# Contents

# Intégration continue avec GitLab

*Nicolas, BERTRAND*

*Le-Point-Technique, Janvier/2022*

**abstract**: GitLab est une plateforme de développement open source dédiée à la gestion de projet informatique. De la gestion de version du code source, en passant par son tableau de bord qui permet de suivre les tâches en cours ou encore par la définition précise des rôles de chaque membre de l'équipe, GitLab offre un grand nombre de fonctionnalités qui facilitent le travail collaboratif. Dans ce tutoriel, je vais tenter d'expliquer quelques notions techniques et fournir des extraits de code en me concentrant sur l'aspect intégration continue. Pour ce faire, je vais utiliser la plateforme DevOps accessible en ligne à l'adresse About GitLab. L'objectif est de créer un pipeline d'intégration continue contenant six étapes d'automatisation, à savoir, l'étape de compilation, des tests unitaires, de la couverture du code par les tests, de la qualité du code, de la création de package pour terminer avec la création d'image pour conteneuriser nos applications.

**keywords**: pipeline CI, intégration continue, GitLab, GitLab Runner, build, unit test, code coverage, code quality, Spring, Maven, Docker, Kaniko

## Introduction

Afin de garantir une certaine compréhension, je vais commencer par décrire quelques concepts, en fournissant la définition des mots clés utilisés sur la plateforme GitLab. Je vais poursuivre avec un mot sur l'installation de l'outil et sur la création d'un nouveau projet. Enfin, je vais expliquer comment mettre en place le pipeline en fournissant d'abord un exemple basique, puis des exemples plus complets de manière à créer notre pipeline d'intégration continue.

## Concepts clés

Dans cette partie, je vais définir le vocabulaire employé pour ce tutoriel d'un point de vue utilisateur de la solution GitLab.

### Intégration continue

L'intégration continue est une pratique qui consiste à mettre en place un ensemble de vérifications qui se déclenchent automatiquement lorsque les développeurs envoient les modifications apportées au code source, lui même stocké dans un dépôt Git, dans notre cas sur un serveur GitLab. L'exécution de scripts automatiques permet de réduire le risque d'introduction de nouveaux bugs dans l'application et de garantir que les modifications passent tous les tests et respectent les différentes normes qualitatives exigées pour un projet.

### Job

Un *job* est une tâche regroupant un ensemble de commandes à exécuter.

### Job Artifacts

L'exécution d'un job peut produire une archive, un fichier, un répertoire. Ce sont des artefacts que l'on peut télécharger ou visualiser en utilisant l'interface utilisateur de GitLab.

### Pipeline

Représente le composant de plus haut niveau. Il est composé de jobs (tâches), qui définissent ce qu'il faut faire, et de *stages* (étapes) qui définissent quand les tâches qui donnent le timing d'exécution des dites tâches. Dans notre cas, les six stages que nous allons mettre en place sont 'build', 'unit-test', 'coverage', 'quality', 'package' et 'docker'.

### Gitlab Runners

Gitlab Runner est une application qui prend en charge l'exécution automatique des builds, tests et différents scripts avant d'intégrer le code source au dépôt et d'envoyer les rapports d'exécutions à GitLab. Ce sont des processus qui récupèrent et exécutent les jobs des pipelines pour GitLab. Il existe deux types de runner, les *shared runners*, qui sont mis à notre disposition à travers la plateforme et les *specific runners* qui sont spécifiques à un projet et peuvent être installés sur nos machines.

### Gitlab Server

Le serveur GitLab est un serveur web qui fournit à l'utilisateur des informations sur les dépôts git hébergés dans son espace. Il a essentiellement deux fonctions. Il contient le dépôt git et il contrôle les runners.

## Installation

Je ne vais pas expliquer comment installer GitLab dans cette présentation car vous trouverez toutes les informations nécessaires sur ce sujet dans la documentation officielle (voir Install GitLab). De plus, une inscription à l'offre gratuite de GitLab permet de profiter des fonctionnalités de la solution SaaS sans configuration technique ni téléchargement ou installation. Cela dit, il est important de préciser que tous les nouveaux inscrits, à compter du 17 Mai 2021 doivent fournir une carte de paiement valide afin d'utiliser les shared runners de GitLab.com (voir How to prevent crypto mining abuse on GitLab.com SaaS). L'objectif de cette décision est de mettre fin aux consommations abusives des minutes gratuites de pipeline offertes par GitLab pour miner des crypto-monnaies. Si vous ne pouvez pas fournir ces informations, vous avez la possibilité d'installer un runner sur

votre machine (voir Install GitLab Runner). Dans le paragraphe suivant je vais vous montrer comment installer, paramétrer et utiliser un runner spécifique.

**Specific Runners**

Par défaut, le pipeline de GitLab utilise les shared runners pour exécuter les jobs. Vous trouverez ces informations en naviguant dans le menu *Settings* de votre projet, puis *CI/CD*, et enfin *Runners* (voir *Figure 1* ci-dessous).

image

*Figure 1: Les runners du projet*

À gauche, nous pouvons voir la colonne *Specific runners*, dans laquelle nous trouvons des liens vers la procédure d'installation suivant différents environnements (voir *Figure 2* ci-dessous).

image

*Figure 2: Installer un runner spécifique*

Une fois installé en local, nous devons enregistrer le runner pour notre projet (voir Registering runners). Nous pouvons réaliser cette tâche en mode interactif ou one-line. Voici les différentes étapes communes à tous les environnements en mode interactif :

1. Exécutez la commande (suivant votre os, voir la doc) : sudo gitlab-runner register
2. Entrez l'URL de l'instance GitLab (voir colonne Specific runners) : https://gitlab.com/
3. Entrez le jeton fourni pour enregistrer le runner (voir colonne Specific runners) : uytryuBN76545fgcv
4. Entrez une description pour votre runner : myLocalRunner
5. Entrez un ou plusieurs tags pour votre runner
6. Entrez le runner executor : docker
7. Si vous avez entré docker à l'étape précédente une image par défaut doit être spécifiée : maven:latest

Après avoir désactivé l'option shared runners de la page de configuration des runners de notre projet nous devrions voir le runner spécifique de notre machine disponible et actif pour exécuter les jobs de notre pipeline (voir *Figure 3* ci-dessous).

image

*Figure 3: Notre runner spécifique est disponible*

L'action d'enregistrer un runner spécifique pour notre projet crée un fichier de configuration local appelé *config.toml* sur notre machine (dans le répertoire /etc/gitlab-runner pour un environnement linux). C'est dans ce fichier que l'on retrouve les informations transmises lors de l'enregistrement de notre runner.

Vous trouverez ci-dessous le fichier config.toml de ma configuration qui contient un peu plus d'option que celles générées par défaut, en particulier pour le paramètre volumes des options runners.docker. Je vous laisse découvrir la documentation Configuring GitLab Runner et GitLab Runner commands pour approfondir les informations délivrées dans cette parenthèse.

```
concurrent = 1
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "myLocalRunner"
  url = "https://gitlab.com/"
  token = "uytryuBN76545fgcv"
  executor = "docker"
  builds_dir = "/tmp/builds"
  [runners.custom_build_dir]
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    image = "maven:latest"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock", "/tmp/builds:/tmp/buil
    shm_size = 0
```

## Création d'un nouveau projet

Une fois les étapes d'inscription et d'installation franchies, nous pouvons créer un nouveau projet (voir *Figure 4* ci-dessous). Remplissez les champs avec les informations de votre choix.

image

*Figure 4: Page de création d'un nouveau projet*

Gitlab crée un repository vide (voir *Figure 5* ci-dessous) et nous indique les commandes git à exécuter (voir *Figure 6* ci-dessous) afin de poursuivre la création du projet.

image

*Figure 5: Page d'accueil du nouveau projet vide'*

image

*Figure 6: Commandes git à exécuter*

## Création du pipeline

Nous allons maintenant construire le pipeline et mettre en place les différents jobs. Pour continuer cette présentation, je vais utiliser le projet accessible à cette adresse MedHead (voir *Figure 7* ci-dessous). Tous les extraits de code et figures qui vont suivre sont tirés de ce projet, réalisé dans le cadre d'une formation qualifiante de la plateforme OpenClassrooms. Il s'agit de plusieurs applications Spring Boot, qui utilisent l'outil Maven et le langage Java.

image

*Figure 7: Repository du projet MedHead*

### .gitlab-ci.yml, exemple simple

Afin de paramétrer un pipeline sur la plateforme GitLab, nous devons commencer par créer un fichier .gitlab-ci.yml à la racine de notre repository. Ce fichier est organisé autour de deux notions importantes, les *stages* et les jobs. Les stages indiquent le nom et l'ordre d'exécution des jobs, qui sont eux-mêmes attachés à un stage. L'extrait de code ci-dessous montre une écriture minimale du fichier.

```yaml
stages:
  - build
  - test

build-job:
    stage: build
    script:
        - echo "Le projet build..."

test-job:
    stage: test
    script:
        - echo "Les tests s'exécutent..."
```

Maintenant que le fichier est créé nous pouvons effectuer un commit, cette action va démarrer l'exécution automatique du pipeline, que nous pouvons suivre dans l'onglet *Pipelines* de l'interface (voir *Figure 8* ci-dessous). Le pipeline peut avoir différents états, *running* quand il est en cours d'exécution puis, *passed* ou *failed*, qui indiquent respectivement que l'exécution s'est déroulée correctement ou, au contraire, qu'elle est stoppée car des erreurs ont été trouvées.

image

*Figure 8: Exécution du pipeline*

### .gitlab-ci.yml, build et tests unitaires

Voyons maintenant un extrait du code permettant de compiler puis d'exécuter les tests de l'application *emergency* du projet MedHead.

```
image: maven:latest # J'ajoute l'image docker que le runner va utiliser pour exécuter mes s

stages:
  - build
  - unit-test

build-ms-emergency:
  stage: build
  script:
    - cd emergency
    - ./mvnw compile # Commande maven pour compiler le code source du projet

unit-test-ms-emergency:
  stage: unit-test
  script:
    - cd emergency
    - ./mvnw surefire-report:report # Crée un rapport d'exécution des tests au format html
  artifacts:
    when: always
    # paths permet de sauvegarder les artefacts générés pendant l'exécution du script sur l
    # et de les retrouver dans l'onglet browse du job ou le bouton download du pipeline
    paths:
      - emergency/target/site/surefire-report.html
    # reports:junit permet de récupérer les artefacts TEST-com.ocr.medhead.emergency.*.xml
    #afin d'intégrer les rapports dans l'onglet test des détails d'un job
    reports:
      junit:
        - emergency/target/surefire-reports/TEST-*.xml
```

Dans cet extrait de code, le mot clé *image* indique quelle image docker doit être employée par le *runner* pour exécuter les jobs.

Une fois le job *build-ms-emergency* traité par le pipeline, nous pouvons visualiser le résultat du build en naviguant dans l'onglet *Jobs* et en le sélectionnant dans la liste (voir *Figure 9* ci-dessous).

image

*Figure 9: Visualisation du résultat du build*

Le pipeline poursuit son exécution avec le job nommé *unit-test-ms-emergency*. Ce job va nous permettre d'exécuter les tests unitaires développés pour l'application

*emergency*. En complément, nous allons générer un rapport que nous pourrons sauvegarder grâce à l'utilisation du mot clé *artifacts*. Nous spécifierons la fréquence de création de ce rapport avec *when* et le sauvegarderons, dans un format html avec *paths* pour le consulter ou le télécharger ultérieurement (voir *Figure 10* ci-dessous), ainsi que dans un format xml avec *reports:junit* pour qu'il soit intégré dans l'interface utilisateur de Gitlab (voir *Figure 11* ci-dessous).

image

*Figure 10: Télécharger ou visualiser un rapport d'exécution des tests unitaires*

image

*Figure 11: Intégration du rapport d'exécution des tests unitaires dans GitLab*

### .gitlab-ci.yml, Code Coverage

Dans l'extrait de code suivant, j'ai ajouté le stage *coverage* et le job *coverage-ms-emergency*. Cela va nous permettre de générer automatiquement un rapport de couverture du code par les tests. Comme pour le job précédent celui-ci est généré puis sauvegardé afin d'être consulté ou téléchargé ultérieurement (voir *Figure 12* ci-dessous). L'intégration des résultats du rapport dans l'interface GitlLab n'est pas abordée dans ce tutoriel. Si vous le souhaitez vous trouverez les informations nécessaires pour activer cette visualisation à cette adresse Test coverage visualization.

```
image: maven:latest

stages:
  - build
  - unit-test
  - coverage

build-ms-emergency:
  ...

unit-test-ms-emergency:
  ...

coverage-ms-emergency:
  stage: coverage
  script:
    - cd emergency
    # Le plugin JaCoCo (Java Code Coverage) génère un rapport de couverture du code source p
    - ./mvnw jacoco:report
  artifacts:
```

```
    when: always
    # paths permet de sauvegarder les artefacts générés pendant l'execution du script sur le
    # et de les retrouver dans l'onglet browse du job ou download du pipeline
    paths:
      - emergency/target/site/jacoco/
```

image

*Figure 12: Rapport html généré par JaCoCo*

### .gitlab-ci.yml, Code Quality

Dans ce paragraphe, je vais décrire le stage *quality* de notre pipeline d'intégration continue. Pour l'exécution du job *code_quality_job* nous allons utiliser une *image* docker, différente de l'image maven utilisée jusqu'à maintenant. Une chose intéressante à remarquer est le mot clé *services* qui spécifie *docker:stable-dind*. Le *dind* signifie Docker in Docker et veut dire que le runner va utiliser Docker comme *executor*, pour exécuter les scripts, et que le script utilise lui même une image Docker de Code Climate afin de créer un rapport sur la qualité du code source. Nous ajoutons le plugin *SonarJava*, un analyseur de code qui nous permet de détecter les code smells, bugs et failles de sécurité. Pour activer ce plugin nous créons un fichier nommé *.codeclimate.yml* à la racine du projet. Ce fichier nous permet non seulement d'activer le plugin mais aussi d'exclure les répertoires *mvn*, *test* et *target* de l'analyse. Une fois terminé, comme pour les autres jobs, un artefact est créé et peut être téléchargé ou visualisé dans un navigateur (voir *Figure 13* ci-dessous).

*.gitlab-ci.yml*

```
image: maven:latest

stages:
  - build
  - unit-test
  - coverage
  - quality

build-ms-emergency:
  ...

unit-test-ms-emergency:
  ...

coverage-ms-emergency:
  ...

code_quality_job:
  stage: quality
```

```
  image: docker:stable
  services:
    - docker:stable-dind
  script:
    - mkdir codequality-results
    - docker run
      --env CODECLIMATE_CODE="$PWD"
      --volume "$PWD":/code
      --volume /var/run/docker.sock:/var/run/docker.sock
      --volume /tmp/cc:/tmp/cc
      codeclimate/codeclimate analyze -f html > ./codequality-results/index.html
  artifacts:
    paths:
      - codequality-results/
```

*.codeclimate.yml*

```
plugins:
  sonar-java:
    enabled: true
    config:
      sonar.java.source: "17"
exclude_patterns:
  - "**/.mvn/"
  - "**/target/"
  - "**/test/"
```

image

*Figure 13: Rapport html généré par Code Climate*

### .gitlab-ci.yml, Package

Cette nouvelle étape est celle dans laquelle nous allons créer des packages pour notre application et les stocker dans le *Package Registry* de GitLab. Le but de ce processus est de préparer le travail de déploiement de la phase de livraison continue en sauvegardant nos artefacts. Pour cela nous ajoutons un nouveau stage, *package* et un job *package-ms-emergency*. Dans la partie script du job, j'utilise le plugin deploy de Maven pour ajouter les différents artefacts au package registry. Deploy représente la dernière phase du cycle de vie par defaut de la création d'une application avec Maven. Une fois le job achevé nous retrouvons les Fat JAR de nos applications contenant toutes nos classes, ressources et dépendances dans le menu Package Registry (voir *Figure 14* ci-dessous). Nous pouvons aussi accéder à notre application via l'onglet dowload ou browse du job, comme pour les rapports.

```
image: maven:latest
```

```yaml
stages:
  - build
  - unit-test
  - coverage
  - quality
  - package

build-ms-emergency:
  ...

unit-test-ms-emergency:
  ...

coverage-ms-emergency:
  ...

code_quality_job:
  ...

package-ms-emergency:
  stage: package
  script:
    - cd emergency
    - 'mvn deploy -s ci_settings.xml'
  artifacts:
    when: always
    paths:
      - emergency/target/*.jar
```

*Figure 14: Visualisation du Package Registry de GitLab*

**.gitlab-ci.yml, Docker**

L'ultime étape de notre pipeline d'intégration continue, repose sur la création d'image docker que nous stockons dans le *Container Registry* de GitLab. L'objectif de ce processus est de faciliter le travail de déploiement de la phase de livraison continue. Pour cela nous ajoutons un nouveau stage, *docker* et un job *dockerize-ms-emergency*. J'utilise *Kaniko*, un outil de création d'image docker qui n'a pas besoin d'accès privilégiés sur la machine hôte et nous évite des problèmes de sécurité. Une fois le job achevé nous retrouvons les images de nos applications dans le menu Container Registry (voir *Figure 15* ci-dessous). Désormais, nous pouvons utiliser ces images pour la création de nos conteneurs.

```yaml
image: maven:latest
```

```yaml
stages:
  - build
  - unit-test
  - coverage
  - quality
  - package
  - docker

build-ms-emergency:
  ...

unit-test-ms-emergency:
  ...

coverage-ms-emergency:
  ...

code_quality_job:
  ...

package-ms-emergency:
  ...

dockerize-ms-emergency:
  stage: docker
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
  script:
    - mkdir -p /kaniko/.docker
    - echo "{\"auths\":{\"${CI_REGISTRY}\":{\"auth\":\"$(printf "%s:%s" "${CI_REGISTRY_USER}
    - >-
      /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/emergency/Dockerfile"
      --destination "${CI_REGISTRY_IMAGE}/emergency:${CI_COMMIT_TAG}"
```

image

*Figure 15: Visualisation du Container Registry de GitLab*

## Conclusion

Dans cette présentation, nous avons vu comment construire un pipeline d'intégration continue avec GitLab. Les différentes étapes du pipeline nous permettent dorénavant de récupérer des rapports sur l'exécution des tests unitaires, sur la couverture du code par les tests, sur la qualité du code source

de notre projet et finalement, non seulement créer, mais aussi conteneuriser nos applications. Cette configuration peut évidemment être améliorée. Elle constitue une base de travail à laquelle nous pouvons par exemple ajouter des tests d'intégration mais aussi une étape de vérification des vulnérabilités de nos conteneurs pour ensuite compléter d'autres aspects DevOps, comme la mise en production automatisée de nos applications.

## Références

### Liens présents dans la documentation

About GitLab How to prevent crypto mining abuse on GitLab.com SaaS Install GitLab Install GitLab Runner Registering runners Configuring GitLab Runner GitLab Runner commands Maven MedHead OpenClassrooms Test coverage visualization

### Liens attachés à un paragraphe du document

Introduction | GitLab Job Artifacts | GitLab Création du pipeline | GitLab Build et tests unitaires | Maven Code Coverage | Jacoco | Code Quality | Code Climate Code Quality | Sonar Java Code Quality | Docker in Docker Package | Maven Gitlab Package | Maven Deploy Docker | Kaniko

### Vidéos

Gitlab CI pipeline tutorial for beginners 1. Switzerland GitLab meetup: First time GitLab & CI/CD workshop with Michael Friedrich GitLab Virtual Meetup - Intro to GitLab CI featuring Michael Friedrich Getting started with GitLab CI/CD GitLab Code Quality: Speed Run Philippe Charrière - Se former “en douceur” à GitLab, GitLab CI & CD avec OpenFaas