



DESCRIPTION DU PIPELINE

**Projet : PoC pour le sous-système
d'intervention d'urgence en temps réel**

Client : MedHead

Pipeline : OC-P11-MEDHEAD-POC

HISTORIQUE DES RÉVISIONS

Date	Version	Commentaires
12 janvier 2022	0.01	Description du pipeline préliminaire.

TABLE DES MATIÈRES

Historique des révisions	2
Table des matières	3
Introduction	4
Concepts clés de GitLab	4
Intégration continue	4
Job	4
Job Artifacts	4
Pipeline	4
Gitlab Runners	5
Gitlab Server	5
Installation	5
Specific Runners	5
Création du pipeline	8
Application	8
Configuration	10
Builds	11
Tests unitaires	12
Code Coverage	14
Code Quality	15
Package	17
Docker	18
Conclusion	19
Table des illustrations	20
Figure	20
Images	20
Références	21
Liens	21
Vidéos	21

INTRODUCTION

Dans ce document, nous décrivons le pipeline d'intégration continue mis en place pour réaliser la preuve de concept du projet de création d'un sous-système d'intervention d'urgence en temps réel souhaité par MedHead. Nous commençons par présenter l'outil que nous avons retenu, GitLab. Puis, nous aborderons brièvement l'installation de cette plateforme avant de poursuivre avec une présentation de l'application mise en place pour cette POC. Enfin, nous terminerons avec la création du pipeline et l'automatisation de son exécution.

CONCEPTS CLÉS DE GITLAB

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 cette partie, je vais définir le vocabulaire employé dans ce document 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'objectif est d'exécuter des scripts automatiques qui permettent 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 notre 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 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 via la plateforme et les *specific runners* qui sont spécifiques à un projet et peuvent être installés localement 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

Nous n'allons pas expliquer comment installer GitLab dans ce document car nous avons choisi de profiter de la possibilité de s'inscrire à l'offre gratuite afin 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 pour 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. Un quota, exprimé en temps d'utilisation, limite lui aussi l'utilisation des shared runners. Lors de l'implémentation du pipeline, nous avons choisi d'installer un runner sur notre machine de façon à lever les différentes limitations de l'offre gratuite. Dans le paragraphe suivant nous décrivons l'installation d'un runner spécifique, réalisée à l'aide de la documentation officielle ([voir Install GitLab Runner](#)).

SPECIFIC RUNNERS

Par défaut, le pipeline de GitLab utilise les shared runners pour exécuter les jobs. Vous trouvez cette information en naviguant dans le menu *Settings* du projet, puis *CI/CD*, et enfin *Runners* ([voir Image 1 ci-dessous](#)).

The screenshot shows the 'Runners' page in the GitLab interface. On the left, there's a sidebar with various project settings like 'Project information', 'Repository', 'Issues', 'Merge requests', 'CI/CD', etc. The 'CI/CD' tab is selected. The main content area is titled 'Runners' and contains two sections: 'Specific runners' and 'Shared runners'. The 'Shared runners' section is highlighted with a red circle and a red arrow pointing to it from the top right. It says: 'These runners are shared across this GitLab instance. They're available on GitLab.com run in autoscale mode and are powered by Google Cloud Platform. Autoscaling means reduced wait times to spin up builds, and isolated VMs for each project, thus maximizing security.' Below this, there's a list of shared runners with their status (green or red), ID, name, and environment tags (e.g., docker, east-c, go, git-annex, linux, mongo, mysql, postgres, ruby, shared). There are also buttons for 'Reset registration token' and 'Show Runner installation instructions'.

Image 1 - Les runners du projet

Dans la colonne de gauche, *Specific runners*, nous trouvons des liens vers la procédure d'installation suivant différents environnements (voir *Image 2* ci-dessous).

Environment

[Linux](#) [macOS](#) [Windows](#) [Docker](#) [Kubernetes](#)

Architecture

amd64 ▾

Download and install binary

[Download latest binary](#)

```
# Download the binary for your system
sudo curl -L --output /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64

# Give it permissions to execute
sudo chmod +x /usr/local/bin/gitlab-runner

# Create a GitLab CI user
sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash

# Install and run as service
sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
sudo gitlab-runner start
```



Command to register runner

```
sudo gitlab-runner register --url https://gitlab.com/ --registration-token $REGISTRATION_TOKEN
```



Une fois installé en local, nous devons enregistrer le runner pour notre projet. Nous pouvons réaliser cette tâche en mode interactif ou one-line. Nous avons choisi le mode interactif, dont voici les étapes pour un environnement linux :

1. Exécutez la commande : `sudo gitlab-runner register`
2. Entrez l'URL de l'instance GitLab : <https://gitlab.com/>
3. Entrez le jeton fourni pour enregistrer le runner : `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, nous devrions voir le runner spécifique de notre machine disponible et actif pour exécuter les jobs de notre pipeline (voir *Image 3* ci-dessous).

The screenshot shows the GitLab interface for managing runners. On the left, there's a sidebar with project navigation and settings. The main area is titled 'Runners' and contains sections for 'Runners' and 'Shared runners'. Under 'Runners', it says 'Runners are processes that pick up and execute CI/CD jobs for GitLab. How do I configure runners?'. It explains that runners can be registered as separate users or on local machines. There are two status categories: 'active' (green) and 'paused' (red). The 'Specific runners' section is titled 'These runners are specific to this project.' It includes instructions to set up a runner for the project, which involves installing GitLab Runner and registering it with the URL <https://gitlab.com/>. It also asks for a registration token, which is shown as a redacted string. Below this are buttons for 'Reset registration token' and 'Show Runner installation instructions'. To the right, the 'Shared runners' section is titled 'These runners are shared across this GitLab instance.' It lists several shared runners, each with a green circular icon and a unique ID. One specific runner, '#13145631 (TBczVoWb)', is highlighted with a red circle and a red arrow pointing from the left side of the screen towards it. This runner is associated with the project and has the tag 'myLocalRunner'. The 'Shared runners' section also includes a button to enable shared runners for the project.

Image 3 - Notre runner spécifique est disponible

L'action d'enregistrer un runner spécifique pour notre projet crée un fichier de configuration appelé `config.toml` (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.

CRÉATION DU PIPELINE

Dans ce chapitre, nous présentons l'application associée au pipeline. Puis, nous décrivons les différents stages et jobs contenus dans le fichier `.gitlab-ci.yml`, élément central de la plateforme DevOps, en fournissant un extrait du code présent dans le fichier et en décrivant le résultat attendu.

APPLICATION

La création du pipeline est liée au projet accessible à cette adresse [MedHead](#) (voir *Image 4* ci-dessous). Il s'agit de plusieurs applications Spring Boot, hébergées dans un repository GitLab, qui utilisent l'outil Maven et le langage Java. La *Figure 1* ci-dessous représente les relations entre les différentes applications.

Name	Last commit	Last update
emergency	Test code clean	2 days ago
gateway	Adding possibility to create an emergency ...	1 week ago
hospital	Adding Frontend & Cypress tests executio...	1 week ago
patient	Adding possibility to create an emergency ...	1 week ago
registry	Code refactor integrating SonarLint guida...	3 weeks ago
.codeclimate.yml	Exclude test directories from codequality	1 week ago
.gitignore	Excluding test directories from codequality	1 week ago
.gitlab-ci.yml	Removing Integration tests from pipeline	1 week ago

Image 4 - Repository du projet MedHead

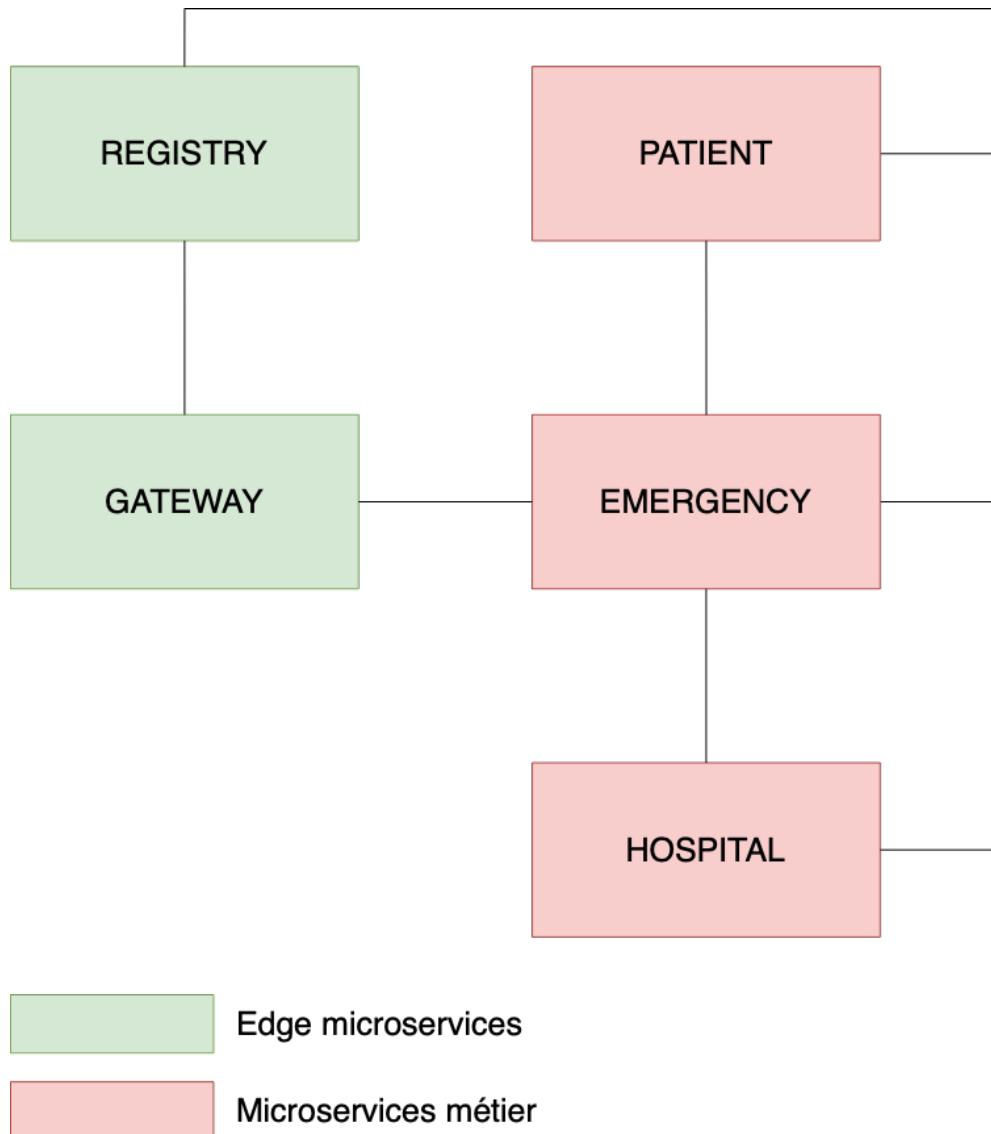


Figure 1 - Architecture de l'application MedHead

CONFIGURATION

Afin de paramétriser notre pipeline avec la plateforme GitLab, nous devons commencer par créer un fichier, nommé `.gitlab-ci.yml` (voir *Image 5* ci-dessous), 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. Dès lors que le fichier est présent, lorsque un développeur envoie les modifications apportées au code source sur le dépôt distant, alors l'exécution du pipeline démarre automatiquement. Les différents états associés à ce traitement sont *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 (voir *Image 6* ci-dessous).

```
stages:
  - build
  - test

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

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

Image 5 - `.gitlab-ci.yml`, exemple simple

Status	Pipeline ID	Triggerer	Commit	Stages	Duration	Actions
running	#446654010 latest	main -> beed5a8	Update .gitlab-ci.yml	Passed	00:01:58 1 day ago	
failed	#446529423 latest	cocowatersw... -> 1c388870	Update .gitlab-ci.y...	Failed	1 day ago	
passed	#415601713	main -> 87a23566	Update .gitlab-ci.y...	Passed	00:08:17 1 month ago	
canceled	#41596497	main -> 1c106922	Update .gitlab-ci.y...	Passed	00:02:02 1 month ago	
canceled	#415581810	main -> f3d3087c	Update .gitlab-ci.y...	Passed	00:01:54 1 month ago	
canceled	#415577752	main -> c12139c8	Update .gitlab-ci.y...	Passed	00:04:33 1 month ago	
passed	#415569366	main -> f2b69db9	...	Passed	00:05:04	

Image 6 - Exécution du pipeline

BUILDS

C'est le premier stage de notre pipeline, nous l'appelons *build*. Nous présentons dans ce document des extraits de code et des rapports d'exécution des jobs liés à l'application *emergency* uniquement mais le principe est identique pour les applications *patient* et *hospital*. Le job associé au stage *build* pour *emergency* est appelé *build-ms-emergency* (voir *Image 7* ci-dessous). Ce script vérifie que la compilation du projet se déroule sans erreur. Nous devons obtenir le message 'Build success' à l'issue de cette étape (voir *Image 8* ci-dessous).

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

stages:
- build

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

Image 7 - .gitlab-ci.yml, build de ms-emergency

The screenshot shows the GitLab CI interface for the project 'ocr-p11-medhead-poc'. On the left, the sidebar navigation includes 'Project information', 'Repository', 'Issues' (10), 'Merge requests' (0), 'CI/CD', 'Pipelines', 'Editor', 'Jobs' (selected), 'Schedules', 'Test Cases', 'Security & Compliance', 'Deployments', 'Monitor', 'Infrastructure', 'Packages & Registries', 'Analytics', 'Wiki', 'Snippets', and 'Settings'. The main area displays the 'build-ms-emergency' job details. The job log shows the following steps:

```
15 Fetching changes with git depth set to 50...
16 Initialized empty Git repository in /builds/cocowaterswing/ocr-p11-medhead-poc/.git/
17 Created fresh repository.
18 Checking out 08109014 as main...
19 Skipping Git submodules setup
20 Restoring cache
21 Checking cache for main...
22 Downloading cache.zip from https://storage.googleapis.com/gitlab-com-runners-cache/project/32054401/main
23 Successfully extracted cache
24 Executing "step_script" stage of the job script
25 Using docker image sha256:59fcf607b5128e18da32a7c5c1cacd53a36e08b66ab49ccb4a50509bd/b91 for maven:latest with digest maven@sha256:208ceab2ef5edc2845e18942fd63a153842ed7aa9868731a8012c53c0b9b3ef ...
26 $ cd emergency
27 $ ./mvnw compile
28 [INFO] Scanning for projects...
29 [INFO] 2340 [INFO]
30 [INFO] 2350 [INFO] ocr.medhead:emergency >
31 [INFO] 2350 [INFO] Building emergency 0.8.1-SNAPSHOT
32 [INFO] 2351 [INFO] [jar]
33 [INFO] 3773 [INFO]
34 [INFO] 3773 [INFO] --- maven-resources-plugin:3.2.0:resources (default-resources) @ emergency ---
35 [INFO] 4143 [INFO] Using 'UTF-8' encoding to copy filtered resources.
36 [INFO] 4146 [INFO] Using 'UTF-8' encoding to copy filtered properties files.
37 [INFO] 4151 [INFO] Using 1 resource
38 [INFO] 4247 [INFO] Copying 0 resource
39 [INFO] 4249 [INFO]
40 [INFO] 4250 [INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ emergency ---
41 [INFO] 4604 [INFO] Changes detected - recompiling the module!
42 [INFO] 4608 [INFO] Compiling 17 source files to /builds/cocowaterswing/ocr-p11-medhead-poc/emergency/target/classes
43 [INFO] 8558 [INFO]
44 [INFO] 8559 [INFO] BUILD SUCCESS
45 [INFO] 8560 [INFO]
46 [INFO] 8563 [INFO] Total time: 6.6828 s
47 [INFO] 8567 [INFO] Finished at: 2022-01-06T16:06:18Z
48 [INFO] 8567 [INFO]
49 [INFO] 52 Saving cache for successful job
50 Creating cache main...
51 .m2/repository: Found 3856 matching files and directories
52 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/32054401/main
53 Created cache
54 Cleaning up project directory and file based variables
55 Job succeeded
```

The right panel shows the pipeline summary: Duration: 56 seconds, Finished: 2 days ago, Timeout: 1h (from project), Runner: #1270852 (Jhc_jxvh) 3-green.shared.runners-manager.gitlab.com/default. It also lists the commit 08109014, Pipeline #442816345 for main, and branches build-ms-emergency, build-ms-hospital, and build-ms-patient.

Image 8 - Visualisation du résultat du build pour ms-emergency

TESTS UNITAIRES

Pour cette étape, nous ajoutons un stage nommé *unit-test*. Si le build s'est terminé sans erreur, le pipeline poursuit son exécution avec le job nommé *unit-test-ms-emergency* (voir *Image 9* ci-dessous). Ce job nous permet 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 *Image 10* ci-dessous), ainsi que dans un format *xml* avec *reports:junit* pour qu'il soit intégré dans l'interface utilisateur de Gitlab (voir *Image 11* ci-dessous).

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

stages:
- build
- unit-test

build-ms-emergency:
...

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 le GitLab Server
    # 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
```

Image 9 - .gitlab-ci.yml, tests unitaires de ms-emergency

```

18:20:34.622 [parallel=1] DEBUG org.springframework.web.reactive.server.HttpHandlerConnector - Creating client response for POST "/emergencies"
843 18:20:34.624 [parallel=1] DEBUG org.springframework.web.server.adapter.HttpWebHandlerAdapter - [73ec597d] Completed 201 CREATED
844 18:20:34.628 [parallel=1] DEBUG org.springframework.web.reactive.function.client.ExchangeFunctions - [d325518] [4b43cb9] Response 201 CREATED
845 18:20:34.634 [main] DEBUG org.springframework.core.codec.ResourceDecoder - [d325518] [4b43cb9] Read 159 bytes
846 18:20:34.708 [main] DEBUG com.jayway.jsonpath.internal.path.CompiledPath - Evaluating path: ${'id'}
847 18:20:34.709 [main] DEBUG com.jayway.jsonpath.internal.path.CompiledPath - Evaluating path: ${'patientSurname'}
848 18:20:34.709 [main] DEBUG com.jayway.jsonpath.internal.path.CompiledPath - Evaluating path: ${'hospitalId'}
849 18:20:34.710 [main] DEBUG com.jayway.jsonpath.internal.path.CompiledPath - Evaluating path: ${'hospitalId'}
850 18:20:34.710 [main] DEBUG com.jayway.jsonpath.internal.path.CompiledPath - Evaluating path: ${'hospitalName'}
851 16832 [INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.882 s - in ocr.medhead.emergency.controller.EmergencyControllerTests
852 17164 [INFO] Results:
853 17165 [INFO]
854 17165 [INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
855 17166 [INFO]
856 17171 [INFO] <<< maven-surefire-report-plugin:3.0.0-M5:report (default-cli) < [surefire]test @ emergency <<<
857 17172 [INFO] ----
858 17171 [INFO] --- maven-surefire-report-plugin:3.0.0-M5:report (default-cli) @ emergency ----
859 17172 [INFO] -----
860 17175 [WARNING] Unable to locate Test Source XRef to link to - DISABLED
861 18124 [INFO] -----
862 18124 [INFO] BUILD SUCCESS
863 18125 [INFO] -----
864 18124 [INFO] Total time: 16.489 s
865 18127 [INFO] Finished at: 2022-01-12T18:20:36Z
866 18129 [INFO] -----
867 18130 [INFO] -----
868 18130 [INFO] Saving cache for successful job
869 18130 [INFO] Creating cache main...
870 18130 [INFO] .m2/repository: found 4410 matching files and directories
871 18130 [INFO] Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/32054401/main
872 18130 [INFO] Created cache
873 18130 [INFO] -----
874 18130 [INFO] Uploading artifacts for successful job
875 18130 [INFO] Uploading artifacts...
876 18130 [INFO] emergency/target/site/surefire-report.html: found 1 matching files and directories
877 18130 [INFO] Uploading artifacts as "archive" to coordinator... ok id=1968441085 responseStatus=201 Created token=NdhG31
878 18130 [INFO] Uploading artifacts...
879 18130 [INFO] emergency/target/surefire-reports/TEST-*.xml: found 1 matching files and directories
880 18130 [INFO] Uploading artifacts as "junit" to coordinator... ok id=1968441085 responseStatus=201 Created token=NdhG31
881 18130 [INFO] Cleaning up project directory and file based variables
882 18130 [INFO] Job succeeded
883 18130 [INFO] 00:01

```

Commit: 9
Config to create emergency and hospital packages

Pipeline #446736360 for main
unit-test

→ unit-test-ms-emergency
unit-test-ms-hospital
unit-test-ms-patient

Image 10 - Télécharger ou visualiser un rapport d'exécution des tests unitaires

Suite	Name	Filename	Status	Duration	Details
ocr.medhead.emergency.controller. EmergencyControllerTests	canRecordAnEmergencyByHospitalNameAndSpecialtyAndNoBedAvailable		✓	3.27s	<button>View details</button>
ocr.medhead.emergency.controller. EmergencyControllerTests	canRetrieveAllEmergencies		✓	355.00ms	<button>View details</button>
ocr.medhead.emergency.controller. EmergencyControllerTests	canRecordAnEmergencyByHospitalNameAndAvailableBed		✓	217.00ms	<button>View details</button>
ocr.medhead.emergency.controller. EmergencyControllerTests	canRecordAnEmergencyByHospitalNameAndSpecialtyAndAvailableBed		✓	213.00ms	<button>View details</button>
ocr.medhead.emergency.controller. EmergencyControllerTests	cannotCreateEmergencyWithPatientIdOnly		✓	186.00ms	<button>View details</button>
ocr.medhead.emergency.controller. EmergencyControllerTests	canRecordAnEmergencyBySpecialtyAndAvailableBed		✓	179.00ms	<button>View details</button>
ocr.medhead.emergency.controller. EmergencyControllerTests	canRecordAnEmergencyByHospitalNameAndNoBedAvailable		✓	169.00ms	<button>View details</button>

Image 11 - Intégration du rapport d'exécution des tests unitaire dans GitLab

CODE COVERAGE

Dans cette troisième étape, nous ajoutons le stage `coverage` et le job `coverage-ms-emergency` au fichier `.gitlab-ci.yml` (voir *Image 12* ci-dessous). 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 *Image 13* ci-dessous).

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

Image 12 - `.gitlab-ci.yml`, code coverage de ms-emergency

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
ocr.medhead.emergency.domain		33%		16%	47 67	3 21	13 31	0 2
ocr.medhead.emergency.repository.hospital		0%		0%	14 14	19 19	4 4	1 1
ocr.medhead.emergency.service		0%	n/a	5 5	17 17	5 5	1 1	
ocr.medhead.emergency.repository.emergency		0%	n/a	4 4	12 12	4 4	1 1	
ocr.medhead.emergency.beans		41%	n/a	13 20	10 22	13 20	0 2	
ocr.medhead.emergency.repository.patient		0%		0%	5 5	7 7	4 4	1 1
ocr.medhead.emergency.exceptions		0%	n/a	9 9	11 11	9 9	3 3	
ocr.medhead.emergency		0%	n/a	2 2	3 3	2 2	1 1	
ocr.medhead.emergency.controller		100%	n/a	0 6	0 11	0 6	0 2	
Total	740 of 968	23%	82 of 94	12%	99 132	82 123	54 85	8 14

Created with JaCoCo 0.8.7.202105040129

Image 13 - Rapport html généré par JaCoCo

CODE QUALITY

Dans ce paragraphe, je vais décrire la quatrième phase de notre pipeline d'intégration continue, le stage *quality*. Pour l'exécution du job *code_quality_job* nous allons utiliser une *image docker*, différente de l'image maven utilisée jusqu'à maintenant (voir *Image 14* ci-dessous). 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* (voir *Image 15* ci-dessous) 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 *Image 16* ci-dessous).

```
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/
```

Image 14 - Le job pour le code quality du projet

```

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

```

Image 15 - .codeclimate.yml, plugin Sonar Java

The screenshot shows a detailed code review report from Code Climate. It highlights three specific issues in the code:

- Method 'findHospital' has a Cognitive Complexity of 6 (exceeds 5 allowed). Consider refactoring.**
- Method 'findBySpecialityAndBeds' has a Cognitive Complexity of 6 (exceeds 5 allowed). Consider refactoring.**
- Similar blocks of code found in 2 locations. Consider refactoring.**

Each issue is accompanied by a snippet of Java code, a 'Details' link, and a 'structure' link. The interface includes dropdown menus for 'Category' (All Categories) and 'Engine' (All Engines).

Image 16 - Rapport html généré par Code Climate

PACKAGE

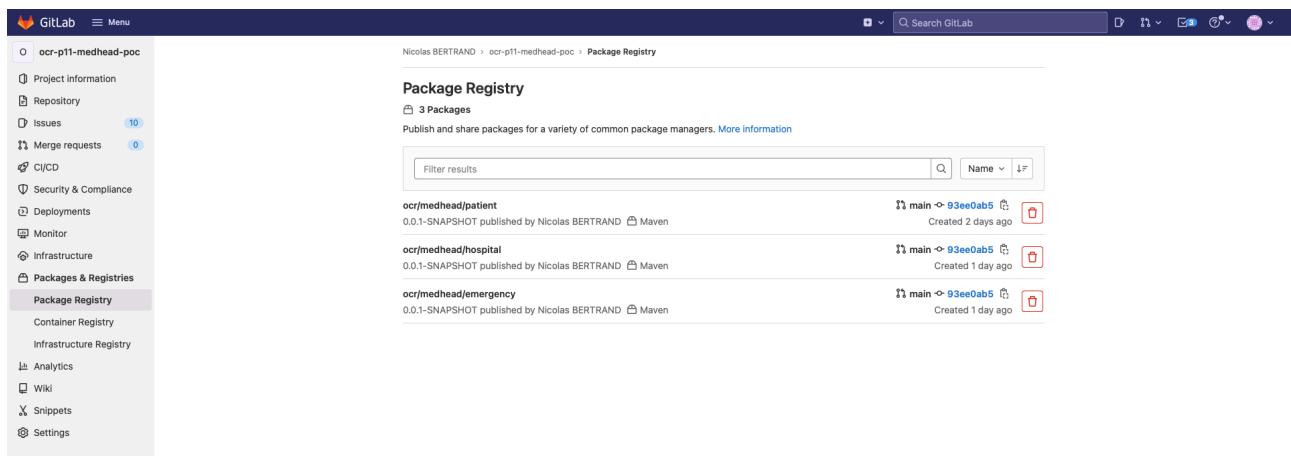
Dans cette étape, nous allons créer des packages pour notre application et les stocker dans le *Package Registry* de GitLab. Le but de ce processus est de préparer le travail de déploiement de la phase de livraison continue en sauvegardant nos artefacts. Pour cela nous ajoutons un nouveau stage, *package* et un job *package-ms-emergency* (voir *Image 17* ci-dessous). 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 *Image 18* ci-dessous).

```
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 mergency
    - 'mvn deploy -s ci_settings.xml'
  artifacts:
    when: always
    paths:
      - target/*.jar
```

Image 17 - Le job pour la création du package ms-emergency



The screenshot shows the GitLab interface with the sidebar open. Under the 'Packages & Registries' section, 'Package Registry' is selected. The main area displays the 'Package Registry' page with the heading '3 Packages'. Below this, there is a table listing three packages:

Name	Version	Published By	Created	Action
ocr/medhead/patient	0.0.1-SNAPSHOT	Nicolas BERTRAND	Maven	main 93ee0ab5 trash
ocr/medhead/hospital	0.0.1-SNAPSHOT	Nicolas BERTRAND	Maven	main 93ee0ab5 trash
ocr/medhead/emergency	0.0.1-SNAPSHOT	Nicolas BERTRAND	Maven	main 93ee0ab5 trash

Image 18 - Visualisation du Package Registry de GitLab

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* (voir *Image 19* ci-dessous). Une fois le job achevé nous retrouvons les images de nos applications, utilisables pour la création de conteneurs dans le menu *Container Registry* (voir *Image 20* ci-dessous).

```
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/executor
--context "${CI_PROJECT_DIR}"
--dockerfile "${CI_PROJECT_DIR}/emergency/Dockerfile"
--destination "${CI_REGISTRY_IMAGE}/emergency:${CI_COMMIT_TAG}"
```

Image 19 - Le job pour la création d'une image docker de ms-emergency

The screenshot shows the GitLab Container Registry interface. On the left, there is a sidebar with project navigation links like Project information, Repository, Issues (10), Merge requests (0), CI/CD, Security & Compliance, Deployments, Monitor, Infrastructure, Packages & Registries, Container Registry (selected), Infrastructure Registry, Analytics, Wiki, Snippets, and Settings. The main area is titled "Container Registry" and shows "3 Image repositories". It includes a search bar, a filter results dropdown, and a "Updated" dropdown. Three repositories are listed: "cocowaterswing/ocr-p11-medhead-poc/hospital" (1 Tag), "cocowaterswing/ocr-p11-medhead-poc/emergency" (1 Tag), and "cocowaterswing/ocr-p11-medhead-poc/patient" (1 Tag). Each repository has a delete icon next to it.

Image 20 - Visualisation du Container Registry de GitLab

CONCLUSION

Dans ce document, nous avons décrit la construction du pipeline d'intégration continue de la preuve de concept du projet de création d'un sous-système d'intervention d'urgence en temps réel, souhaité par MedHead. 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.

TABLE DES ILLUSTRATIONS

FIGURE

Figure 1 - Architecture de l'application MedHead	9
--	---

IMAGES

Image 1 - Les runners du projet	6
Image 2 - Installer un runner spécifique	6
Image 3 - Notre runner spécifique est disponible	7
Image 4 - Repository du projet MedHead	8
Image 5 - .gitlab-ci.yml, exemple simple	10
Image 6 - Exécution du pipeline	10
Image 7 - .gitlab-ci.yml, build de ms-emergency	11
Image 8 - Visualisation du résultat du build pour ms-emergency	11
Image 9 - .gitlab-ci.yml, tests unitaires de ms-emergency	12
Image 10 - Télécharger ou visualiser un rapport d'exécution des tests unitaires	13
Image 11 - Intégration du rapport d'exécution des tests unitaire dans GitLab	13
Image 12 - .gitlab-ci.yml, code coverage de ms-emergency	14
Image 13 - Rapport html généré par JaCoCo	14
Image 14 - Le job pour le code quality du projet	15
Image 15 - .codeclimate.yml, plugin Sonar Java	16
Image 16 - Rapport html généré par Code Climate	16
Image 17 - Le job pour la création du package ms-emergency	17
Image 18 - Visualisation du Package Registry de GitLab	17
Image 19 - Le job pour la création d'une image docker de ms-emergency	18
Image 20 - Visualisation du Container Registry de GitLab	18

RÉFÉRENCES

LIENS

[OpenClassrooms - Mettez en place l'intégration continue](#)

[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](#)

[Test coverage visualization](#)

[Create Maven packages with GitLab CI/CD](#)

[Use kaniko to build Docker images](#)

[Container Scanning](#)

VIDÉOS

[Gitlab CI pipeline tutorial for beginners](#)

[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](#)