Tutoriel d'utilisation

❖ Matériel Pré-requis

Afin d'assurer le bon déroulement de l'ensemble du processus, des prérequis matériels en termes de performance serveur sont nécessaires. Dans cette chaîne d'automatisation, trois serveurs sont déployés pour des usages distincts. (Dans le cadre de ce projet, certains services ont été regroupés sur un même serveur afin d'optimiser l'utilisation des ressources.)

- Orchestrator Central (GitLab – gestion du code source)
    - RAM: 8 Go, Espace disque: 15 Go(minimum recommandé)
- GitLab Runner (exécution des pipelines) & SonarQube (analyse de la qualité du code)
    - RAM: 4 Go, Espace disque: 10 Go(minimum recommandé)
- Harbor (registre distant d'images Docker) & Serveur de déploiement
    - RAM: 8 Go, Espace disque: 15 Go(minimum recommandé)

❖ Mise en place de serveur

Orchestrator Central

L'utilisation de GitLab repose sur la containerisation : une image Docker est utilisée pour lancer un conteneur GitLab. Afin de standardiser le processus de déploiement, le démarrage de GitLab est défini dans un

fichier docker-compose.yml. Ainsi, à chaque redémarrage, il suffit de se placer dans le répertoire concerné et d'exécuter la commande :

```yaml
services:
  gitlab:
    image: gitlab/gitlab-ce:17.11.4-ce.0
    container_name: gitlab
    restart: always
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        external_url 'http://34.155.84.19:9999'
        gitlab_rails['gitlab_shell_ssh_port'] = 2222
    ports:
      - "9999:9999"
      - "2222:2222"
    volumes:
      - ./config:/etc/config/gitlab
      - ./logs:/var/log/gitlab
      - ./data:/var/opt/gitlab
```

```
root@instance-20250413-194918:/usr/local/gitlab# ls
config  data  docker-compose.yml  logs
```

Remarque:

- Remplacer external_url par votre propre adresse IP correspondante

- Pour la première connexion, le mot de passe initial du compte **root** se trouve dans le conteneur, à l'emplacement suivant: /etc/gitlab/initial_root_password.

- Pour y accéder, utilisez la commande suivante afin d'entrer dans le conteneur GitLab :sudo docker exec -it <ID_du_conteneur> bash

- Une fois à l'intérieur, affichez le mot de passe avec : cat /etc/gitlab/initial_root_password

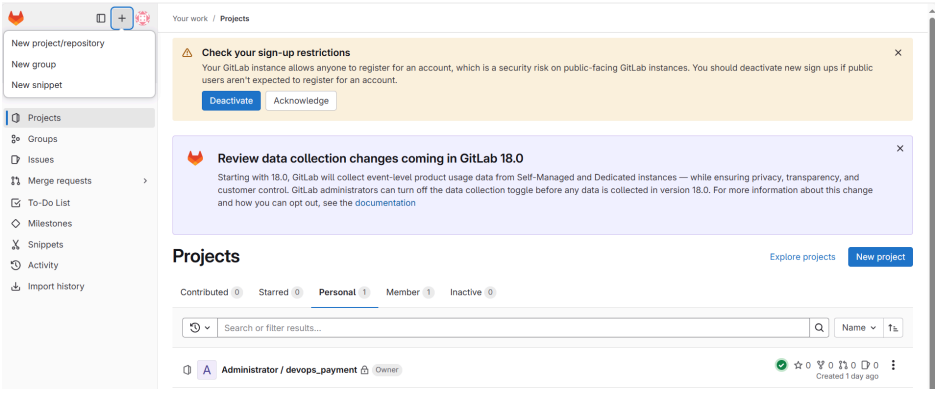
GitLab Runner（Ubuntu/Debian):

- Ajouter le repository Gitlab runner: curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash

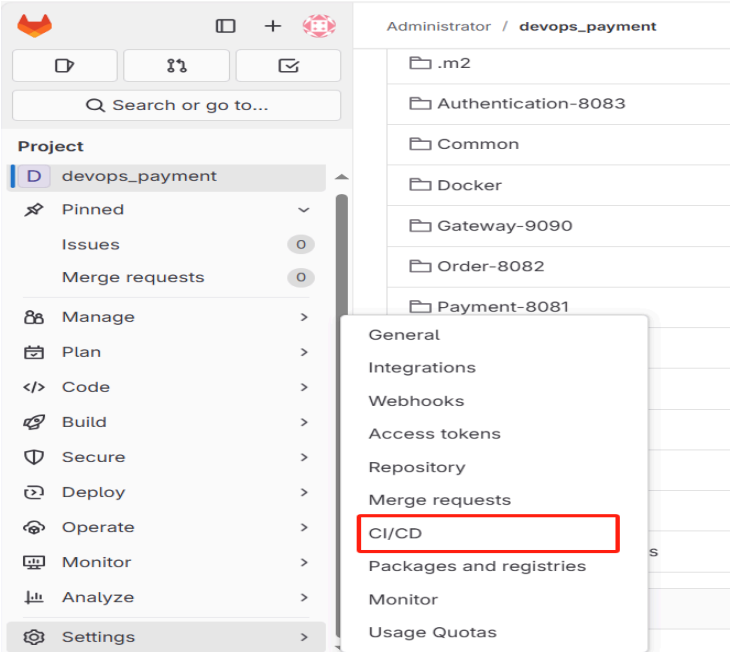
- Installer gitlab runner: sudo apt-get install gitlab-runner -y
- Inscrire Runner à Gitlab: sudo gitlab-runner register

  Dans un premier temps, vous avez besoin de générer un token sur Gitlab

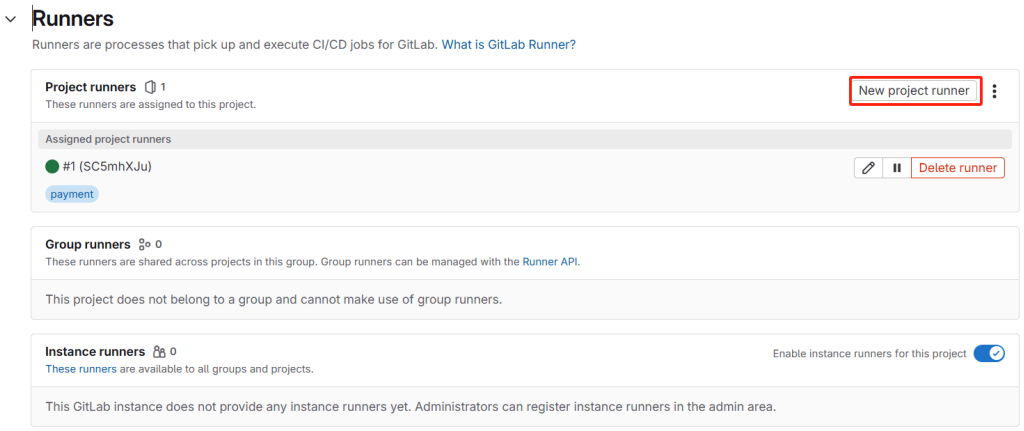
  - ☐ Créer un nouveau projet

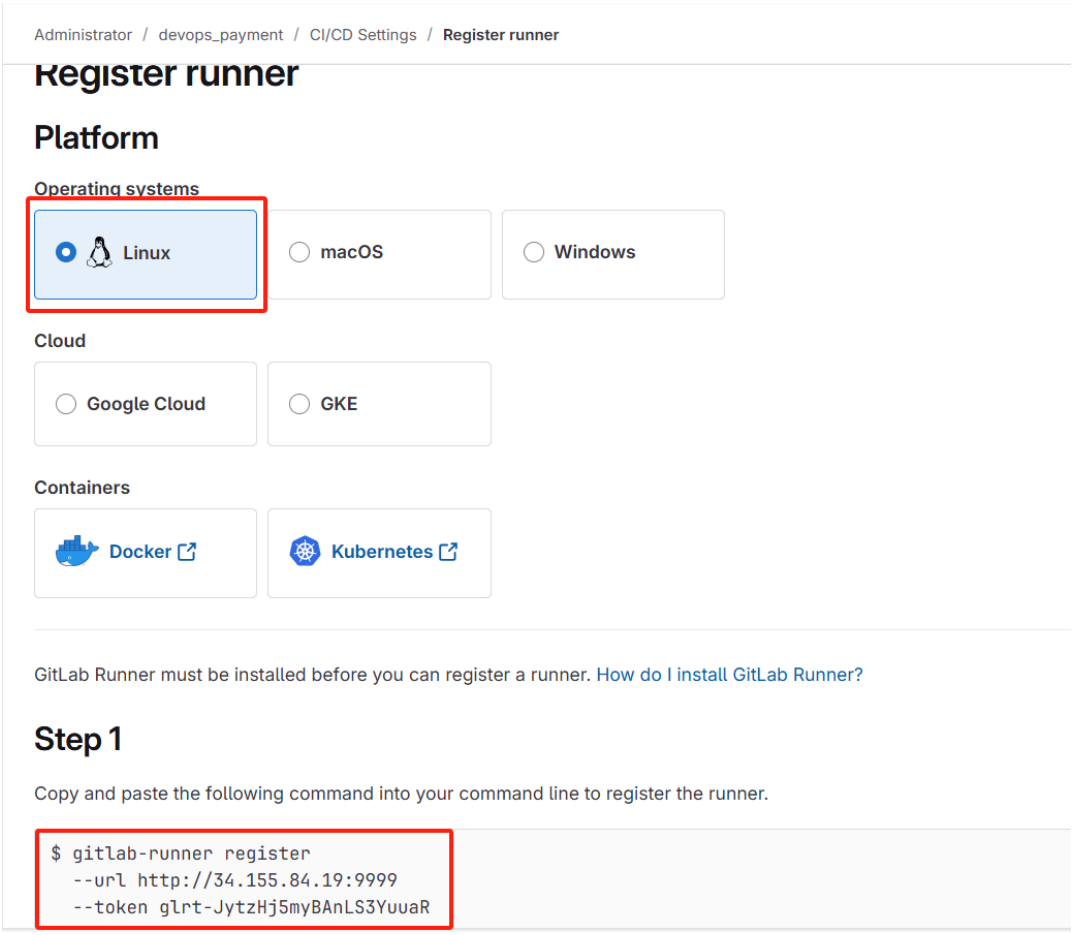


  - ☐ Entrer dans la page CICD



☐ Créer un nouveau runner





☐ Taper la commande dessus dans le serveur qui est serveur runner

☐ Par la suite, choisir l'executor 'Docker'

☐ Définir une image par défaut: maven:3.9-eclipse-temurin-17

- Jusqu'à cette étape, runner peut déjà être démarré. Au cas où, vous tapez gitlab-runner run afin de le démarrer manuellement

- Par ailleurs, il faut configurer la connexion HTTP Harbor pour que le serveur puisse push des images.

    - Accéder au fichier daemon.json : vim /etc/docker/daemon.json

    - Modifier ou insérer la configuration de la connexion HTTP (Adresse IP est l'adresse du serveur dans lequel on installe Harbor)

SonarQube:

Pour ce service, nous utilisons également Docker afin de simplifier sa mise en place. Comme pour GitLab, l'ensemble du processus est défini dans un fichier `docker-compose.yml`. Il suffit de se rendre dans le répertoire correspondant et d'exécuter la commande suivante : sudo docker compose up -d, le service SonarQube sera alors démarré automatiquement en arrière-plan.

```yaml
services:
  postgres:
    image: postgres
    container_name: postgres_db
    restart: always
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar

  sonarqube:
    image: sonarqube:9.9-community
    container_name: sonarqube
    restart: always
    depends_on:
      - postgres
    ports:
      - 9000:9000
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://postgres_db:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
```

```
pei_liu@instance-20250410-200502:/usr/local/sonarqub$ ls
docker-compose.yml
```

Harbor:

- Télécharger le package zip harbor :
  wget
  https://github.com/goharbor/harbor/releases/download/v2.9.4/harbor-online-installer-v2.9.4.tgz

- Extraire le package: tar -zxvf harbor-online-installer-v2.9.4.tgz
- Aller dans le répertoire harbor: cd harbor
- Copier le fichier harbor.yml.tmpl pour faire la modification: cp harbor.yml.tmpl harbor.yml
- Entre dans le fichier harbor.yml: vim harbor.yml
- Modifier ces configurations en fonction de votre besoin

```
# Configuration file of Harbor

# The IP address or hostname to access admin UI and registry service.
# DO NOT use localhost or 127.0.0.1, because Harbor needs to be accessed by external clients.
hostname: 34.155.208.12

# http related config
http:
  # port for http, default is 80. If https enabled, this port will redirect to https port
  port: 8080

# https related config
https:
  # https port for harbor, default is 443
  # port: 443
  # The path of cert and key files for nginx
  # certificate: /your/certificate/path
  # private_key: /your/private/key/path

# # Uncomment following will enable tls communication between all harbor components
# internal_tls:
#   # set enabled to true means internal tls is enabled
#   enabled: true
#   # put your cert and key files on dir
#   dir: /etc/harbor/tls/internal
#   # enable strong ssl ciphers (default: false)
#   strong_ssl_ciphers: false

# Uncomment external_url if you want to enable external proxy
# And when it enabled the hostname will no longer used
# external_url: https://reg.mydomain.com:8433

# The initial password of Harbor admin
# It only works in first time to install harbor
# Remember Change the admin password from UI after launching Harbor.
harbor_admin_password: Harbor12345
```

- Enfin, démarrer harbor: sudo ./install.sh

Serveur de déploiement:

- Pour que le serveur puisse pull des images sur Harbor, nous allons procéder à la même étape.
    - Accéder au fichier daemon.json : vim /etc/docker/daemon.json
    - Modifier ou insérer la configuration de la connexion HTTP (Adresse IP est l'adresse du serveur dans lequel on installe Harbor)

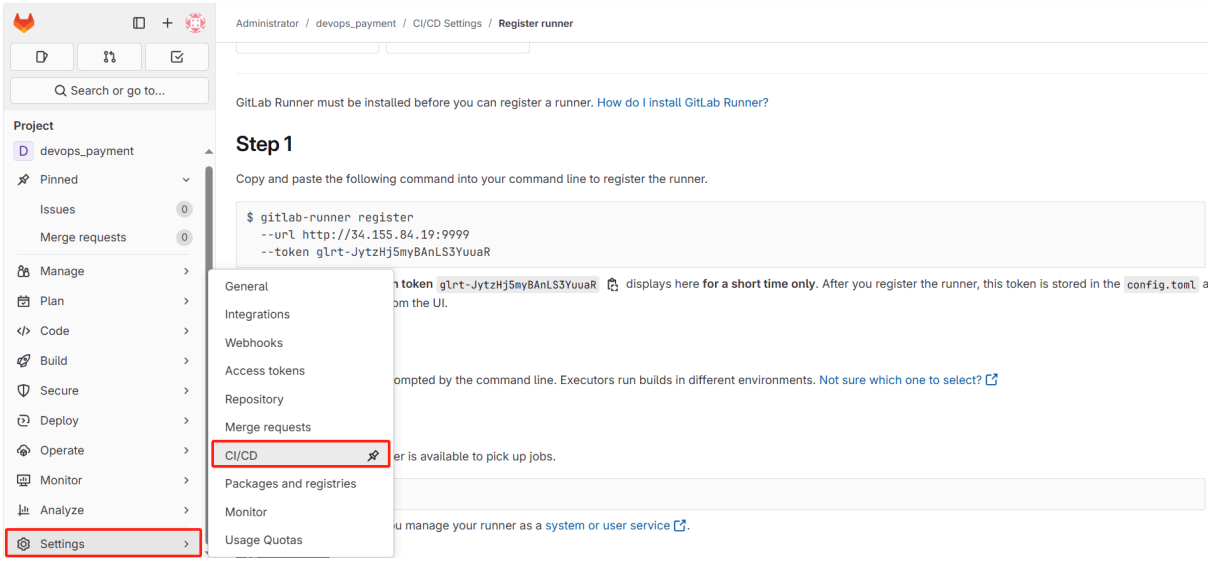
```
{
  "insecure-registries": ["34.155.208.12:8080"]
}
~
~
~
```

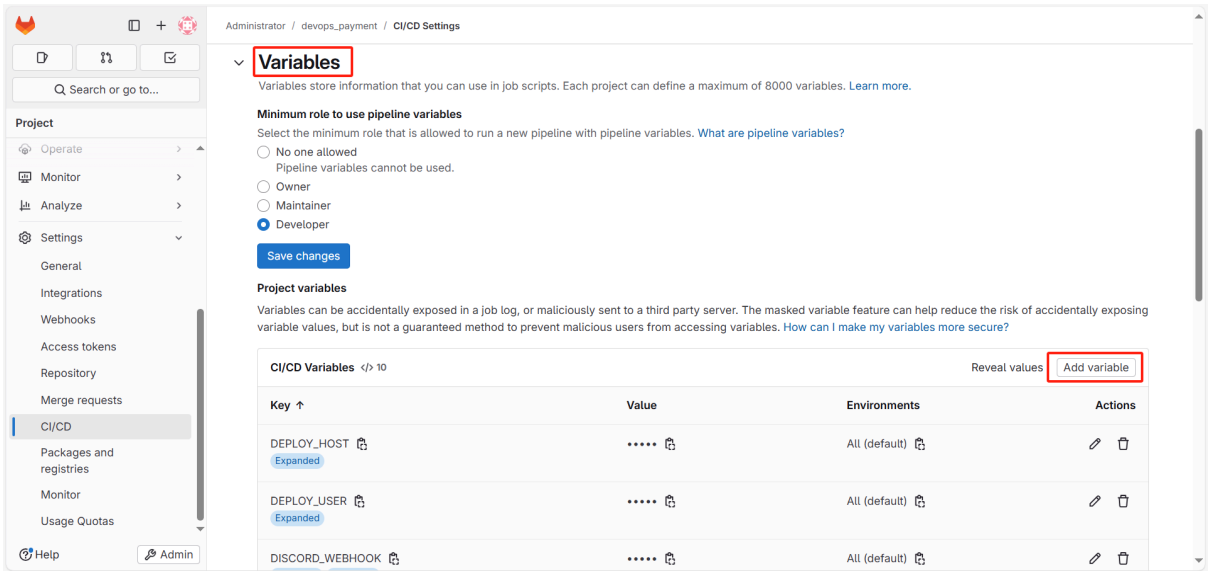
❖ Connexion SSH entre Runner et Serveur de déploiement

Serveur Runner:

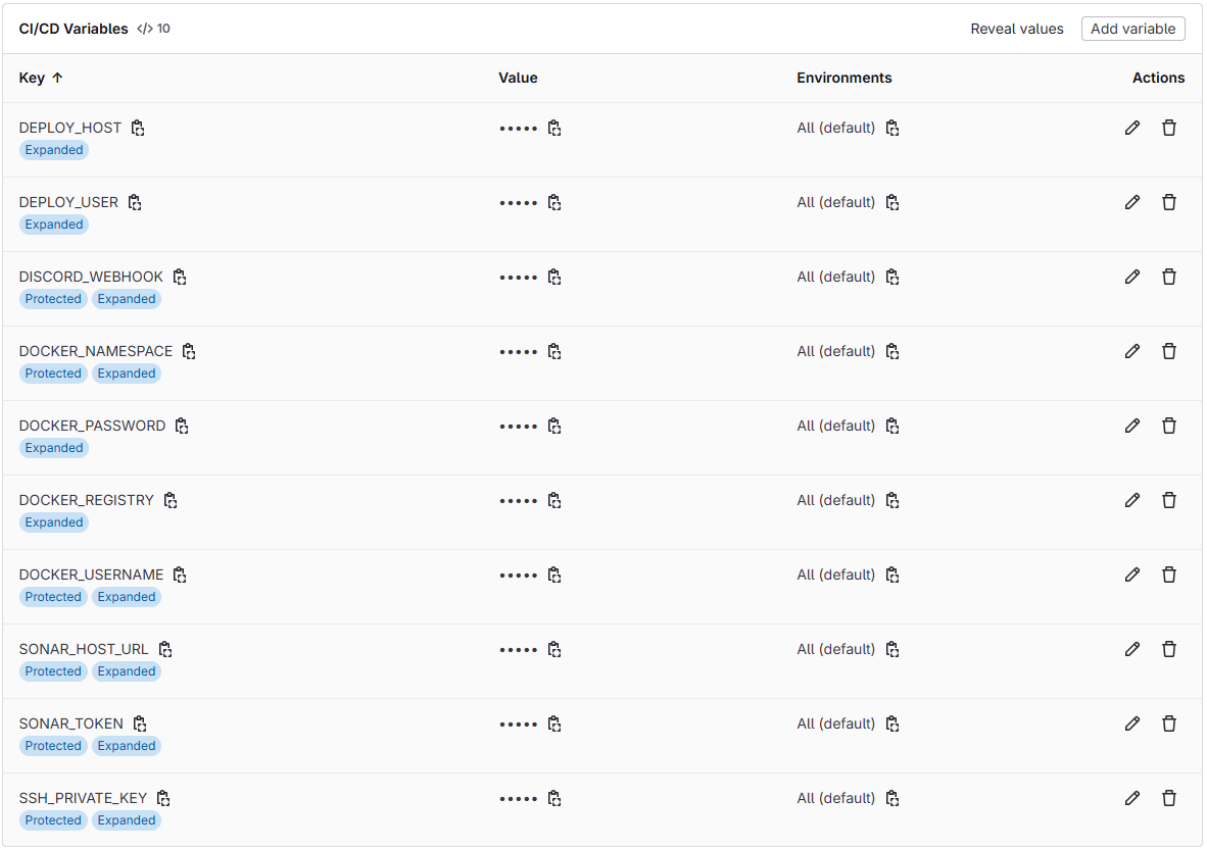
- Générer la clé secrète: ssh-keygen -t rsa -b 4096 -C "secret"
- Envoyer la clé secrète au serveur destinataire: ssh-id-copy username@adresse_ip(remplacer l'information selon votre besoin)

❖ Ajout des variables sur Gitlab pour que le script de la pipeline puisse les utiliser.





List des variables à ajouter:



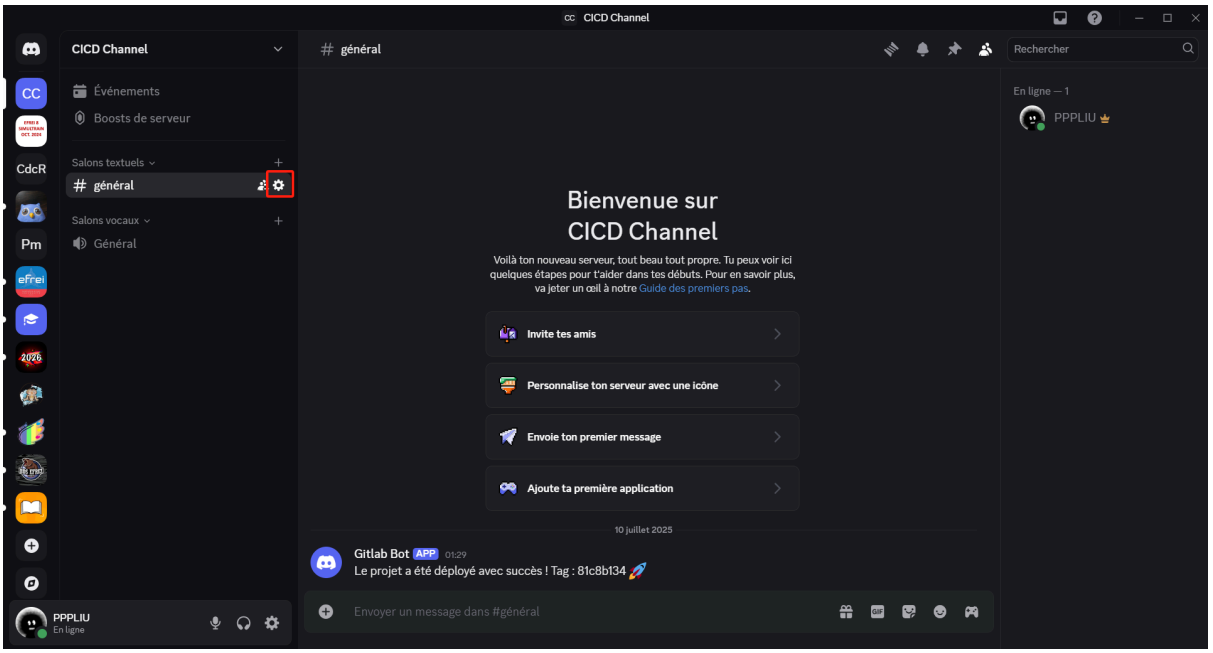
DEPLOY_HOST: Adresse IP du serveur de déploiement
DEPLOY_USER: Nom d'utilisateur du serveur
DISCORD_WEBHOOK: Lien de discord utilisé pour envoyer la notification
DOCKER_NAMESPACE: Nom de répertoire Habor
DOCKER_PASSWORD: Mot de passe pour se connecter sur Harbor
DOCKER_REGISTRY: Adresse IP du serveur où on installe Harbor
DOCKER_USERNAME: Nom d'utilsateur de Habor
SONAR_HOST_URL: Adresse IP du serveur où on installe SonarQube
SONAR_TOKEN: Token d'authentification généré sur SonarQube
SSH_PRIVATE_KEY: Clé secrète privée du serveur Runner

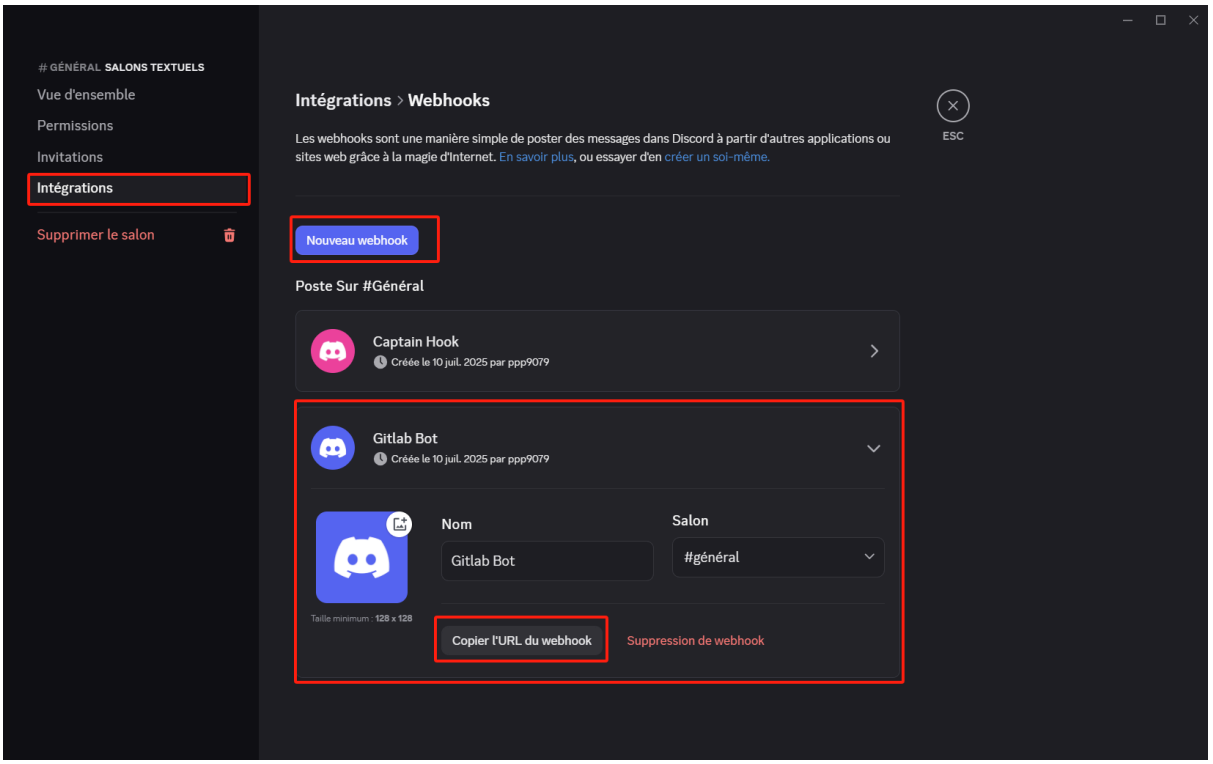
Comment trouver des informations correspondant aux variables dessus

DISCORD_WEBHOOK:

- Créer un serveur dédié au projet
- Cliquer sur le bouton Paramètre

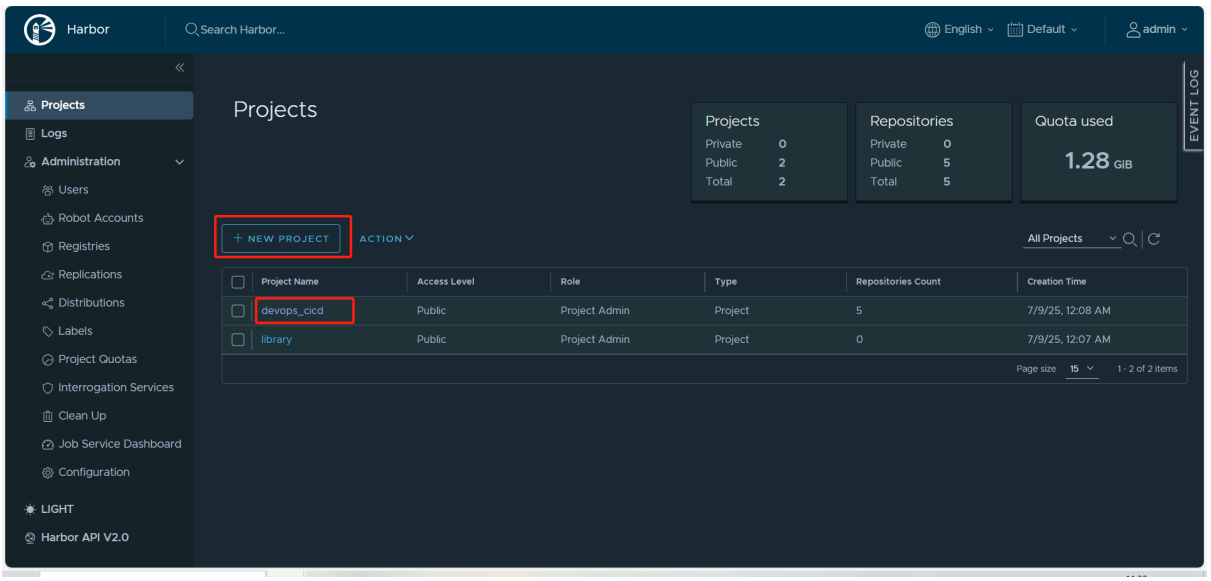
- Créer un nouveau webhook
- Copier l’url généré à la variable





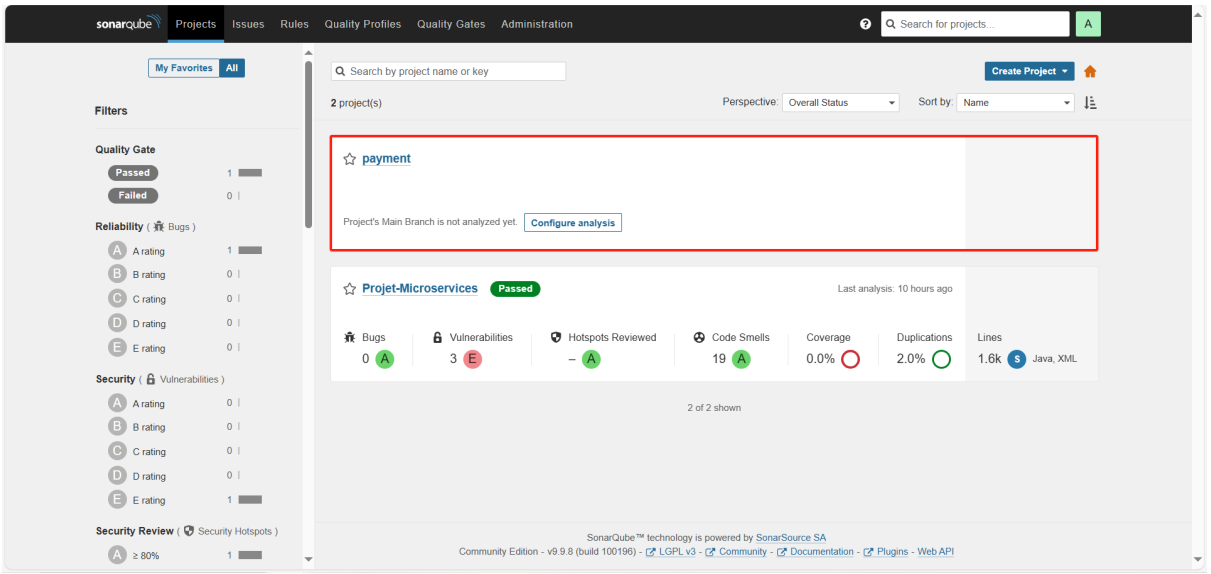
DOCKER_NAMESPACE:

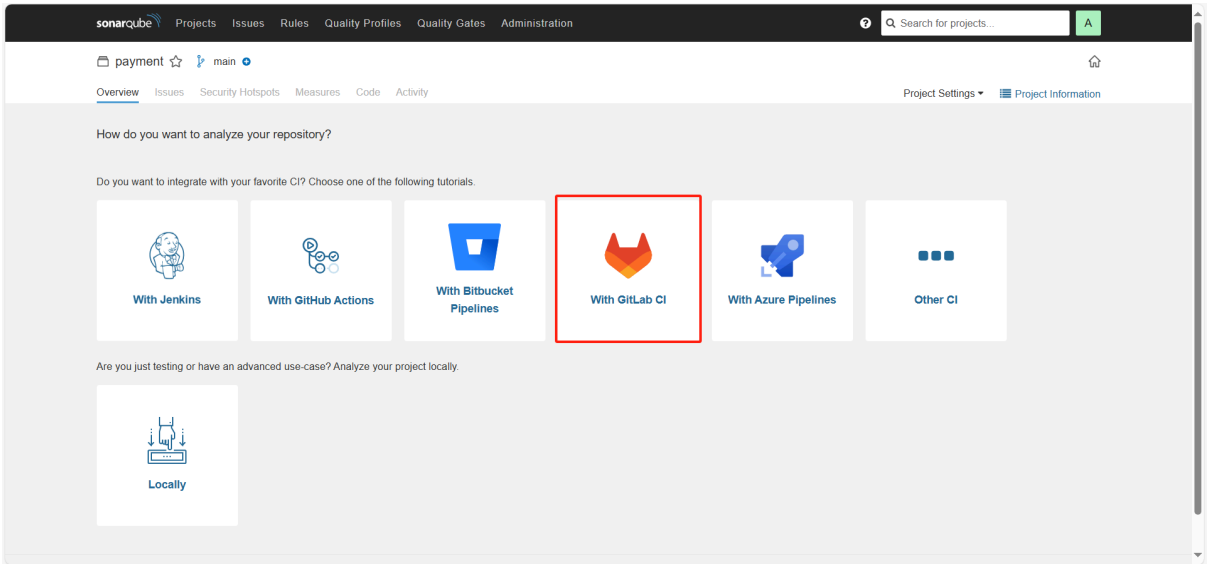
- Créer un nouveau projet en définissant le nom
- Mettre le nom du projet à la variable



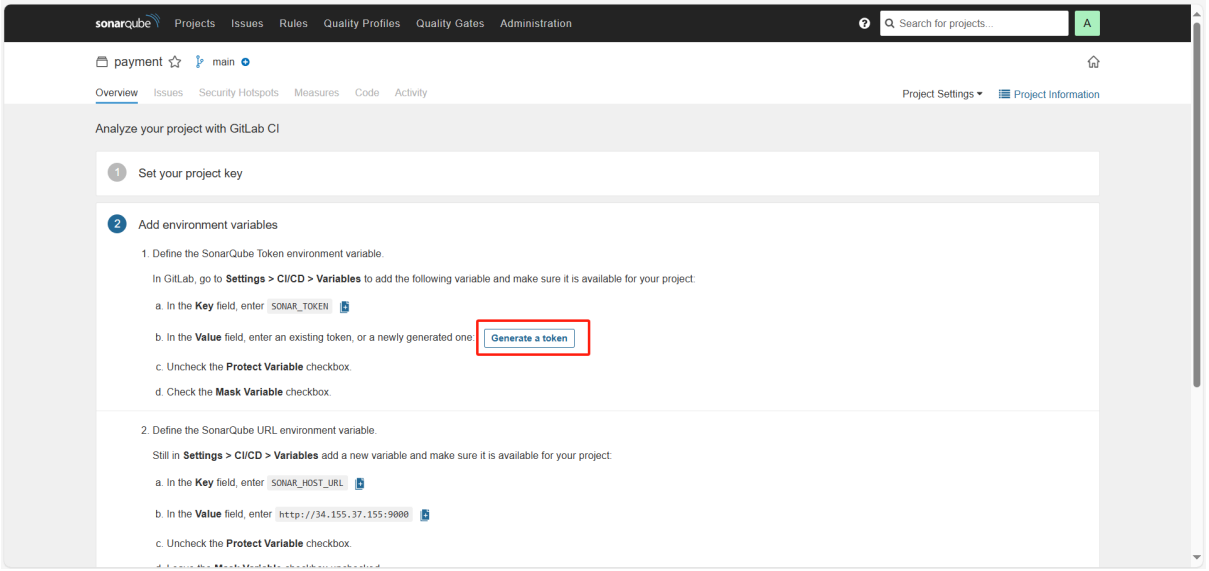
SONAR_TOKEN:

- Créer un nouveau projet
- Selectionner ‘With Gitlab CI’





- Générer le token
- Mettre ce token à la variable

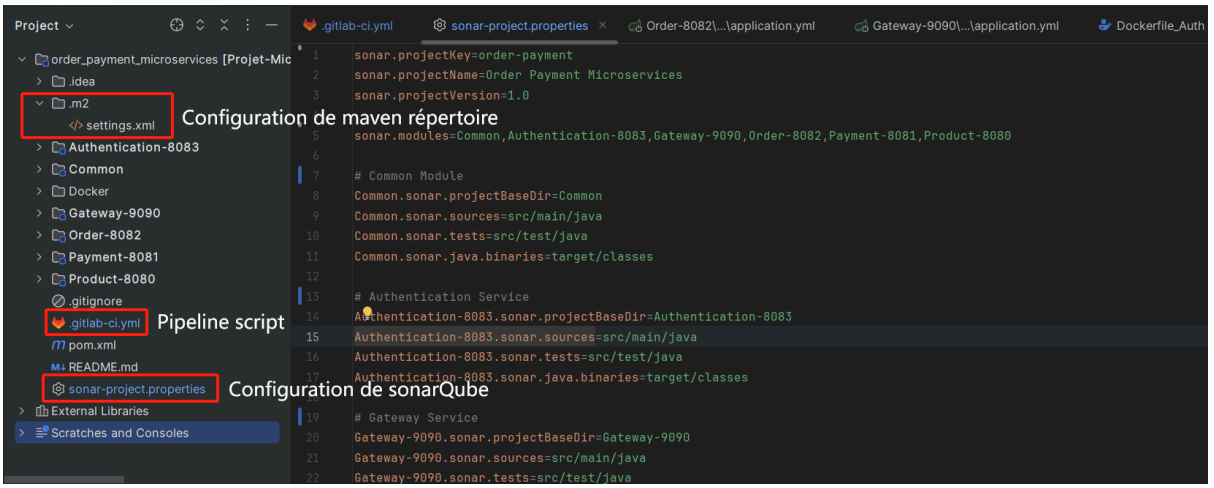


SSH_PRIVATE_KEY:

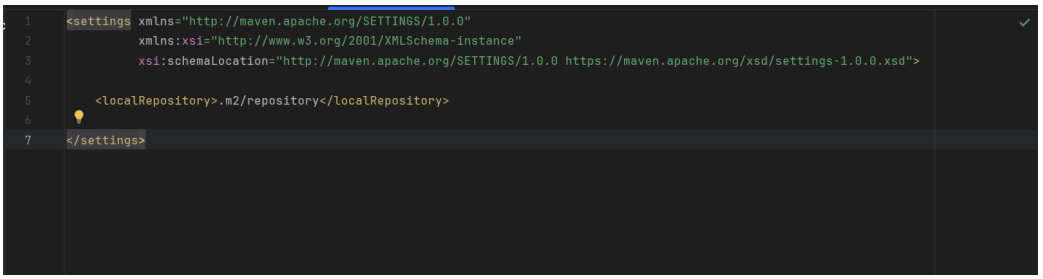
- Trouver la clé secrète privé dans le serveur Runner:
  cat ~/.ssh/id_rsa
- Copier cette clé dans la variable.

❖ **Compléter les fichiers de configuration nécessaires et construire le script de la *pipeline CI/CD***

Trois fichiers de configuration clés



- Le fichier ~/.m2/settings.xml permet à Maven de configurer l'accès aux dépôts distants afin de télécharger automatiquement les dépendances qui ne sont pas disponibles localement.

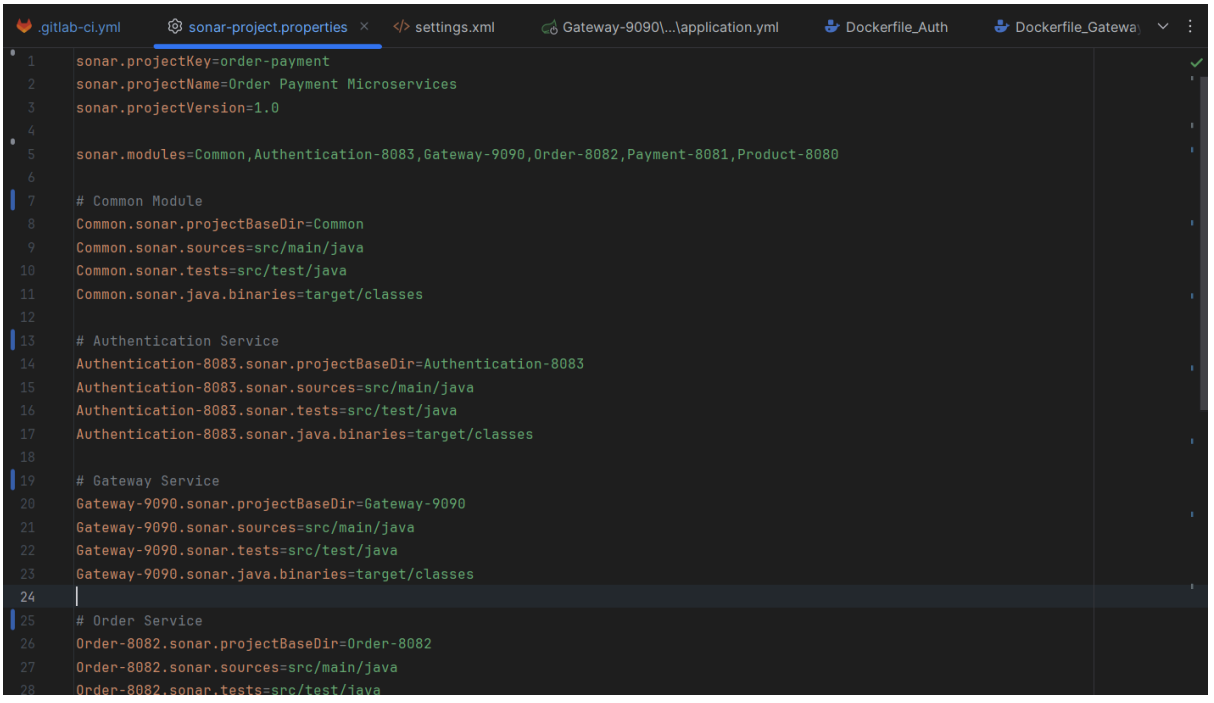


```xml
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

  <localRepository>.m2/repository</localRepository>

</settings>
```

- sonar-project.properties: utilisé pour **configurer l'analyse SonarQube** d'un projet **multi-modules**, comme un projet de microservices Java (Spring Boot, par exemple).

  Il permet à **SonarQube Scanner** de savoir :

  - quels modules analyser
  - où se trouvent les **sources Java**, les **tests**, et les **fichiers compilés**
  - comment organiser le projet pour que tout soit bien pris en compte dans le rapport de qualité



```
sonar.projectKey=order-payment
sonar.projectName=Order Payment Microservices
sonar.projectVersion=1.0

sonar.modules=Common,Authentication-8083,Gateway-9090,Order-8082,Payment-8081,Product-8080

# Common Module
Common.sonar.projectBaseDir=Common
Common.sonar.sources=src/main/java
Common.sonar.tests=src/test/java
Common.sonar.java.binaries=target/classes

# Authentication Service
Authentication-8083.sonar.projectBaseDir=Authentication-8083
Authentication-8083.sonar.sources=src/main/java
Authentication-8083.sonar.tests=src/test/java
Authentication-8083.sonar.java.binaries=target/classes

# Gateway Service
Gateway-9090.sonar.projectBaseDir=Gateway-9090
Gateway-9090.sonar.sources=src/main/java
Gateway-9090.sonar.tests=src/test/java
Gateway-9090.sonar.java.binaries=target/classes

# Order Service
Order-8082.sonar.projectBaseDir=Order-8082
Order-8082.sonar.sources=src/main/java
Order-8082.sonar.tests=src/test/java
```

.gitlab-ci.yml est le script de configuration du pipeline CI/CD. Il permet de lancer automatiquement les différentes étapes du processus (build, test, déploiement, etc.) à chaque **push** de code dans le dépôt GitLab.

- **4 grandes étapes du pipeline**

  stages:

  - build: compilation du code Java et exécution des testes unitaires

  - test: analyse SonarQube

  - docker: construction + push des images Docker

  - deploy: déploiement sur un serveur distant

- **Variables globales**

  variables:

  GIT_DEPTH: 0
  #0 signifie "cloner l'historique complet"

  MAVEN_CLI_OPTS: "--batch-mode"
  MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"
  #options Maven pour accélérer les builds

  DOCKER_TAG: $CI_COMMIT_SHORT_SHA
  #version de l'image Docker basée sur le hash du commit

- **Cache Maven local**

  cache:
    paths:
      - .m2/repository

  #Stocker les dépendances téléchargées dans le cache, et empêcher Maven de retélécharger toutes les dépendances à chaque job.

- **Build de chaque microservice (par service)**

  build_authentication:

```
 image: maven:3.9-eclipse-temurin-17
#Utilise l'image Maven pour compiler un seul service à la fois

 stage: build
 tags:
   - payment
 script:
     - mvn $MAVEN_CLI_OPTS -pl Authentication-8083 -am clean
install -DskipTests

#Produit un .jar dans target/

 artifacts:
   paths:
     - Authentication-8083/target/*.jar
   expire_in: 1 hour

#Fichier conservé avec artifacts (1h)
```

- **Analyse SonarQube**

```
test:
 stage: test
 tags:
   - payment

 script:

   - echo "Running SonarQube analysis"

   - mvn $MAVEN_CLI_OPTS clean verify  sonar:sonar
   # clean : nettoie les fichiers compilés
   #verify : compile + teste + valide le build
   #sonar:sonar : déclenche le plugin SonarQube pour analyser le
code avec les informations d'authentification dessous

   -Dsonar.projectKey=order-payment

   -Dsonar.host.url=$SONAR_HOST_URL

   -Dsonar.login=$SONAR_TOKEN


 only: #Valable uniquement pour ces trois cas
   - merge_requests
   - main
   - develop
```

- **Génération des images docker et les envoyer à Harbor**

```
docker_build_push:
 image: docker:24.0.7
#On utilise docker image dans cette étape, car on a besoin la
commande docker afin de push les images généré à Harbor
 stage: docker
 tags:
```

```
  - payment
 services:
  - name: docker:24.0.7-dind
    command: ["--insecure-registry=34.155.208.12:8080"]
   #Configurer la connexion HTTP à Harbor
 dependencies:
 #Récupérer les artifacts sauvegardés dans la phrase précédente
  - build_authentication
  - build_gateway
  - build_order
  - build_payment
  - build_product
 variables:
  DOCKER_HOST: tcp://docker:2375
  # indique au client Docker d'utiliser le démon Docker du service
  DOCKER_TLS_CERTDIR: ""
  # vide : désactive TLS, ce qui est nécessaire pour que ça marche en
HTTP.
 script:
  - echo "Logging into Docker registry"
  - echo "$DOCKER_PASSWORD" | docker login -u
"$DOCKER_USERNAME" --password-stdin $DOCKER_REGISTRY

   #Se connecter à Harbar avec l'identifiant et le mot de passe

  - echo "Building & pushing images with tag $DOCKER_TAG"
  - |
    for dir in Authentication-8083 Gateway-9090 Order-8082
Payment-8081 Product-8080; do
      SERVICE=$(echo $dir | cut -d'-' -f1 | sed 's/Authentication/Auth/')
      IMAGE_NAME=$(echo $dir | tr '[:upper:]' '[:lower:]')
      echo "Building $IMAGE_NAME using
Docker/Dockerfile_${SERVICE}"
      docker build -t
$DOCKER_REGISTRY/$DOCKER_NAMESPACE/$IMAGE_NAME:$DO
CKER_TAG \
        -f Docker/Dockerfile_${SERVICE} $dir
```

```
      docker push
$DOCKER_REGISTRY/$DOCKER_NAMESPACE/$IMAGE_NAME:$DO
CKER_TAG
    done
#Parcourir tous les services pour générer l'image avec le fichier
DockerFile correspondant, après l'envoyer dans le dépôt de Harbor
 only: # Valable seulement pour la branche main
  - main
```

- **Déploiment**

```
deploy_to_server:
 stage: deploy
 tags:
  - payment
 before_script:
  - apt-get update && apt-get install -y openssh-client
  #installe le client SSH dans le conteneur
  - mkdir -p ~/.ssh
  #Crée le dossier .ssh
  - echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
  #Injecte la clé privée dans ~/.ssh/id_rsa
  - chmod 600 ~/.ssh/id_rsa
  #Restreint les droits d'accès
  - ssh-keyscan -H $DEPLOY_HOST >> ~/.ssh/known_hosts
  #Ajouter l'empreinte SSH du serveur distant dans known_hosts pour
éviter le prompt yes/no lors de la première connexion.
 script:
  - echo "Generating .env"
  - echo "IMAGE_TAG=$CI_COMMIT_SHORT_SHA" > Docker/.env
  #Créer un fichier .env contenant le tag de l'image à déployer
  - echo "Sending files to server"
  - ssh $DEPLOY_USER@$DEPLOY_HOST "mkdir -p
/home/$DEPLOY_USER/app"
  #Créer le répertoire au serveur destinataire
```

```
  - scp Docker/docker-compose.yml
$DEPLOY_USER@$DEPLOY_HOST:/home/$DEPLOY_USER/app/
  - scp Docker/.env
$DEPLOY_USER@$DEPLOY_HOST:/home/$DEPLOY_USER/app/
  #Envoyer le fichier .env et docker-compose.yml au serveur de
déploiement
  - echo "Remote docker-compose up"
  - |
    ssh $DEPLOY_USER@$DEPLOY_HOST <<EOF
      cd /home/$DEPLOY_USER/app
      #Entrer dans le répertoire correspondant du serveur de déploiement
      docker compose pull
      #Récupérer les images définies dans docker-compose.yml
      docker compose up -d
      #Générer les conteneurs et les démarrer
    EOF
 only:
  - main
```

- **Notification**

```
discord_notify_success:
 image: curlimages/curl:latest
 #Utilise l'image curlimages/curl:latest pour envoyer des requêtes HTTP
 stage: deploy
 tags:
  - payment
 needs:
  - job: deploy_to_server
  #Cela garantit que cette notification ne s'exécute qu'après le
déploiement, et seulement si ce dernier a été tenté.
 script:
  - >
    curl -H "Content-Type: application/json" \
         -X POST \
```

```
        -d "{\"content\": \"Le projet a été déployé avec succès ! Tag :
$CI_COMMIT_SHORT_SHA\"}" \
        $DISCORD_WEBHOOK
#Utiliser Curl pour envoyer une requête POST vers ton webhook
Discord.

 only:
   - main
 when: on_success
#Il s'exécute uniquement si le job deploy_to_server a réussi
(on_success).
```

❖ **Déclenchement de CI/CD**

Pour déclencher le processus de CI/CD, il suffit de push la nouvelle version à Gitlab, par la suite, Runner va exécuter le pipeline automatiquement. A la fin du pipeline, une notification sera envoyée dans votre chaîne Discord pour vous informer de la réussite ou l'échec d'exécution.

```
D:\Efrei\S7\OO System\order_payment_microservices>git add .

D:\Efrei\S7\OO System\order_payment_microservices>git commit -m ".gitlab-ci updated"
[develop 00eb200] .gitlab-ci updated
 2 files changed, 9 insertions(+), 11 deletions(-)

D:\Efrei\S7\OO System\order_payment_microservices>git push origin main
```