IUT Valence
Université Grenoble Alpes

Kaan Topkaya, Anthony Saillard, Théo Riotte, Lucy Dumarey
Groupe 1.7

Intégration Continue

SAE - Qui Fait Quoi Quand

Sommaire

I - Objectif

L'intégration continue mise en place pour ce projet vise à automatiser la vérification du code et à garantir sa qualité à chaque *commit* ou *push* sur le dépôt GitLab. Elle couvre les tests unitaires, l'analyse statique et le *lint* des différentes technologies utilisées, soit PHP et Symfony vérifiés avec PHPStan et PHPUnit, et VueJS et Quasar vérifiés avec un ESLint.

II - Outils et Technologies

Pour automatiser la vérification du code sur GitLab, nous avons mis en place un *pipeline CI*, via les paramètres de GitLab CI/CD et basé sur *Docker-in-Docker*, afin de déclencher la vérification, configurer des "*jobs*", vérifier et installer des dépendances, lancer l'analyse avec PHPStan et les tests avec PHPUnit, et afficher les résultats de ces vérifications, à chaque *push* et *merge request*.

A - Les déclencheurs

Le pipeline s'exécute automatiquement lors de *push* sur chaque branche créée depuis la branche *develop*, sur la branche *develop* et sur la branche *main*. Il en est de même lors de la création d'une *merge request*.

Il est également possible, sur demande manuelle, de lancer un nouveau pipeline.

B - La configuration des *Jobs*

Concernant la configuration, le *pipeline* est lancé via un *runner* local. La configuration de ce dernier se trouve dans le répertoire "./gitlab-runner/config". Le fichier de configuration ressemble à :

[[runners]]

name = "runner-linux01"

url = "https://gitlab.iut-valence.fr" <- remplacer par l'url nécessaire si elle n'est bonne

token = "glrt-xxxxxxxxxxxxxxxxxxxx" <- remplacer par le token créé dans les paramètres GitLab CI/CD.

executor = "docker"

[runners.docker]

tls_verify = true

privileged = true

disable_entrypoint_overwrite = false

oom_kill_disable = false

disable_cache = false

volumes = [

"/var/run/docker.sock:/var/run/docker.sock",

"/cache",

]

shm_size = 0

allowed_pull_policies = ["always", "if-not-present"]

pull_policy = ["if-not-present"]

Il faut également ajouter et compléter le fichier de configuration ".gitlab-ci.yml", se trouvant à la racine du projet.

La configuration complète du runner est expliquée dans la documentation technique également, dans la partie B.2. nommé "Ajout de *pipelines* et de *runners*".

1. La configuration du ".gitlab-ci"

La configuration dans le fichier ".gitlab-ci", nous avons une configuration générale, déterminant les étapes (*stages*) qui seront lancés, le service et les

variables utilisés, le stockage du cache et la commande lancée avant chaque script, permettant ici de se connecter au répertoire disponible sur Docker Hub.

```
image: docker:latest

stages:
  - build
  - test
  - deploy

services:
  - docker:dind

variables:
  DOCKER_IMAGE: lucy135/qfqq
  DOCKER_IMAGE_TAG: CI-${CI_PIPELINE_ID}
  DOCKER_BUILDKIT: 0

cache:
  paths:
    - vendor/
    - node_modules/

before_script:
   - echo "$DOCKER_HUB_PASSWORD" | docker login -u "$CI_REGISTRY_USER" --password-stdin
```

2. La construction de l’image Docker

L'étape de construction, soit "*build*", se nomme "build" et permet de construire le projet et de stocker l'image créée sur le répertoire Docker Hub, sur lequel on pousse l'image.

```
build:
  stage: build
  script:
    - echo "Build docker image ${DOCKER_IMAGE}"
    - docker image build ./frontend/ -t $DOCKER_IMAGE:$DOCKER_IMAGE_TAG
    - docker image push $DOCKER_IMAGE:$DOCKER_IMAGE_TAG
```

3. L'analyse statique PHP : PHPStan

L'étape d'analyse statique PHP, faite à partir de PHPStan, se nomme "test:lint" et permet d'exécuter dans le bash le script "./scripts/lint.sh", seulement si ce dernier existe.

```
test:lint:
  stage: test
  script:
    - echo "Running linter on test image"
    - if [ -f ./scripts/lint.sh ]; then echo "File exists"; else echo "File does not exist"; fi
    - chmod +x ./scripts/lint.sh
    - sh ./scripts/lint.sh
```

Ce script cité efface le cache de PHPSstan dans le docker, vérifie la version de PHPStan afin de vérifier que cette librairie existe et/ou que les conteneurs du projet sont en fonctionnement. Enfin, il démarre l'analyse, de niveau huit sur neuf, sur les tests et le *backend*.

4. Les tests unitaires PHP : PHPUnit

L'étape de tests unitaires PHP, faite à partir de PHPUnit, se nomme "test:unit" et permet d'exécuter dans le bash le script "./scripts/run-tests.sh".

```
test:unit:
  stage: test
  script:
    - echo "Running test image"
    - sh ./scripts/run-tests.sh
  coverage: '/(?i)total.*? (100(?:\.0+)?\%|[1-9]?\d(?:\.\d+)?\%)$/'
```

Ce script cité lance l'analyse PHPUnit, configurée via le fichier phpunit.xml.

5. Le Déploiement de l'application

Dans le fichier .gitlab-ci, nous avons une étape permettant de déployer une image sur Docker Hub, afin d'obtenir une nouvelle image à jour, dite "latest".

```
deploy:
  stage: deploy
  script:
    - echo "Deployment..."
    - docker pull $DOCKER_IMAGE:latest
  only:
    - tags
```

C - Résultats du pipeline

En cas d'échec d'une étape, lors de l'exécution du pipeline, ce dernier s'arrête immédiatement. Pour être informé(e) du résultat des pipelines, nous recevons un mail informant de la réussite ou de l'échec du pipeline concerné. De plus, sur GitLab, il est possible d'obtenir les détails de l'erreur en cliquant sur l'étape erronée.

Conclusion

Grâce à ce pipeline CI, basé sur Docker, nous nous assurons que le code est de qualité, fonctionnel et respecte les standards avant d'être intégré dans la branche principale.