

Contents

Editorial	2
Intégration continue avec GitLab	3
Introduction	3
Concepts clés	3
Installation	4
Création d'un nouveau projet	6
Création du pipeline	7
Conclusion	13
Références	14
GitLab Action with MySQL on a Java Spring Application	15
Introduction	15
Création du pipeline CI	15
Conclusion	18
References	19
ColorPicker: Une librairie pour choisir une couleur sous Android	20
Introduction	20
Procédure Installation	20
Procédure D'utilisation	20
References	21

Editorial

Le-Point-Technique est un journal annuel regroupant les travaux étudiants, généralement sous la forme de tutoriel ou de notice explicative.

Ce premier numéro contient trois travaux. Un article sur une librairie développée par un étudiant afin de sélectionner une couleur sur Android. Et deux travaux centres DevOps avec la mise en place d'une pipeline Ci/Cd sous GitLab.

Bonne lecture !

Intégration continue avec GitLab

Nicolas, BERTRAND

Le-Point-Technique, January/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/builds"]
    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.

```
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 pourrions 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 GitLab 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

- ./mvnw jacoco:report

```
  artifacts:
```

```

when: always
# paths permet de sauvegarder les artefacts générés pendant l'exécution du script sur l'agent
# 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 défaut 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 download ou browse du job, comme pour les rapports.

```
image: maven:latest
```

```

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
  image

```

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.

```
image: maven:latest
```

```

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\" \"${CI_REGISTRY_USER}\" \"${CI_REGISTRY_PASSWORD}\")\"}}}" > /kaniko/.docker/config.json
    - >-
      /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

GitLab Action with MySQL on a Java Spring Application

Cansell, Maxime Le-Point-Technique, March/2022

abstract: Ce document présente la configuration d'une action GitLab avec une base de données MySQL dans un environnement Java Spring.

keywords: GitLab, DevOps, MySQL, Java, Spring

Introduction

Ce document décrit les étapes de configuration d'une action GitLab avec une base de données MySQL dans une application Java utilisant le framework Spring. Il s'inspire en partie du tutoriel DevOps with GitLab. Ce document comprend deux sections: création du pipeline CI et conclusion. La création du pipeline CI contient cinq sous-sections : configuration du pipeline, Instanciation du service MySQL, enregistrement des variables d'environnement avec GitLab, exécution de la pipeline et présentation des artifacts de builds.

Création du pipeline CI

Dans cette section, nous décrivons les différentes étapes de la configuration du fichier `.gitlab-ci.yml`, configuration centrale de notre pipeline. Nous verrons ensuite comment se déroulent les tests, les rapports associés et comment déboguer notre pipeline jusqu'à son fonctionnement.

Configuration

Afin de paramétrer notre pipeline, nous devons commencer par créer un fichier, nommé `.gitlab-ci.yml`, à la racine de notre repository. Ce fichier est composé de l'extrait de code visible *Figure 1*. Nous allons décrire sa composition pour chaque élément de code dans les sous-sections suivantes. Plus tard nous rajouterons le code qui concerne la configuration de l'image de la base de données MySQL.

A chaque push du code source des applications ou du fichier `.gitlab-ci.yml` sur le repository, le code de ce fichier `.gitlab-ci.yml` sera exécuté automatiquement.

image 1

Figure 1: Exemple d'action GitLab.

Prenons le temps d'analyser cette exemple ligne par ligne:

- **image** (ligne 1) : Nous installons l'image docker Alpine qui est une distribution Linux ultra-légère que notre pipeline va utiliser pour exécuter nos scripts. Nous y intégrons le JDK JAVA 11.

- **stages** (ligne 5) : Nous décrivons ici de quoi sera composée notre pipeline ; trois stage, un nommé build qui testera que le build de l'application fonctionne normalement et un nommé tests qui réalisera les tests unitaires et d'intégration. Le troisième stage sera chargé de vérifier si les tests ont réussis ou échoués. Ces trois étapes sont totalement indépendantes. En effet, chaque stage se déroule un après l'autre et crée son conteneur docker. La base de données MySQL sera installée dans le conteneur du stage "tests" et donc en ligne de commande dans son script (voir point MySQL).
- **build-ms-ERS-Emergency-Responder** (ligne 10) : Nous nommons ici notre premier job et l'attribuons au stage build (ligne 11). Les lignes de commandes du script (ligne 13) sont ensuite exécutées : parcourir vers le dossier de l'application, build l'application avec Maven et enfin afficher dans nos logs que cela s'exécute.
- **tests-ms-Emertency-Responder** (ligne 17) : Deuxième job, même principe que le job précédent. C'est ici les tests unitaires et d'intégrations que contient notre application qui sont lancés lors du script. La clé nommée artifacts décrit les méthodes de sauvegarde des rapports (se référer aux commentaires de l'extrait de code). Le pipeline échoue uniquement si une des commandes échoue (Elles renvoient alors le code 1 ou 2 qui font fail le pipeline et le job). La réussite d'une commande renvoi 0 et le job passe à la commande suivante. Ici le job n'analyse pas la réussite ou l'échec des tests mais seulement le bon déroulement des commandes. Le job réussi donc même si des tests unitaires ou d'intégration échouent. Le job suivant se chargera de vérifier et de faire fail le pipeline si un des tests ne passe pas.
- **check-tests-failure** (ligne 38) : Ce troisième job nous permet donc d'analyser le rapport Junit pour vérifier si les tests unitaires ou d'intégration ont échoué.
- **dependencies** (ligne 40) : Le mot clef dependencies permet de transmettre les rapports générés dans le job précédent et de les rendre accessibles dans ce stage.
- **script** (ligne 42) : Le script utilise la commande grep. Grep est un acronyme qui signifie Global Regular Expression Print. C'est un outil en ligne de commande Linux / Unix utilisé pour rechercher une chaîne de caractères dans un fichier spécifié. Le modèle de recherche de texte est appelé une expression régulière (regex). Nous cherchons donc à savoir si une balise "<failure[...]" ou "<error[...]" existe dans le rapport Junit. Cette commande renvoie 0 si elle trouve l'expression et 1 si elle ne la trouve pas. Nous inversons la condition avec le point d'exclamation : s' il la trouve il renverra 1 qui fera échouer le job et donc notre pipeline.

Instanciation du service MySQL

Nous devons tout d’abord ajouter (ligne 9) nos variables MySQL pour permettre la connexion à la base de données. Attention, il conviendra de mettre en place External Secret GitLab pour sécuriser les variables de connexion qui sont visibles (*Figure 2*).

image 2

Figure 2: Configuration des variables MySQL

Nous modifions ensuite le stage tests (*Figure 3*).

image 3

Figure 3: Screenshot des modifications sur le stage tests

Nous allons utiliser un service nommé mariadb pour y créer notre base de données MySQL (ligne 31). Un service est lui même créé dans un conteneur docker à l’intérieur du conteneur du stage test. Pour accéder à celui-ci il faut donc s’adresser au conteneur docker portant son nom “mariadb”. L’URL d’accès à la base de données ne sera donc pas localhost mais mariadb (exemple mariadb://3306/hospitals). Nous y reviendrons au point suivant. La commande apk (ligne 34) installe le client mysql. Comme nous le constatons sur ce lien il convient d’attendre 20 secondes (ligne 36) pour s’assurer de l’installation correcte. (Nous pourrions optimiser cela dans les développements futurs en testant si ce temps pourrait être réduit). Nous nous connectons ensuite et créons notre base de données grâce au script data.sql contenu à la racine de notre application (ligne 37). Nous affichons les tables créées de la base de données dans nos logs (ligne 38) puis exécutons les tests grâce aux commandes déjà vues précédemment (lignes 39/40).

Enregistrement des variables GitLab

Gitlab permet d’enregistrer des variables d’environnement qui seront réutilisées dans notre pipeline et donc par notre application en fonctionnement dans nos stages (voir *Figure 4*). Suivre : `seetings > Ci/CD > Variables` et `Expand`.

image 4

Figure 4: Capture d’écran des variables d’environnement sous Gitlab

Ces variables peuvent écraser et remplacer les variables de notre application comme celles contenues dans le fichier de configuration Spring de notre application : `application.properties` mais aussi celles du fichier de configuration du pipeline : `.gitlab-ci.yml`.

Il convient de respecter la case majuscule underscore pour écraser les variables du fichier yml et minuscule underscore pour celles du fichier `application.properties`. Pour que notre conteneur où s’exécute l’application et ses tests puissent communiquer avec le conteneur du service mariadb et sa base de donnée MySQL

nous devons écraser la valeur de la propriété “spring.datasource.url” du fichier application.properties de notre application Spring :

image 5

Figure 5: Mise à jour des propriétés spring boot.

La variable se nommera donc “spring_datasource_url” et aura pour valeur, comme nous l’avons vu précédemment : “jdbc:mysql://mariadb:3306/hospitals”
Note concernant la conception de la base de donnée : il convient de respecter la case minuscule underscore dans le nom des tables et des de champs de la base de donnée pour respecter les paramètres par défauts de JPA hibernate. Attention, une variable gitlab même indiquée comme masquée n’est pas entièrement sécurisée. Pour sécuriser correctement les variables comme par exemple des clefs API ou des clefs de connexion à des bases de données il conviendra d’utiliser des external secret gitlab

Exécution de la pipeline

A chaque commit le pipeline est exécuté. Son exécution peut passer par l’état “running” quand il est en cours d’exécution, “passed” quand l’exécution s’est déroulée correctement ou “failed” quand elle à échoué (indépendant de la réussite des tests ou de leurs échecs) - *Figure 6*.

image 6

Figure 6: Statut de la pipeline.

La capture d’écran suivante (*Figure 7*) montre les logs de l’exécution du job stage tests qui s’est correctement déroulée.

image 7

Figure 7: Exemple de logs d’exécution.

Artefacts créés lors du pipeline

Nous disposons d’un rapport sauvegardé au format html disponible en suivant le chemin suivant : <<appli>>/target/site/surefire-report.html Nous pouvons également voir le statut de réussite ou d’échec des tests et leurs origines sur la **Figure 8** (cliquer sur le stage puis sur l’onglet test) :

image 8

Figure 8: Screenshot de l’onglet Pipelines.

Conclusion

Ce document nous a permis de mettre en place notre pipeline d’intégration continue pour une application Spring avec sa base de données de test MySQL.

Ce document pourra être complété avec la mise en place d'outils de sécurité, notamment des tests de fuzzing.

References

DevOps with GitLab

GitLab - API Fuzzing

GitLab - Application Security

External Secret Gitlab

ColorPicker: Une librairie pour choisir une couleur sous Android

Amelin, Lucas; Abdelouahad, Mustapha

Le-Point-Technique, March/2022

abstract: Présentation d'une librairie pour choisir une couleur sous Android. La librairie s'installe facilement avec Jetpack, et permet d'intégrer un bouton dit **flottant** au layout de l'application. Un clic sur ce bouton permet d'ouvrir une fenêtre de sélection sous la forme d'un disque colore. L'utilisateur peut alors sélectionner la couleur de son choix en cliquant sur un endroit du disque.

keywords: Android, Jetpack, library, color picker

Introduction

ColorPicker est une librairie Android qui permet a un utilisateur de choisir une couleur sur un disque coloré. La librairie s'intègre sous la forme d'un bouton dit **flottant** au layout de l'application (*Figure 1*).

image 1

Figure 1: La librairie se présente sous la forme d'un bouton a intégrer a l'application

Un clic sur ce bouton permet d'ouvrir une interface de sélection sous la forme d'un disque colore (*Figure 2*).

image 2

Figure 2: Visualisation de l'interface permettant de sélectionner une couleur

L'utilisateur clic sur un endroit du disque pour choisir la couleur. La couleur du bouton **valider** en bas de l'écran est alors mise à jour en fonction de la couleur sélectionnée. Le clic sur le bouton **valider** permet de sélectionner la couleur.

Procédure Installation

1. Dans `app/build.gradle` ajouter la dépendance suivante : *implementation 'com.github.LucasGitHubz:ColorPicker:Release Number'* Où **Release Number** correspond à une des versions sur cette page.
2. A la racine du projet dans le fichier `settings.gradle` ajouter dans `repositories { ... }` : `maven { url 'https://jitpack.io' }`

Procédure D'utilisation

Dans votre layout ajouter le bouton :

```
<com.lucasgithubz.colorpicker.FloatingButton
    android:id="@+id/colorPickerButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Du côté code, utiliser ensuite la fonctionnalité de *view binding* ou butterknife pour récupérer le bouton `flottant` grâce à son id. Les méthodes `getColor` et `setColor(int)` permettront respectivement de récupérer et programmer la couleur du bouton.

References

ColorPicker