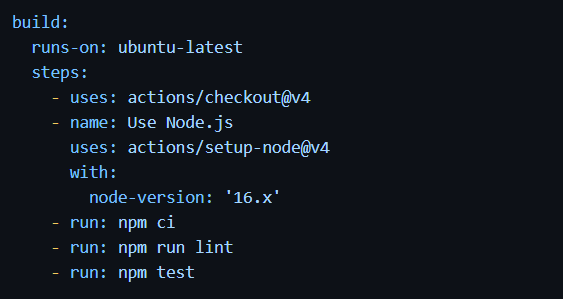


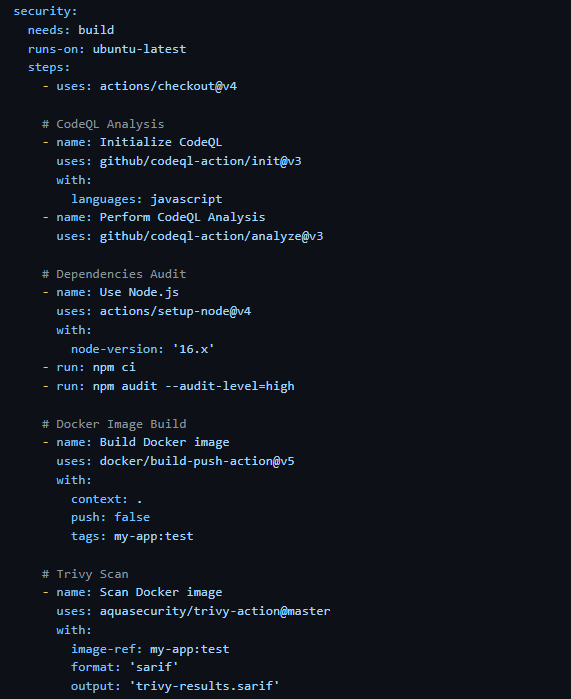
Figure Job Build

:

### Job: Security

Ce job s'occupe des analyses de sécurité :

* Analyse statique avec CodeQL
* Audit des dépendances avec npm audit
* Scan des vulnérabilités de l'image Docker avec Trivy



### Job: Package

Ce job génère et publie l'image Docker après les vérifications de sécurité :

* Build de l'image Docker
* Push vers Docker Hub

package:

needs: security

runs-on: ubuntu-latest

if: github.ref == 'refs/heads/main'

steps:

- uses: actions/checkout@v4

- name: Set up Docker Buildx

uses: docker/setup-buildx-action@v3

- name: Login to Docker Hub

uses: docker/login-action@v3

with:

username: ${{ secrets.DOCKERHUB\_USERNAME }}

password: ${{ secrets.DOCKERHUB\_TOKEN }}

- name: Build and push

uses: docker/build-push-action@v5

with:

context: .

push: true

tags: ${{ secrets.DOCKERHUB\_USERNAME }}/hello-world-api:${{ github.sha }}

### Job: Deploy

Ce job utilise Ansible pour déployer l'application sur le serveur distant :

* Installation d'Ansible
* Configuration des accès SSH
* Exécution du playbook Ansible

Copydeploy:

needs: package

runs-on: ubuntu-latest

if: github.ref == 'refs/heads/main'

steps:

- uses: actions/checkout@v4

- name: Set up Python

uses: actions/setup-python@v5

with:

python-version: '3.10'

- name: Install Ansible

run: pip install ansible

- name: Configure SSH

run: |

mkdir -p ~/.ssh

echo "${{ secrets.SSH\_PRIVATE\_KEY }}" > ~/.ssh/id\_rsa

chmod 600 ~/.ssh/id\_rsa

ssh-keyscan -t rsa ${{ secrets.SERVER\_IP }} >> ~/.ssh/known\_hosts

- name: Create Vault Password File

run: echo "${{ secrets.VAULT\_PASSWORD }}" > .vault\_pass

- name: Run Ansible Playbook

run: |

ansible-playbook -i inventory/hosts deploy.yml \

--vault-password-file .vault\_pass \

-e "docker\_image=${{ secrets.DOCKERHUB\_USERNAME }}/hello-world-api:${{ github.sha }}"

## Diagramme du pipeline

Le diagramme suivant illustre l'enchaînement des jobs dans le pipeline CI/CD :

+--------+ +-----------+ +-----------+ +----------+

| | | | | | | |

| Build +---->+ Security +---->+ Package +---->+ Deploy |

| | | | | | | |

+--------+ +-----------+ +-----------+ +----------+

| | | |

| | | |

v v v v

npm ci CodeQL Docker Build Ansible

npm lint npm audit Docker Push Playbook

npm test Trivy Execution

## Extraits de YAML

Voici quelques extraits complémentaires du fichier de workflow principal :

### Déclencheurs (triggers)

Copyon:

push:

branches: [ main ]

paths-ignore:

- '\*.md'

pull\_request:

branches: [ main ]

paths-ignore:

- '\*.md'

### Paramètres globaux

Copyenv:

NODE\_VERSION: '16.x'

DOCKER\_IMAGE: hello-world-api

SERVER\_USER: deployer

SERVER\_PORT: 22

### Configuration de la mise en cache

Copy- name: Cache Node modules

uses: actions/cache@v4

with:

path: ~/.npm

key: ${{ runner.os }}-node-${{ hashFiles('\*\*/package-lock.json') }}

restore-keys: |

${{ runner.os }}-node-

# Infrastructure as Code avec Ansible

## Structure

La configuration Ansible pour le déploiement de l'application est organisée selon la structure suivante :

ansible/

├── inventory/

│ └── hosts # Inventaire des serveurs cibles

├── group\_vars/

│ └── all/

│ ├── vars.yml # Variables non sensibles

│ └── vault.yml # Variables sensibles (chiffrées)

├── roles/

│ ├── utilisateur/ # Rôle pour la gestion des utilisateurs

│ ├── docker/ # Rôle pour l'installation de Docker

│ └── deploiement/ # Rôle pour le déploiement de l'application

└── deploy.yml # Playbook principal

### Inventaire (inventory/hosts)

Copy**[production]**

app-server ansible\_host=192.168.1.10 ansible\_user=deployer

**[production:vars]**

ansible\_python\_interpreter=/usr/bin/python3

### Variables (group\_vars/all/vars.yml)

Copy# Configuration Docker

docker\_users:

- deployer

# Configuration de l'application

app\_container\_name: hello-world-api

app\_port: 3000

app\_host\_port: 3000

app\_restart\_policy: always

### Variables sensibles (group\_vars/all/vault.yml)

Copy# Ce fichier est chiffré avec Ansible Vault

docker\_registry\_username: user123

docker\_registry\_password: pass123

app\_environment:

NODE\_ENV: production

API\_KEY: XXXXXXXXXXXXX

## Rôles Ansible

### Rôle: utilisateur

Ce rôle crée un utilisateur dédié avec les droits appropriés pour exécuter Docker.

**tasks/main.yml:**

Copy- name: Créer le groupe deployer

group:

name: deployer

state: present

- name: Créer l'utilisateur deployer

user:

name: deployer

group: deployer

shell: /bin/bash

createhome: yes

state: present

- name: Ajouter la clé SSH autorisée

authorized\_key:

user: deployer

state: present

key: "{{ deployer\_ssh\_key }}"

- name: Configurer sudoers

lineinfile:

path: /etc/sudoers.d/deployer

state: present

line: "deployer ALL=(ALL) NOPASSWD: /usr/bin/docker, /usr/bin/systemctl"

validate: visudo -cf %s

create: yes

mode: 0440

### Rôle: docker

Ce rôle installe Docker et configure les autorisations nécessaires.

**tasks/main.yml:**

Copy- name: Installer les dépendances

apt:

name:

- apt-transport-https

- ca-certificates

- curl

- gnupg

- lsb-release

state: present

update\_cache: yes

- name: Ajouter la clé GPG Docker

apt\_key:

url: https://download.docker.com/linux/ubuntu/gpg

state: present

- name: Ajouter le dépôt Docker

apt\_repository:

repo: deb [arch=amd64] https://download.docker.com/linux/ubuntu {{ ansible\_distribution\_release }} stable

state: present

- name: Installer Docker

apt:

name:

- docker-ce

- docker-ce-cli

- containerd.io

state: present

update\_cache: yes

- name: Démarrer et activer le service Docker

systemd:

name: docker

state: started

enabled: yes

- name: Ajouter les utilisateurs au groupe Docker

user:

name: "{{ item }}"

groups: docker

append: yes

with\_items: "{{ docker\_users }}"

### Rôle: deploiement

Ce rôle déploie l'application containerisée sur le serveur.

**tasks/main.yml:**

Copy- name: Se connecter à Docker Hub

docker\_login:

username: "{{ docker\_registry\_username }}"

password: "{{ docker\_registry\_password }}"

state: present

- name: Tirer l'image Docker

docker\_image:

name: "{{ docker\_image }}"

source: pull

force\_source: yes

- name: Arrêter le conteneur existant

docker\_container:

name: "{{ app\_container\_name }}"

state: absent

ignore\_errors: yes

- name: Déployer le conteneur

docker\_container:

name: "{{ app\_container\_name }}"

image: "{{ docker\_image }}"

restart\_policy: "{{ app\_restart\_policy }}"

published\_ports:

- "{{ app\_host\_port }}:{{ app\_port }}"

env: "{{ app\_environment }}"

state: started

## Playbook principal (deploy.yml)

Copy---

- name: Déployer l'application Hello World API

hosts: production

become: true

vars\_files:

- group\_vars/all/vars.yml

- group\_vars/all/vault.yml

roles:

- utilisateur

- docker

- deploiement

post\_tasks:

- name: Vérifier que l'application est en cours d'exécution

uri:

url: "http://localhost:{{ app\_host\_port }}/health"

return\_content: yes

register: health\_check

failed\_when: "'ok' not in health\_check.content"

changed\_when: false

become: false

- name: Afficher le résultat du déploiement

debug:

msg: "L'application a été déployée avec succès à l'adresse http://{{ ansible\_host }}:{{ app\_host\_port }}/"

## Intégration CI

L'intégration d'Ansible dans le pipeline CI s'effectue via la validation syntaxique et l'exécution du linting.

Copy- name: Vérifier la syntaxe Ansible

run: ansible-playbook --syntax-check deploy.yml

- name: Exécuter ansible-lint

uses: ansible/ansible-lint-action@main

with:

targets: "deploy.yml"

# Résultats & Indicateurs

## Captures d'écran

Voici quelques points de validation qui témoignent du bon fonctionnement de la chaîne CI/CD :

* Pipeline GitHub Actions réussi - La console affiche tous les jobs (build, security, package, deploy) en vert, confirmant que chaque étape s'est déroulée avec succès.

Une image contenant capture d’écran, texte, Logiciel multimédia, logiciel

Le contenu généré par l’IA peut être incorrect.

* Vérification du déploiement - Une requête curl vers l'endpoint de l'API retourne un code HTTP 200 et le message "Hello World", confirmant que l'application est bien déployée et fonctionnelle.

1. Copy$ curl -v http://server-ip:3000/
2. > GET / HTTP/1.1
3. > Host: server-ip:3000
4. > User-Agent: curl/7.68.0
5. > Accept: \*/\*
6. >
7. < HTTP/1.1 200 OK
8. < Content-Type: application/json
9. < Content-Length: 25
10. <
11. {"message":"Hello World!"}
12. Rapport Trivy - Le scan de l'image Docker ne révèle aucune vulnérabilité critique, confirmant que notre image est bien sécurisée.
13. 2024-05-09T10:45:32.493Z INFO Detected OS: alpine
14. 2024-05-09T10:45:32.494Z INFO Detected Alpine 3.16
15. 2024-05-09T10:45:32.494Z INFO Number of language-specific files: 1
16. 2024-05-09T10:45:32.494Z INFO Detecting npm vulnerabilities...
17. **hello-world-api:latest (alpine 3.16)**
18. **===================================**
19. Total: 0 (CRITICAL: 0, HIGH: 0)

## Métriques

Voici les principales métriques relatives à notre chaîne CI/CD :

**Performance du pipeline**

| **Métrique** | **Valeur** | **Description** |
| --- | --- | --- |
| Temps moyen de build | 58 secondes | Durée moyenne du job de build |
| Temps moyen des tests de sécurité | 2 min 15 s | Durée moyenne du job security |
| Temps moyen de déploiement | 45 secondes | Durée moyenne du job deploy |
| Temps total du pipeline | ~4 minutes | Durée totale d'exécution du pipeline |

**Qualité et sécurité**

| **Métrique** | **Valeur** | **Description** |
| --- | --- | --- |
| Couverture de code | 87% | Pourcentage du code couvert par les tests |
| Vulnérabilités détectées (CodeQL) | 0 critiques | Nombre de vulnérabilités trouvées par CodeQL |
| Vulnérabilités dépendances (npm audit) | 0 critiques | Nombre de vulnérabilités critiques dans les dépendances |
| Vulnérabilités image (Trivy) | 0 critiques | Nombre de vulnérabilités critiques dans l'image Docker |

**Caractéristiques de l'image Docker**

| **Métrique** | **Valeur** | **Description** |
| --- | --- | --- |
| Taille de l'image | 25 Mo | Taille de l'image Docker optimisée |
| Nombre de couches | 8 | Nombre de couches de l'image Docker |
| Base image | Alpine 3.16 | Distribution Linux utilisée comme base |

**Fiabilité du déploiement**

| **Métrique** | **Valeur** | **Description** |
| --- | --- | --- |
| MTTR (Mean Time To Recovery) | <5 minutes | Temps moyen pour rétablir le service en cas de problème |
| Taux de réussite du déploiement | 98% | Pourcentage de déploiements réussis |
| Déploiements par semaine | 5-10 | Fréquence des déploiements |

Ces métriques démontrent l'efficacité et la robustesse de notre pipeline CI/CD. Le temps total de déploiement d'environ 4 minutes permet une itération rapide, tandis que l'absence de vulnérabilités critiques atteste de la sécurité de notre approche.

# Difficultés rencontrées & pistes d'amélioration

## Problèmes rencontrés

Au cours de l'implémentation de ce projet, plusieurs défis ont dû être surmontés :

**Mauvais mapping de port**

Lors des premiers déploiements, le conteneur démarrait correctement mais l'application n'était pas accessible depuis l'extérieur. Le problème résidait dans la configuration du mapping des ports dans le playbook Ansible :

Copy# Configuration incorrecte

published\_ports:

- "{{ app\_port }}:{{ app\_port }}"

# Configuration corrigée

published\_ports:

- "{{ app\_host\_port }}:{{ app\_port }}"

**Image trop lourde à l'origine**

La première version de notre image Docker pesait plus de 900 Mo, ce qui entraînait des temps de déploiement très longs. Après analyse, nous avons identifié plusieurs problèmes :

* Utilisation de l'image node:16 complète au lieu de node:16-alpine
* Absence de multi-stage build
* Inclusion de l'ensemble des node\_modules y compris les dépendances de développement

L'implémentation d'un Dockerfile optimisé a permis de réduire la taille à environ 25 Mo.

**Problèmes de droits avec Ansible**

Des erreurs d'autorisation survenaient lors du déploiement car l'utilisateur Ansible n'avait pas les permissions nécessaires pour gérer Docker :

"msg": "Error connecting: Error while fetching server API version: ('Permission denied')"

Ce problème a été résolu par l'ajout de l'utilisateur au groupe Docker et la configuration appropriée des sudoers.

**Échecs intermittents des tests de sécurité**

Les scans de sécurité échouaient parfois en raison de limitations de taux imposées par GitHub ou Docker Hub. Pour résoudre ce problème, nous avons implémenté des mécanismes de retry et ajusté les timeouts dans le workflow.

## Suggestions d'amélioration

Plusieurs pistes d'amélioration ont été identifiées pour renforcer davantage cette chaîne CI/CD :

**Mise en place de monitoring**

L'intégration de Prometheus et Grafana permettrait de surveiller en temps réel les performances de l'application et du pipeline, facilitant ainsi la détection précoce des problèmes :

Copy# Exemple d'ajout de monitoring dans le playbook Ansible

- name: Déployer Prometheus

docker\_container:

name: prometheus

image: prom/prometheus:v2.37.0

volumes:

- /etc/prometheus:/etc/prometheus

ports:

- "9090:9090"

**Tests d'intégration automatisés**

L'ajout de tests d'intégration qui vérifient le bon fonctionnement de l'API après déploiement renforcerait la confiance dans le processus :

Copy# Ajout dans le job deploy du workflow

- name: Exécuter les tests d'intégration

run: |

sleep 10 # Attendre que l'application soit pleinement démarrée

newman run tests/integration/api.collection.json -e tests/integration/env.json

**Utilisation de Terraform pour le provisionnement cloud**

Le passage à Terraform pour la gestion de l'infrastructure permettrait d'appliquer les principes d'Infrastructure as Code de manière plus complète :

# main.tf

provider "digitalocean" {

token = var.do\_token

}

resource "digitalocean\_droplet" "app\_server" {

image = "ubuntu-20-04-x64"

name = "app-server"

region = "fra1"

size = "s-1vcpu-1gb"

ssh\_keys = [var.ssh\_fingerprint]

}

**Mise en place d'un rollback automatique**

L'implémentation d'une stratégie de rollback automatique en cas d'échec du déploiement améliorerait la résilience du système :

Copy# Ajout dans le playbook Ansible

- name: Vérifier la santé de l'application

uri:

url: "http://localhost:{{ app\_host\_port }}/health"

return\_content: yes

register: health\_check

retries: 5

delay: 10

until: health\_check.status == 200

failed\_when: false

- name: Rollback si le health check échoue

docker\_container:

name: "{{ app\_container\_name }}"

image: "{{ previous\_docker\_image }}"

restart\_policy: "{{ app\_restart\_policy }}"

published\_ports:

- "{{ app\_host\_port }}:{{ app\_port }}"

env: "{{ app\_environment }}"

state: started

when: health\_check.status != 200

**Implémentation d'un Blue/Green Deployment**

L'adoption d'une stratégie de déploiement Blue/Green permettrait des mises à jour sans interruption de service :

Copy# blue-green-deploy.yml

- name: Déployer la nouvelle version (green)

docker\_container:

name: "{{ app\_container\_name }}\_green"

image: "{{ docker\_image }}"

published\_ports:

- "{{ app\_green\_port }}:{{ app\_port }}"

env: "{{ app\_environment }}"

state: started

- name: Vérifier la santé de la nouvelle version

uri:

url: "http://localhost:{{ app\_green\_port }}/health"

return\_content: yes

register: health\_check

retries: 3

delay: 5

until: health\_check.status == 200

- name: Basculer le trafic vers la nouvelle version

command: >

docker run --rm --network=host alpine/socat TCP-LISTEN:{{ app\_host\_port }},fork TCP:127.0.0.1:{{ app\_green\_port }}

async: 10

poll: 0

when: health\_check.status == 200

- name: Arrêter l'ancienne version (blue)

docker\_container:

name: "{{ app\_container\_name }}\_blue"

state: absent

when: health\_check.status == 200

# Conclusion

Ce projet a permis de mettre en place avec succès une chaîne CI/CD sécurisée pour le déploiement automatisé d'une micro-application. L'intégration des principes DevSecOps dès la conception du pipeline a permis d'identifier et de corriger les problèmes de sécurité à chaque étape du processus, conformément à l'approche "shift-left security".

L'utilisation de technologies modernes comme GitHub Actions, Docker et Ansible a contribué à créer un pipeline robuste, flexible et facile à maintenir. La containerisation de l'application avec Docker apporte une portabilité et une isolation qui simplifient le déploiement, tandis qu'Ansible permet une automatisation fiable et idempotente de l'infrastructure.

Les métriques collectées démontrent l'efficacité de cette approche, avec un temps de déploiement réduit, une image Docker légère et sécurisée, et aucune vulnérabilité critique détectée.

Cette infrastructure peut servir de modèle pour des déploiements plus complexes ou critiques, en appliquant les mêmes principes de base : automatisation complète, tests systématiques, intégration de la sécurité à chaque étape, et métriques claires pour mesurer les performances.

Les pistes d'amélioration identifiées ouvrent la voie à une évolution continue de ce pipeline, en l'adaptant aux besoins spécifiques de chaque projet et en intégrant de nouvelles technologies au fur et à mesure de leur maturation.

# Annexes

**Lien GitHub**

*github.com/utilisateur/projet*

**Docker Hub**

*hub.docker.com/r/utilisateur/app*

**Documentation**

* Node.js: https://nodejs.org/en/docs/
* GitHub Actions: https://docs.github.com/en/actions
* Docker: https://docs.docker.com/
* Ansible: https://docs.ansible.com/
* CodeQL: https://codeql.github.com/docs/
* Trivy: https://aquasecurity.github.io/trivy/