

Contents

Editorial	2
Intégration continue avec GitLab	3
Introduction	3
Concepts clés	3
Installation	4
Création d'un nouveau projet	8
Création du pipeline	8
Conclusion	20
Références	21
GitLab Action with MySQL on a Java Spring Application	23
Introduction	23
Création du pipeline CI	23
Conclusion	30
References	30
ColorPicker: Une librairie pour choisir une couleur sous Android	31
Introduction	31
Procédure Installation	31
Procédure D'utilisation	31
References	34
Etude exploratoire pour la production de Médias interactifs	35
Introduction	35
Choix préférés pour les outils et technologies	35
Technologies pour la diffusion des médias vidéo sur le web	35
Technologies pour l'authentification / l'autorisation	36
Briques de solution de référence (SBB)	39
Synthèse de l'étude de la stack technologique	43
Références	44

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. Deux travaux centrés DevOps avec la mise en place d'une pipeline Ci/Cd sous GitLab. Et une étude exploratoire pour la production de médias interactifs.

Bonne lecture !

Grégoire Cattan France, 2022

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, paramétriser 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).

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).

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

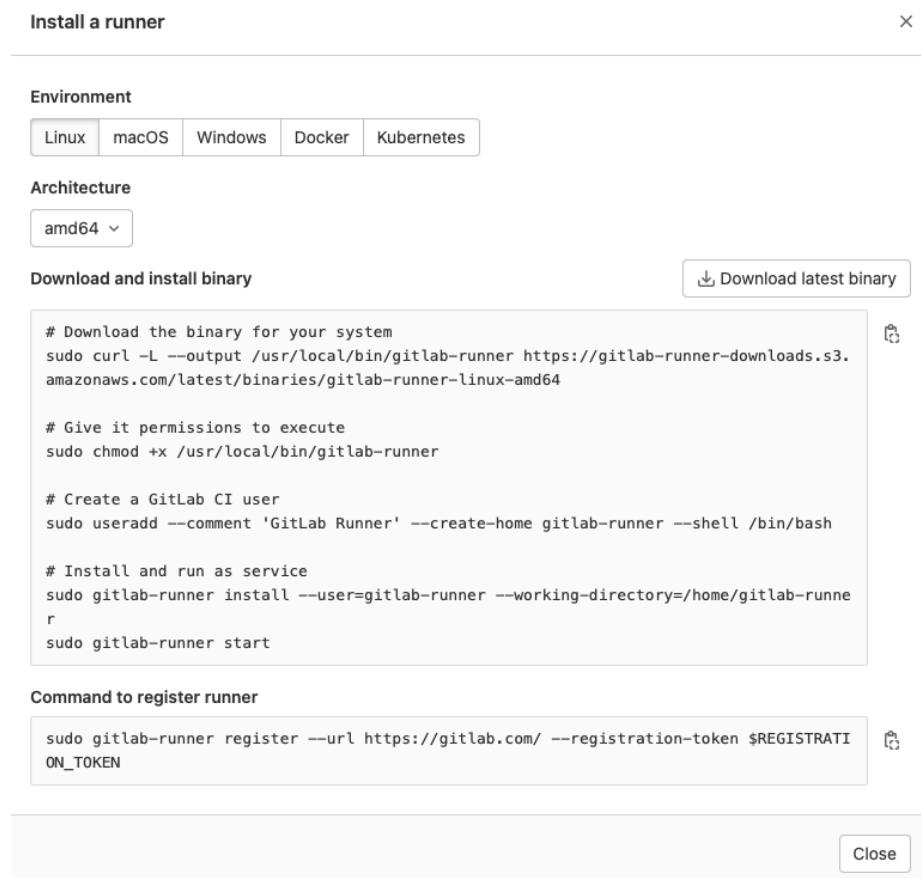


Figure 2: Installer un runner spécifique.

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).

The screenshot shows the 'Runners' settings page in GitLab. On the left sidebar, under 'Project Information', 'Merge requests', and 'CI/CD', the 'Runners' section is selected. The main content area is titled 'Runners' and contains two sections: 'Specific runners' and 'Shared runners'. In the 'Specific runners' section, there is a form to 'Set up a specific Runner for a project' with steps 1 and 2. Step 1 says 'Install GitLab Runner and ensure it's running.' Step 2 says 'Register the runner with this URL: https://gitlab.com/'. Below the form, a registration token is shown. In the 'Available specific runners' list, there is one entry: 'myLocalRunner' (ID #13145631, status green). In the 'Shared runners' section, there is a list of available runners, each with a status icon (green, red, blue), name, and a brief description of its capabilities (e.g., '1-blue.shared-gitlab-org.runners-manager.gitlab.com/default' supports docker, east-e, git, git-anne, linux, mongo, mySQL, postgres, ruby, shared). A button 'Enable shared runners for this project' is visible.

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.

```

1 concurrent = 1
2 check_interval = 0
3
4 [session_server]
5   session_timeout = 1800

```

```

7 [[runners]]
  name = "myLocalRunner"
9  url = "https://gitlab.com/"
  token = "uytryuBN76545fgcv"
11 executor = "docker"
  builds_dir = "/tmp/builds"
13 [runners.custom_build_dir]
[runners.cache]
  [runners.cache.s3]
  [runners.cache.gcs]
17  [runners.cache.azure]
[runners.docker]
  tls_verify = false
  image = "maven:latest"
21  privileged = false
  disable_entrypoint_overwrite = false
23  oom_kill_disable = false
  disable_cache = false
25  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.

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.

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.

New project > Create blank project

Create blank project

Create a blank project to house your files, plan your work, and collaborate on code, among other things.

Project name
My awesome project

Project URL
https://gitlab.com/cocowaterswing/

Project slug
my-awesome-project

Want to house several dependent projects under the same namespace? [Create a group](#).

Project description (optional)

Description format

Visibility Level [?](#)
 [Private](#)
 Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.
 [Public](#)
 The project can be accessed without any authentication.

Project Configuration

Initialize repository with a README
 Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Enable Static Application Security Testing (SAST)
 Analyze your source code for known security vulnerabilities. [Learn more](#).

Create project **Cancel**

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

Nicolas BERTRAND > myTutoProject

Project 'myTutoProject' was successfully created.

myTutoProject [⊕](#)
Project ID: 32672913

Invite your team
Add members to this project and start collaborating with your team.
Invite members

The repository for this project is empty
You can get started by cloning the repository or start adding files to it with one of the following options.

Clone [Upload File](#) [New file](#) [Add README](#) [Add LICENSE](#) [Add CHANGELOG](#) [Add CONTRIBUTING](#)
[Set up CI/CD](#) [Configure Integrations](#)

Command line instructions
You can also upload existing files from your computer using the instructions below.

Git global setup

```
git config --global user.name "Nicolas BERTRAND"
git config --global user.email "cocowaterswing@gmail.com"
```

Create a new repository

```
git clone https://gitlab.com/cocowaterswing/mytutoproject.git
```

Collapse sidebar

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

Command line instructions

You can also upload existing files from your computer using the instructions below.

Git global setup

```
git config --global user.name "Nicolas BERTRAND"
git config --global user.email "cocowaterswing@gmail.com"
```

Create a new repository

```
git clone https://gitlab.com/cocowaterswing/mytutoproject.git
cd mytutoproject
git switch -c main
touch README.md
git add README.md
git commit -m "add README"
git push -u origin main
```

Push an existing folder

```
cd existing_folder
git init --initial-branch=main
git remote add origin https://gitlab.com/cocowaterswing/mytutoproject.git
git add .
git commit -m "Initial commit"
git push -u origin main
```

Push an existing Git repository

```
cd existing_repo
git remote rename origin old-origin
git remote add origin https://gitlab.com/cocowaterswing/mytutoproject.git
git push -u origin --all
git push -u origin --tags
```

« Collapse sidebar

Figure 6: Commandes git à exécuter.

Nicolas BERTRAND > ocr-p11-medhead-poc > Repository

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

main + History Find file Web IDE Clone

Test code clean
Nicolas Bertrand authored 2 days ago

« Collapse sidebar

Figure 7: Repository du projet MedHead.

.gitlab-ci.yml, exemple simple

Afin de paramétriser 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.

```
1 stages:
2   - build
3     - test
4
5 build-job:
6   stage: build
7   script:
8     - echo "Le projet build..."
9
10 test-job:
11   stage: test
12   script:
13     - 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.

.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.

```
1 image: maven:latest # J'ajoute l'image docker que le runner va
2   ↪ utiliser pour exécuter mes scripts
3
4 stages:
5   - build
6   - unit-test
7
8 build-ms-emergency:
9   stage: build
```

Status	Pipeline ID	Triggerer	Commit	Stages	Duration
passed	#442816345	CI	main ~ 08109914 Test code clean	build: passed lint: passed test: passed coverage: passed	00:08:19 2 days ago
passed	#439535620	CI	main ~ db76a759 Excluding test direc...	lint: passed test: passed coverage: passed	00:07:57 1 week ago
passed	#439494375	CI	main ~ 080992ea Deleted emergency...	lint: passed test: passed coverage: passed	00:07:52 1 week ago
passed	#439493299	CI	main ~ bf3a1716 Change ResponseS...	lint: passed test: passed coverage: passed	00:08:19 1 week ago
passed	#435337636	CI	main ~ 5adc282f WebFlux & WebClic...	lint: passed test: passed coverage: passed	00:07:55 2 weeks ago
passed	#431022369	CI	main ~ 7c9f2d10 Revert "no message"	lint: passed test: passed coverage: passed	00:07:43 3 weeks ago
passed	#429992892	CI	main ~ 1a415f48 Code refactor integ...	lint: passed test: passed coverage: passed	00:07:33 3 weeks ago

Figure 8: Exécution du pipeline.

```

9   script:
10    - cd emergency
11    - ./mvnw compile # Commande maven pour compiler le code
12      ↪ source du projet

13 unit-test-ms-emergency:
14   stage: unit-test
15   script:
16     - cd emergency
17     - ./mvnw surefire-report:report # Crée un rapport d'exécution
18       ↪ des tests au format html
19   artifacts:
20     when: always
21     # paths permet de sauvegarder les artefacts générés pendant
22       ↪ l'exécution du script sur le GitLab Server
23     # et de les retrouver dans l'onglet browse du job ou le
24       ↪ bouton download du pipeline
25   paths:
26     - emergency/target/site/surefire-report.html
27   # reports:junit permet de récupérer les artefacts
28     ↪ TEST-com.ocr.medhead.emergency.*.xml
29   #afin d'intégrer les rapports dans l'onglet test des détails
30     ↪ d'un job
31   reports:

```

```
27    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).

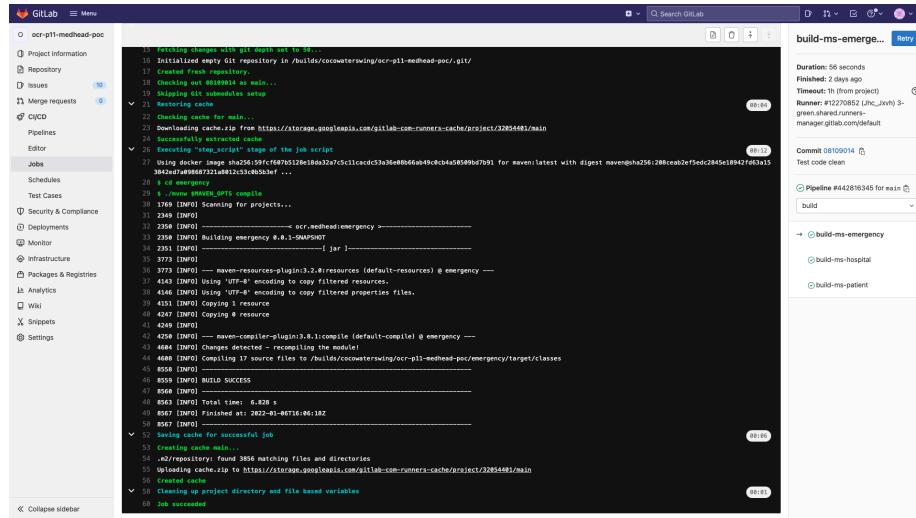


Figure 9: Visualisation du résultat du build.

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).

.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

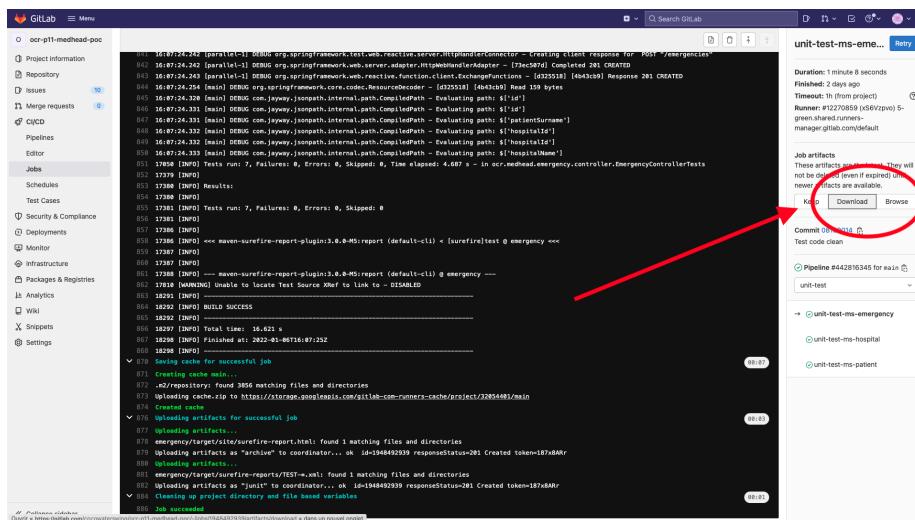


Figure 10: Télécharger ou visualiser un rapport d'exécution des tests unitaires

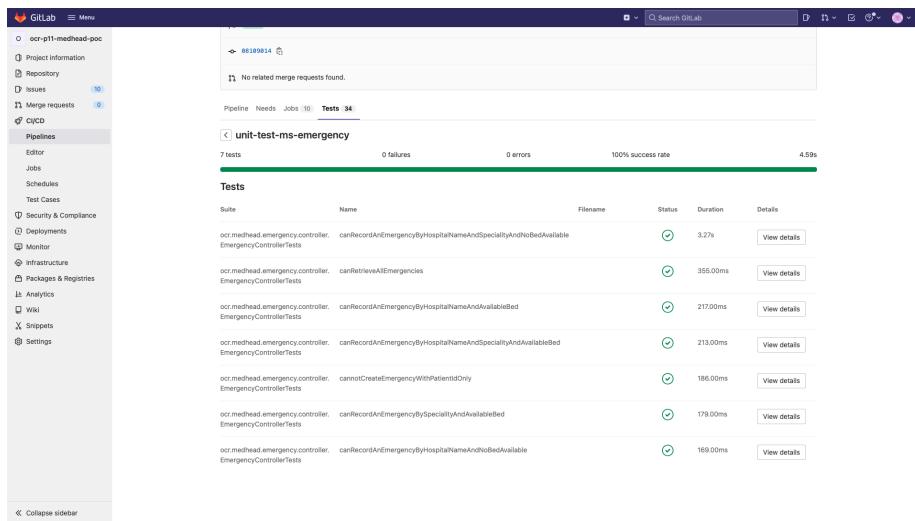


Figure 11: Intégration du rapport d'exécution des tests unitaires dans GitLab.

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
2
stages:
4   - build
5   - unit-test
6   - coverage
8
build-ms-emergency:
...
10
unit-test-ms-emergency:
...
12
14 coverage-ms-emergency:
15   stage: coverage
16   script:
17     - cd emergency
18     # Le plugin JaCoCo (Java Code Coverage) génère un rapport de
19       ↪ couverture du code source par les tests
20     - ./mvnw jacoco:report
21
artifacts:
22   when: always
23   # paths permet de sauvegarder les artefacts générés pendant
24       ↪ l'exécution du script sur le GitLab Server
25   # et de les retrouver dans l'onglet browse du job ou download
26       ↪ du pipeline
27
paths:
28   - emergency/target/site/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

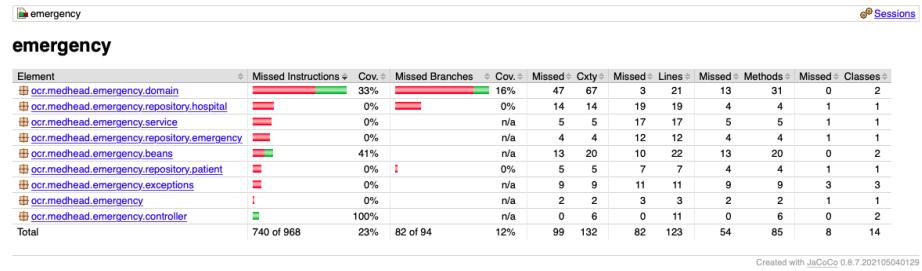


Figure 12: Rapport html généré par JaCoCo.

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

```

1 image: maven:latest

3 stages:
4   - build
5   - unit-test
6   - coverage
7   - quality

9 build-ms-emergency:
10 ...
11
12 unit-test-ms-emergency:
13 ...
14
15 coverage-ms-emergency:
16 ...
17
18 code_quality_job:
19   stage: quality
20   image: docker:stable
21   services:
22     - docker:stable-dind
23   script:

```

```

25   - mkdir codequality-results
26   - docker run
27     --env CODECLIMATE_CODE="$PWD"
28     --volume "$PWD":/code
29     --volume /var/run/docker.sock:/var/run/docker.sock
30     --volume /tmp/cc:/tmp/cc
31     codeclimate/codeclimate analyze -f html >
32       ↳ ./codequality-results/index.html
33 artifacts:
34   paths:
35     - codequality-results/

```

.codeclimate.yml

```

1 plugins:
2   sonar-java:
3     enabled: true
4     config:
5       sonar.java.source: "17"
6 exclude_patterns:
7   - "**/.mvn/"
8   - "**/target/"
9   - "**/test/"

```

The screenshot shows a browser-based Code Climate interface. At the top, there's a navigation bar with icons for code, issues, and settings. Below it is a search bar and a dropdown menu for filters (Category: All Categories, Engine: All Engines). The main area is titled 'issues'.

The first snippet is from `emergency/strategic/api/src/main/java/com/medeo/medeo/api/hospital/repository/HospitalRepository.java`. It highlights a method named `findHospital` with a cognitive complexity of 6, suggesting refactoring. The code shows a loop where it checks if the hospital's name matches the patient's location and then creates a new `Hospital` object with the URL from the discovery client.

The second snippet is from `hospital/ermain/java/com/medeo/medeo/api/hospital/controller/HospitalControllerImpl.java`. It highlights a method `findBySpecialtyAndBeds` with a cognitive complexity of 6, suggesting refactoring. The code shows a loop through hospitals and filtering them based on their specialty and bed count.

At the bottom of each snippet, there's a 'Details' button and a note indicating similar blocks of code were found in other locations.

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.

```
1 image: maven:latest

3 stages:
  - build
  - unit-test
  - coverage
  - quality
  - package
9
build-ms-emergency:
11 ...
13 unit-test-ms-emergency:
15 ...
15 coverage-ms-emergency:
17 ...
19 code_quality_job:
21 ...
21
package-ms-emergency:
23   stage: package
24   script:
25     - cd emergency
26     - 'mvn deploy -s ci_settings.xml'
27   artifacts:
28     when: always
29     paths:
      - emergency/target/*.jar
```

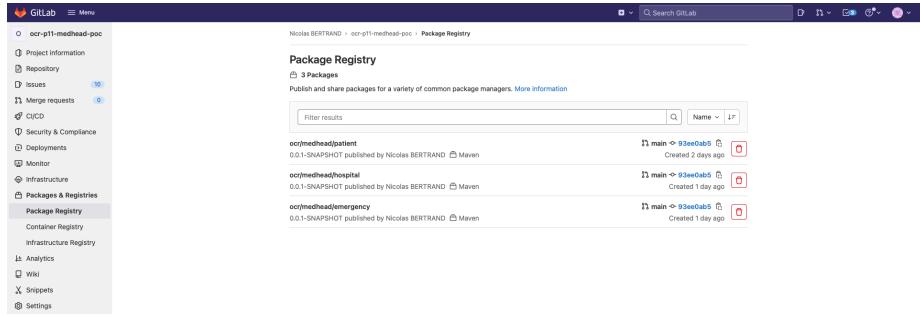


Figure 14: Visualisation du Package Registry de GitLab.

.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.

```

1 image: maven:latest
2
3 stages:
4   - build
5   - unit-test
6   - coverage
7   - quality
8   - package
9   - docker
10
11 build-ms-emergency:
12 ...
13
14 unit-test-ms-emergency:
15 ...
16
17 coverage-ms-emergency:
18 ...

```

```

20 code_quality_job:
...
22
23 package-ms-emergency:
...
24
25
26 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}" | base64 | tr -d '\n')\"}}}"
      ↪ > /kaniko/.docker/config.json
  - >-
    /kaniko/executor
    --context "${CI_PROJECT_DIR}"
    --dockerfile "${CI_PROJECT_DIR}/emergency/Dockerfile"
    --destination
      ↪ "${CI_REGISTRY_IMAGE}/emergency:${CI_COMMIT_TAG}"

```

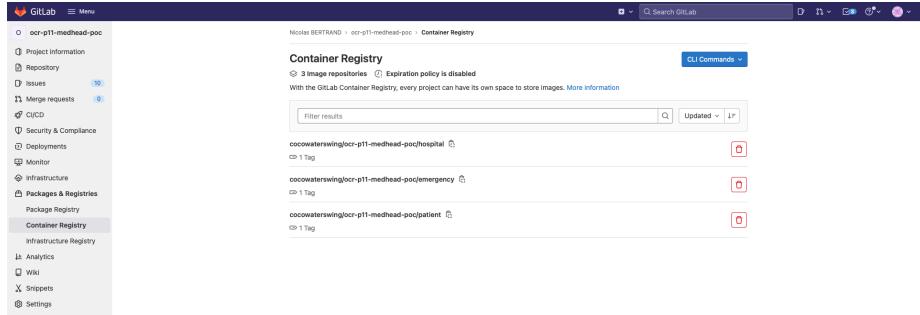


Figure 15: Visualisation du Container Registry de GitLab.

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

'The One DevOps Platform | GitLab'. Accessed 29 June 2022. <https://about.gitlab.com/>.

'How to Prevent Crypto Mining Abuse on GitLab.Com SaaS | GitLab'. Accessed 29 June 2022. <https://about.gitlab.com/blog/2021/05/17/prevent-crypto-mining-abuse/>.

'Installation | GitLab'. Accessed 29 June 2022. <https://docs.gitlab.com/ee/install/>.

'Install GitLab Runner | GitLab'. Accessed 29 June 2022. <https://docs.gitlab.com/runner/install/>.

'Registering Runners | GitLab'. Accessed 29 June 2022. <https://docs.gitlab.com/runner/register/>.

'Configuring GitLab Runner | GitLab'. Accessed 29 June 2022. <https://docs.gitlab.com/runner/configuration/>.

'GitLab Runner Commands | GitLab'. Accessed 29 June 2022. <https://docs.gitlab.com/runner/commands/>.

'Maven – Introduction to the Build Lifecycle'. Accessed 29 June 2022. <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>.

Bertrand, Nicolas. MedHead. Accessed 29 June 2022. <https://gitlab.com/cocowaterswing/ocr-p11-medhead-poc>.

'OpenClassrooms'. Accessed 29 June 2022. <https://openclassrooms.com/fr/>.

'Test Coverage Visualization | GitLab'. Accessed 29 June 2022. https://docs.gitlab.com/ee/ci/testing/test_coverage_visualization.html.

Liens attachés à un paragraphe du document

'Get Started with GitLab CI/CD | GitLab'. Accessed 29 June 2022. https://docs.gitlab.com/ee/ci/quick_start/.

'Job Artifacts | GitLab'. Accessed 29 June 2022. https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html.

‘The `.Gitlab-Ci.Yml` File | GitLab’. Accessed 29 June 2022. https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html.

‘Maven Surefire Report Plugin – Surefire-Report:Report’. Accessed 29 June 2022. <https://maven.apache.org/surefire/maven-surefire-report-plugin/report-mojo.html>.

‘JaCoCo - Maven Plug-In’. Accessed 29 June 2022. <https://www.jacoco.org/jacoco/trunk/doc/maven.html>.

‘Code Quality | GitLab’. Accessed 29 June 2022. https://docs.gitlab.com/ee/ci/testing/code_quality.html.

Code Climate. ‘SonarJava’. Accessed 29 June 2022. <https://docs.codeclimate.com/docs/sonar-java>.

‘Use Docker to Build Docker Images | GitLab’. Accessed 29 June 2022. https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#use-the-docker-executor-with-the-docker-image-docker-in-docker.

‘Maven Packages in the Package Repository | GitLab’. Accessed 29 June 2022. https://docs.gitlab.com/ee/user/packages/maven_repository/#create-maven-packages-with-gitlab-cicd-by-using-maven.

‘Apache Maven Deploy Plugin – Introduction’. Accessed 29 June 2022. <https://maven.apache.org/plugins/maven-deploy-plugin/>.

‘Use Kaniko to Build Docker Images | GitLab’. Accessed 29 June 2022. https://docs.gitlab.com/ee/ci/docker/using_kaniko.html.

Vidéos

Valentin Despa. Gitlab CI Pipeline Tutorial for Beginners, 2018. <https://www.youtube.com/watch?v=Jav4vbUrqII>.

GitLab Unfiltered. 1. Switzerland GitLab Meetup: First Time GitLab & CI/CD Workshop with Michael Friedrich, 2021. <https://www.youtube.com/watch?v=kTNfi5z6Uvk>.

GitLab Unfiltered. GitLab Virtual Meetup - Intro to GitLab CI Featuring Michael Friedrich, 2020. https://www.youtube.com/watch?v=l5705U8s_nQ.

GitLab Unfiltered. Getting Started with GitLab CI/CD, 2020. <https://www.youtube.com/watch?v=sIegJaLy2ug>.

GitLab Unfiltered. GitLab Code Quality: Speed Run, 2020. <https://www.youtube.com/watch?v=B32LxtJKo9M>.

GDG Toulouse. Philippe Charrière - Se Former ‘En Douceur’ à GitLab, GitLab CI & CD Avec OpenFaas, 2018. https://www.youtube.com/watch?v=xQOcv_Xg-BY.

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 paramétriser 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.

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

```

1  image: adoptopenjdk/openjdk11:jdk-11.0.11_9-alpine
2  # Nous installons l'image docker Alpine qui est une distribution Linux ultra-légère
3  # que le runner va utiliser pour exécuter nos scripts avec le JDK JAVA 11
4
5  stages:
6    - build
7    - tests
8    - .post
9
10 build-ms-ERS-Emergency-Responder:
11   stage: build
12   script:
13     - cd ERS-Emergency-Responder
14     - ./mvnw compile # Commande maven pour compiler le code source du projet
15     - echo "Le projet ERS-Emergency-Responder build..."
16
17 tests-ms-ERS-Emergency-Responder:
18   stage: tests
19
20   script:
21     - cd ERS-Emergency-Responder
22     - ./mvnw surefire-report:report # Crée un rapport d'exécution des tests au format html
23     - echo "Les tests ERS-Emergency-Responder s'exécutent..."
24
25 artifacts:
26   when: always
27   # paths permet de sauvegarder les artefacts générés pendant l'exécution du script sur le GitLab Server
28   # et de les retrouver dans l'onglet browse du job ou le bouton download du pipeline
29   paths:
30     - ERS-Emergency-Responder/target/site/surefire-report.html
31     - ERS-Emergency-Responder/target/surefire-reports/TEST-*.xml
32   # reports:junit permet de récupérer les artefacts de Test xml afin d'intégrer les rapports
33   # dans l'onglet test des détails d'un job
34   reports:
35     junit:
36       - ERS-Emergency-Responder/target/surefire-reports/TEST-*.xml
37
38 check-tests-failure:
39   stage: .post
40   dependencies:
41     - tests-ms-ERS-Emergency-Responder
42   script:
43     - (! grep "<failure" ERS-Emergency-Responder/target/surefire-reports/TEST-*.xml)
44     - (! grep "<error" ERS-Emergency-Responder/target/surefire-reports/TEST-*.xml)

```

Figure 1: Exemple d'action GitLab.

et d'intégration. Le troisième stage sera chargé de vérifier si les tests ont réussi 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).

```
9 variables:  
10   MYSQL_DATABASE_NAME: hospitals  
11   MYSQL_DATABASE: hospitals  
12   MYSQL_ROOT_PASSWORD: ""  
13   MYSQL_ALLOW_EMPTY_PASSWORD: 1  
14   MYSQL_USER: admin  
15   MYSQL_PASSWORD: 1234  
16   MYSQL_HOST: mariadb  
17   MYSQL_PORT: 3306  
18
```

Figure 2: Configuration des variables MySQL.

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

```
27 tests-ms-ERS-Emergency-Responder:
28   stage: tests
29
30   services:
31     - mariadb:latest
32
33   script:
34     - apk --no-cache add mysql-client
35     - mysql --version
36     - sleep 20
37     - mysql --host=mariadb -P 3306 --user=root --password="" "${MYSQL_DATABASE}" < data.sql
38     - mysql --host=mariadb -P 3306 --user=root --password="" -e 'SHOW TABLES FROM `hospitals`';
39     - cd ERS-Emergency-Responder
40     - ./mvnw surefire-report:report # Crée un rapport d'exécution des tests au format html
41     - echo "Les tests ERS-Emergency-Responder s'exécutent..."
```

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 pourrons 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`.

The screenshot shows the 'Variables' section of a GitLab project settings page. On the left, there's a sidebar with project navigation links like 'Project information', 'Repository', 'Issues', 'Merge requests', 'CI/CD', 'Security & Compliance', 'Deployments', 'Monitor', 'Infrastructure', 'Packages & Registries', 'Analytics', 'Wiki', 'Snippets', 'Settings', 'General', 'Integrations', and 'Webhooks'. The main area is titled 'Variables' and contains a sub-header: 'Variables store information, like passwords and secret keys, that you can use in job scripts. Learn more.' Below this, it says 'Variables can be:' with two bullet points: 'Protected: Only exposed to protected branches or tags.' and 'Masked: Hidden in job logs. Must match masking requirements. Learn more.' A note at the top right says 'Environment variables are configured by your administrator to be protected by default.' A table lists five variables:

Type	Key	Value	Protected	Masked	Environments
Variable	MYSQL_HOST	*****	✓	✗	All (default)
Variable	MYSQL_PASSWORD	*****	✓	✗	All (default)
Variable	MYSQL_PORT	*****	✓	✗	All (default)
Variable	MYSQL_ROOT_PASSWORD	*****	✓	✗	All (default)
Variable	spring.datasource.url	*****	✓	✗	All (default)

At the bottom of the table are two buttons: 'Add variable' and 'Reveal values'.

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 (*Figure 5*).

```
spring.datasource.url=jdbc:mysql://localhost:3306/hospitals
```

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 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 a échoué (indépendant de la réussite des tests ou de leurs échecs) - *Figure 6*.

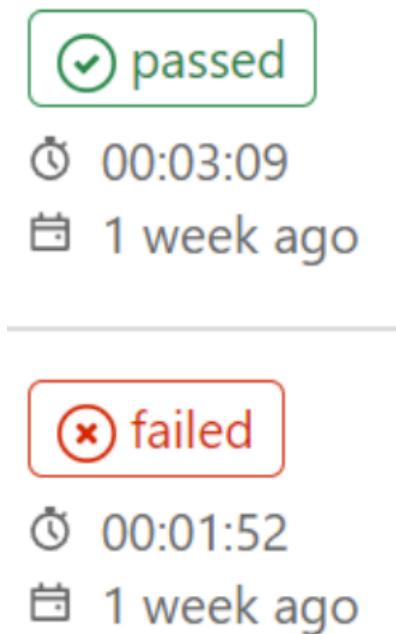
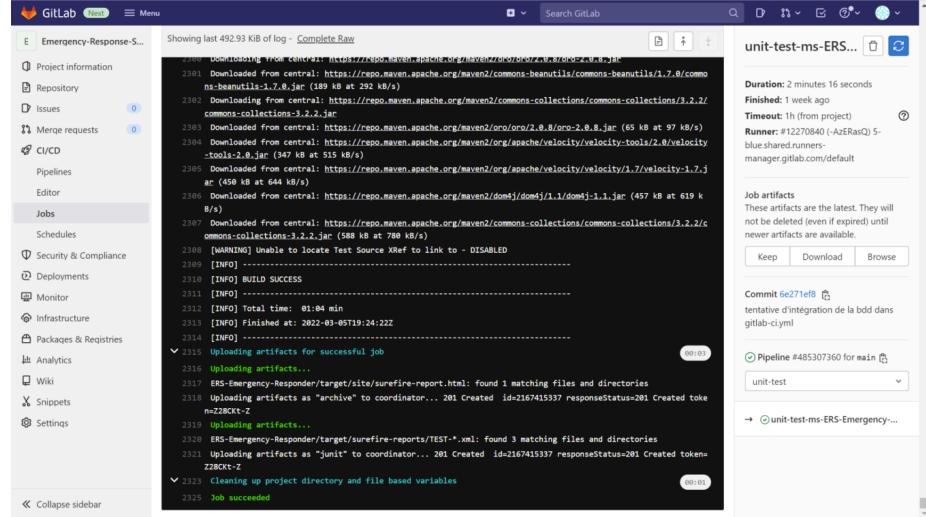


Figure 6: Statut de la pipeline.

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



The screenshot shows the GitLab interface with a pipeline named 'unit-test-ms-ERS...'. The pipeline status is 'Duration: 2 minutes 16 seconds' and it was 'Finished: 1 week ago'. It used a 'blue shared runners' runner with ID #12270840. The logs are displayed in a terminal-like window, showing the process of downloading dependencies from central Maven repositories, building the project, and uploading artifacts. The final message indicates a successful job completion.

```

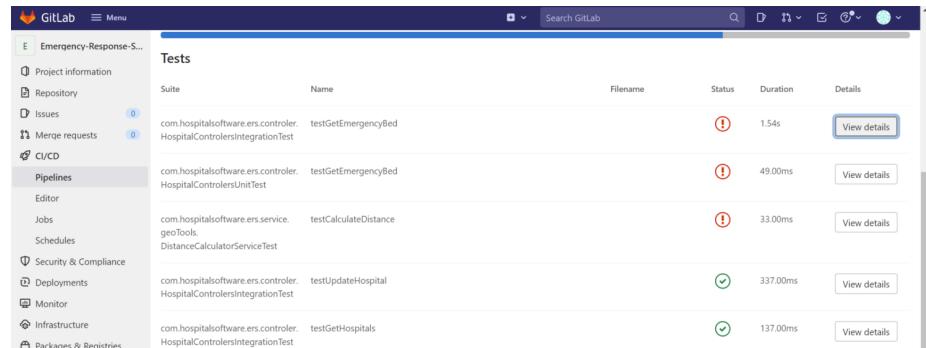
Showing last 492.93 kB of log - Complete Raw
2300 Downloaded from central: https://repo.maven.apache.org/maven2/commons-beanutils/commons-beanutils/1.7.0/commons-beanutils-1.7.0.jar (159 kB at 202 kB/s)
2301 Downloaded from central: https://repo.maven.apache.org/maven2/commons-beanutils/commons-beanutils/1.7.0/commons-beanutils-1.7.0.jar (159 kB at 202 kB/s)
2302 Downloading from central: https://repo.maven.apache.org/maven2/commons-collections/commons-collections/3.2.2/commons-collections-3.2.2.jar
2303 Downloaded from central: https://repo.maven.apache.org/maven2/oro/oro/2.0.8/oro-2.0.8.jar (65 kB at 97 kB/s)
2304 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/velocity-tools/2.0/velocity-tools-2.0.jar (347 kB at 515 kB/s)
2305 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/velocity/velocity/1.7/velocity-1.7.jar (459 kB at 64 kB/s)
2306 Downloaded from central: https://repo.maven.apache.org/maven2/commons-collections/commons-collections/3.2.2/commons-collections-3.2.2.jar (588 kB at 788 kB/s)
2307 [WARNING] Unable to locate Test Source XRef to link to - DISABLED
2308 [INFO] ...
2309 [INFO] BUILD SUCCESS
2310 [INFO] ...
2311 [INFO] -----
2312 [INFO] Total time: 01:04 min
2313 [INFO] Finished at: 2022-03-05T19:26:22Z
2314 [INFO] -----
2315 Uploading artifacts for successful job
2316 Uploading artifacts...
2317 ERS-emergency-Responder/target/site/surefire-report.html: found 1 matching files and directories
2318 Uploading artifacts as "archive" to coordinator... 201 Created id=2167415337 responseStatus=201 Created token=mcZBCKXt-2
2319 Uploading artifacts...
2320 ERS-emergency-Responder/target/surefire-reports/TEST-.xml: found 3 matching files and directories
2321 Uploading artifacts as "junit" to coordinator... 201 Created id=2167415337 responseStatus=201 Created token=mcZBCKXt-2
2322 Cleaning up project directory and file based variables
2323 Job succeeded

```

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) :



The screenshot shows the 'Tests' section of a pipeline run. It lists several test suites and their details:

Suite	Name	Filename	Status	Duration	Details
com.hospitalsoftware.ers.controller.HospitalControllersIntegrationTest	testGetEmergencyBed		!	1.54s	<button>View details</button>
com.hospitalsoftware.ers.controller.HospitalControllersUnitTest	testGetEmergencyBed		!	49.00ms	<button>View details</button>
com.hospitalsoftware.ers.service.geoTools.DistanceCalculatorServiceTest	testCalculateDistance		!	33.00ms	<button>View details</button>
com.hospitalsoftware.ers.controller.HospitalControllersIntegrationTest	testUpdateHospital		✓	337.00ms	<button>View details</button>
com.hospitalsoftware.ers.controller.HospitalControllersIntegrationTest	testGetHospitals		✓	137.00ms	<button>View details</button>

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

'Secure Your Application | GitLab'. Accessed 29 June 2022. https://docs.gitlab.com/ee/user/application_security/.

'Using External Secrets in CI | GitLab'. Accessed 29 June 2022. <https://docs.gitlab.com/ee/ci/secrets/>.

'Web API Fuzz Testing | GitLab'. Accessed 29 June 2022. https://docs.gitlab.com/ee/user/application_security/api_fuzzing/.

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 coloré. 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 à 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*).

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

L'utilisateur clique 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.LucasGitHub: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 :

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



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

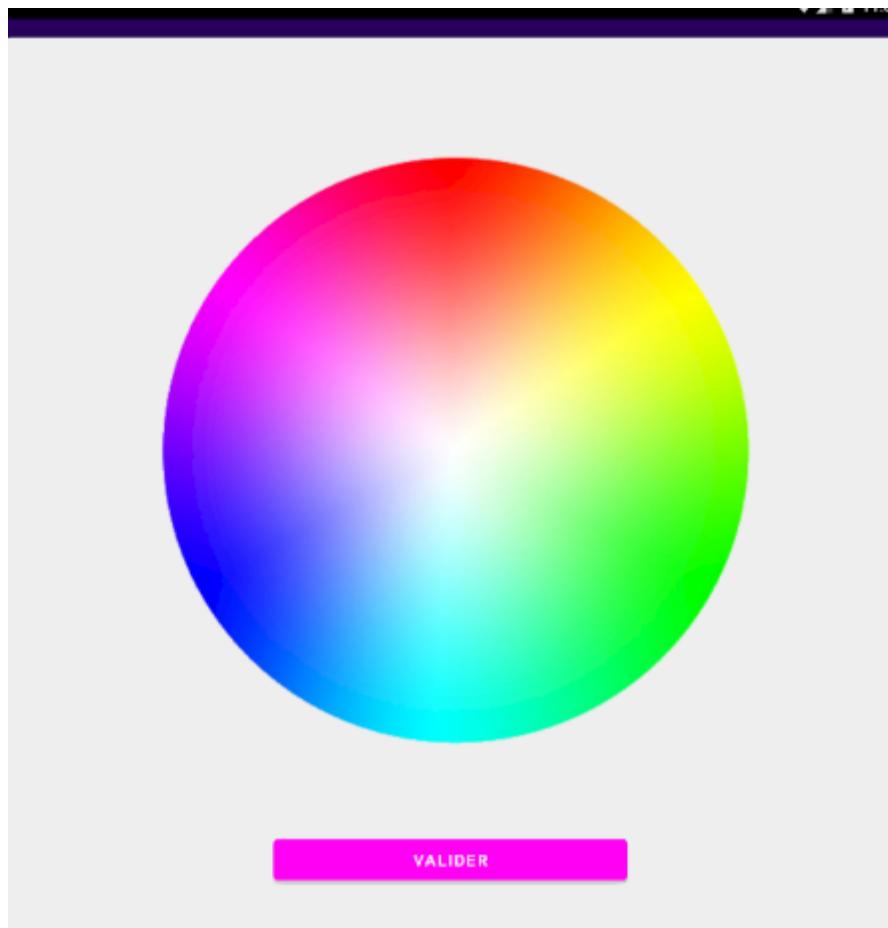


Figure 2: Visualisation de l'interface permettant de sélectionner une couleur.

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

Amelin, Lucas. ColorPicker. Java, 2022. <https://github.com/LucasGitHubz/ColorPicker>.

Etude exploratoire pour la production de Médias interactifs

Evan, David

Le-Point-Technique, June/2022

abstract: Ce document est une étude exploratoire des solutions pour la production de médias interactifs - type vidéos 360 ou avec scénario.

keywords: Etude exploratoire, médias interactifs, Unity, AWS Elemental Media Store

Introduction

Dans cette étude, nous nous intéressons aux différentes options architecturales pour l'implémentation d'un système informatique destiné à la production de médias interactifs, comme la production de vidéos 360 ou avec scénario.

Choix préférés pour les outils et technologies

Bien que les solutions puissent être des conceptions « from scratch », **les solutions préexistantes** seront favorisées dès lorsqu'elles répondent aux besoins définis dans le cahier des charges et que la tarification est adaptée. Cette approche vise à permettre d'assurer une livraison rapide du projet, une réduction des coûts de mise en œuvre et favorise une approche modulaire.

Les choix d'outils et de technologies retenues devront répondre à des critères de cohérence d'ensemble. Les solutions compatibles les unes par rapport aux autres et/ou facilement interopérables seront préférées. Cette approche vise à favoriser l'évolutivité de l'architecture retenue.

Notons que les critères de popularités des outils, de facilité à trouver des ressources et des profils de collaborateurs expérimentés seront aussi analysés pour le choix des solutions.

Le coût des licences et l'adéquation au budget du projet (non défini au moment de la rédaction de ce document) seront pris en compte pour le choix final des solutions.

Technologies pour la diffusion des médias vidéo sur le web

Le streaming de médias vidéo (interactifs ou non) sur le web nécessite l'utilisation de formats adaptés afin de garantir la meilleure qualité selon les performances de la connexion internet de l'utilisateur final.

Les protocoles disposant d'un bitrate adaptatif seront préférés afin de satisfaire à cette exigence.

Protocoles HLS et DASH

Les protocoles HLS et MPEG-DASH seront retenus comme technologie de diffusion pour le projet de média interactif.

De manière simplifiée, HLS (*HTTP Live Streaming*) et DASH (*Dynamic Adaptive Streaming over HTTP*) sont deux protocoles de streaming audio / vidéo basé sur HTTP et visant à délivrer des médias à l'utilisateur en « fragmentant » les fichiers d'origines en plusieurs « sous-fichiers » de qualités différentes afin de fournir le plus adapté à l'utilisateur en fonction de l'avancement de sa visualisation.

HLS et DASH supportent tous deux les principaux codec audio / vidéo de l'industrie : *VP9, AAC-LC, FLAC, H.265, H.264 ... (non-exhaustif)*.

Le protocole DASH embarque plusieurs fonctionnalités supplémentaires par rapport au HLS, notamment la possibilité d'utiliser des DRM sur les médias transportés.

Bien que l'implémentation de DASH soit relativement plus complexe, le fonctionnement général est relativement similaire. (*DASH ne sera pas approfondi dans cette section. Si nécessaire, des ressources documentaires sont disponibles en fin de section pour approfondir la compréhension de ces technologies*)

Le schéma ci-après (*Figure 1*) présente de façon simplifiée le fonctionnement d'un flux HLS :

Comme indiqué précédemment, la découpe des fichiers source en segments de qualité différentes permet l'adaptation du bitrate de la diffusion au fur et à mesure de l'avancé afin de garantir à l'utilisateur une lecture fluide en sacrifiant la qualité sur certains passage (lorsque la connexion de l'utilisateur n'arrive plus à supporter le flux).

Un manifeste est fourni en complément et permet de décrire au client (navigateur web) comment construire la vidéo.

La figure ci-après (*Figure 2*) montre le fonctionnement du principe de segmentation du fichier source en sections différentes.

Documentation complémentaire :

- <https://www.wowza.com/blog/mpeg-dash-dynamic-adaptive-streaming-over-http>
- <https://blog.eleven-labs.com/fr/video-live-dash-hls>

Technologies pour l'authentification / l'autorisation

Afin de disposer d'une solution évolutive et pour permettre de simplifier l'ensemble des mécanismes d'authentification et d'autorisation, les technologies

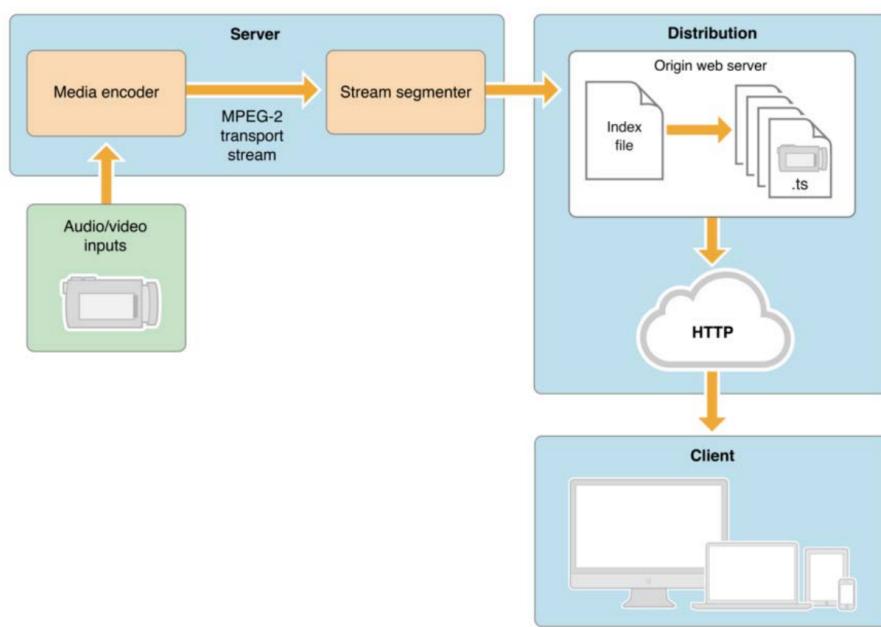


Figure 1: Vue synthétique d'un envoi réalisé avec HLS (source : Eleven Labs Blog).

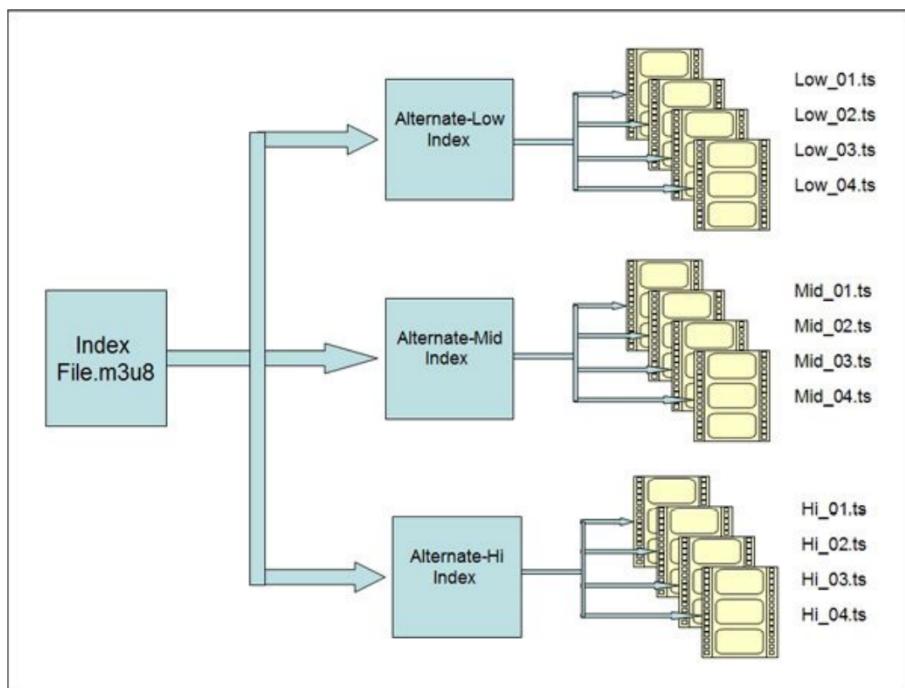


Figure 2: Segmentation d'un média en différentes qualités pour diffusion via HLS (source : Eleven Labs Blog).

OAuth2 et sa couche d'identité OIDC (*Open ID Connect*) seront utilisées pour la gestion des accès aux ressources via un mécanisme de jeton d'accès (*access_tokens*).

Les niveaux d'autorisation seront gérés à l'aide des scopes embarqués dans les jetons OAuth2 et les profils utilisateurs à l'aide des jetons d'identité (*id_tokens*). *Étant hors du scope de ce document de définition d'architecture, les mécanismes de fonctionnement de l'authentification / autorisation seront abordés dans les spécifications techniques.*

Documentation complémentaire :

- <https://datatracker.ietf.org/doc/html/rfc6749>
- <https://openid.net/connect/>

Briques de solution de référence (SBB)

Solution : Production des médias interactifs (SBB-1)

La production de médias interactifs nécessite l'utilisation d'outils complets et performants afin de couvrir l'ensemble des besoins décrits dans le cahier des charges d'architecture (vidéo 360, vidéo interactives, multi-view ...).

Notons par ailleurs que la production de vidéo 360° nécessite par ailleurs des outils complémentaires pour la capture, le stitching et le montage des séquences.

Le logiciel **Unity** (et sa plateforme) sera retenu comme outil de production de médias interactif. Ce logiciel offre une gamme très large d'outils pour la création de média et couvre 100% des besoins définis. Notons qu'un système de « plugins » téléchargeable via le **Unity Asset Store** permet d'enrichir les fonctionnalités offertes par le logiciel de base.

Documentation complémentaire :

- <https://unity.com/fr/solutions/film-animation-cinematics>
- <https://unity.com/fr/solutions/360video>
- <https://assetstore.unity.com/>

Solution : Transcodage / Convertisseur média (SBB-2)

L'un des besoins décrits dans les briques d'architecture de référence consiste dans le transcodage des médias produits (format de sortie de l'outil de production) au format compatible avec la diffusion sur le web (par exemple, HLS / DASH comme décrit dans les technologies retenues).

La solution **AWS Elemental Media Converter** (*Figure 3*) sera utilisée pour la conversion des médias sources. Cet outil couvre l'intégralité des besoins nécessaires au projet et s'adapte au volume nécessaire (tarification à la minute

convertie). Le workflow peut être automatisé et couplé avec AWS S3 pour automatiser l'ensemble de chaîne et libérer les créateurs de contenu de ce travail.

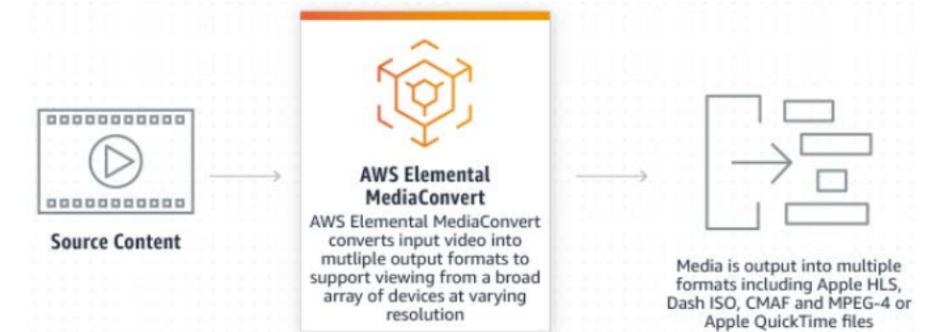


Figure 3: Fonctionnement d'AWS Elemental Media Converter (Source : AWS).

Documentation complémentaire :

- <https://aws.amazon.com/fr/mediaconvert/>

Solution : Stockage des médias (SBB-3)

Afin de permettre le stockage de l'ensemble des médias produits et en attente de diffusion (version drafts, médias non publiés ...) ainsi que les ressources nécessaires à la production (audio, vidéos brutes ...), un espace de stockage de haute capacité est nécessaire.

La solution **AWS S3** (*Simple Storage Service*) sera retenue. Cette solution répond parfaitement aux besoins de la société tout en permettant une intégration simplifiée au système d'autorisation basé sur les rôles (identique pour les clients et les collaborateurs) afin de garantir la sécurité du contenu.

Cette solution est totalement évolutive et garantit une adaptation parfaite aux besoins grandissant de l'entreprise. Le SLA extrêmement important garantit la disponibilité des données.

Documentation complémentaire :

- <https://aws.amazon.com/fr/s3/>

Solution : Diffusion de média sur le web (SBB-4)

La diffusion sur de média sur le web à grande échelle nécessite de prendre en compte les besoins de haute disponibilité, de régularité et de faible latence pour la distribution du contenu. Le service doit par ailleurs disposer de fonctionnalités de scalabilité automatique afin de s'adapter aux pics de diffusion. Plusieurs

options peuvent être envisagées pour le choix de la brique de solution retenu pour l'architecture.

Option 1 : Solution AWS Elemental MediaStore **AWS Elemental Media Store** est un service fourni par la plateforme AWS et est spécialisé dans le stockage et la distribution de vidéo. Il fournit un point de stockage et assure la diffusion des médias tout en s'adaptant automatiquement à la demande.

L'intégration avec les autres services AWS (*IAM* et *CDN* par exemple) en fait une solution de choix pour l'architecture du projet. Notons que la tarification pour la plateforme peut toutefois devenir un frein à la croissance de l'entreprise notamment pour le contenu non premium et faiblement monétisable.

Documentation complémentaire :

- <https://aws.amazon.com/fr/mediastore/>

Option 2 : Solution Unity Multiplay / Unity Build Server La plate-forme Unity embarque une solution alternative, **Unity Multiplay**. Cette solution fournit une plateforme auto-scalable initialement prévue pour la diffusion de jeux vidéo mais pouvant parfaitement couvrir nos besoins de diffusion de média à haute échelle.

Unity Multiplay offre l'avantage de s'intégrer aux service **Unity Build Server** permettant d'envisager la création de l'intégralité de la couche «front-end» avec Unity (C#).

Par ailleurs, l'absence de tarification sur les flux de données entrant / sortant peut représenter une source importante d'économie.

Notons toutefois que cette plateforme impose plusieurs choix technologiques, notamment sur l'OS embarqué (Windows Server 2012, Windows Server 2019 ou Ubuntu 18.04) et dispose d'une communauté et d'une documentation moins importante que la solution AWS.

Documentation complémentaire :

- <https://unity.com/fr/products/multiplay>
- <https://docs.unity.com/multiplay/shared/welcome-to-multiplay.html>

Avantages et inconvénients des deux options

Option	Solution	Avantages	Inconvénients
1	AWS Elemental Media Store	<ul style="list-style-type: none"> Totalement intégré à l'écosystème AWS (Ressources et autres services) Auto-scalable Très hautes performances Disponible sous forme de service Faible configuration (SaaS) 	<ul style="list-style-type: none"> Pas de personnalisation possible des services. Peut nécessiter l'ajout de frontaux (CDN) Tarification sur le stockage et sur les flux de données entrant / sortant
2	Unity Multiplay	<ul style="list-style-type: none"> Totalement personnalisable Tarification avantageuse (uniquement sur les performances de la plateforme) Auto-scalable Intégration des frontaux de base. 	<ul style="list-style-type: none"> Configuration et intégration pouvant être complexe (IaaS) Peu de ressources documentaires Peut être intégré au workflow automatisé mais nécessite un travail important.

Choix de la solution Les deux solutions présentées pour la diffusion de média peuvent être retenues pour la conception de l'architecture.

Au vu des enjeux du projet, il semble nécessaire de réaliser un **PoC** sur la solution Unity afin de garantir sa cohérence vis-à-vis du projet et permettre ainsi un ROI substantiel par rapport à la solution AWS.

Solution : Authentification / Autorisation (SBB-5)

L'autorisation et l'authentification nécessite de faire appel à une solution IAM (Identity Access Manager) compatible avec les standards OAuth2 et OIDC afin de disposer d'une couche « universelle » d'authentification / d'identification / d'autorisation.

Bien que de nombreuses solutions soient envisageables (Okta, Gravitee, Azure AD ...), la solution **AWS Cognito** sera retenue. Ce choix se justifie par sa parfaite intégration à l'environnement AWS permettant de facilement configurer les autres services (*S3, Media Converter, Media Store, API Gateway ...*) pour prendre en compte les règles à appliquer.

Documentation complémentaire :

- <https://aws.amazon.com/fr/cognito/>

Solution : Visualisation des médias (SBB-6)

La visualisation des médias doit répondre à deux caractéristiques : Fournir une plateforme web pour la présentation des médias (ex : SPA, WebApp) et fournir les outils, notamment le player vidéo, compatibles avec l'ensemble des exigences fonctionnelles : lecture de média disposant d'une interaction avec l'utilisateur, vidéo « 360 » (l'utilisateur peut « déplacer » la caméra), support HLS / DASH ...

La contrainte du multi-plateforme doit être respectée afin que la WebApp puisse être facilement accessible depuis les supports désignés dans les exigences non fonctionnelles du cahier des charges d'architecture.

La WebApp sera développée en interne (« from scratch ») en utilisant les technologies adaptées (React, Vue.Js, Angular pour le front-end, Java Spring, PHP Laravel / Symfony pour le back-end) en fonction de l'écosystème de l'entreprise et de l'expertise des développeurs.

La solution **NexPlayer** sera retenue comme player vidéo. Cette solution se présente sous la forme d'un SDK permettant de créer un player vidéo totalement personnalisable à partir d'une solution couvrant la totalité des besoins exprimés (vidéo 360, multi-view, Dynamic Streaming ...)

Ce choix se justifie par plusieurs critères : - Le player vidéo couvre l'ensemble des besoins décrit dans le cahier des charges. - La solution est totalement personnalisable, permettant à l'entreprise de pouvoir créer sa propre UI adaptée à l'image de marque, voire de développer de nouvelles fonctionnalités. - L'outil est open source et dispose d'une implémentation sur les principales plateformes (Web HTML5, Application mobiles, consoles de jeux, smart TV ...) - Le player dispose d'une excellente implémentation dans les outils Unity.

Documentation complémentaire :

- <https://nexplayersdk.com/>
- https://github.com/NexPlayer/NexPlayer_Unity_Plugin

Synthèse de l'étude de la stack technologique

Catalogue des briques de solution de référence

Référence	Id. SBB	Solution	Rôle
Référence	Id. SBB	ABB	Rôle
	SBB-1	ABB-1	Unity
	SBB-2	ABB-2	AWS Elemental Media Converter
	SBB-3	ABB-3	AWS S3 (<i>Simple Storage Service</i>)
			Production médias interactif
			Transcodage / Convertisseur media
			Stockage des médias

Id. SBB	Référence aux ABB	Solution	Rôle
SBB-4-A	ABB-4	AWS Elemental MediaStore	Diffusion de média sur le web
SBB-4-B	ABB-4	Unity Multiplay	Diffusion de média sur le web
SBB-5	ABB-5	AWS Cognito	Authentification / Autorisation
SBB-6	ABB-6	1. Solution custom (Web App) 2. NextPlayer (Player vidéo)	Visualisation des médias

Implémentation de la stack technologique

l'implémentation de la stack technologique proposée vise à exploiter au maximum des services déjà disponibles pour faciliter l'implémentation et accélérer les livraisons. L'ensemble des composants déployés devront être redondés afin de garantir la continuité de service en cas de défaillance ou de maintenance sur tout ou partie des composants.

Des composants additionnels (base de données SQL / NoSQL, MoM, Back-end API) seront probablement nécessaires pour compléter la solution (notamment la web app).

Notons toutefois, ces aspects sortent de la définition de l'architecture et seront précisés lors de la rédaction des spécifications techniques que chaque composants.

Références

Wowza Media Systems. ‘MPEG-DASH: Dynamic Adaptive Streaming Over HTTP Explained’, 18 April 2022. <https://www.wowza.com/blog/mpeg-dash-dynamic-adaptive-streaming-over-http>.

Bernard, Jérémie. ‘Les principaux formats de flux video live DASH et HLS’. Blog Eleven Labs, 19 July 2017. <https://blog.eleven-labs.com/fr/video-live-dash-hls/>.

Hardt, Dick. ‘The OAuth 2.0 Authorization Framework’. Request for Comments. Internet Engineering Task Force, October 2012. <https://doi.org/10.17487/RFC6749>.

‘OpenID Connect | OpenID’, 1 August 2011. <http://openid.net/connect/>.

Technologies, Unity. ‘Logiciel d’animation en 3D pour le cinéma et la télévision | Unity’. Accessed 29 June 2022. <https://unity.com/fr/solutions/film-animation-cinematics>.

‘Créez Des Expériences Vidéo Immersives En 360° | Unity’. Accessed 29 June 2022. <https://unity.com/fr/solutions/360video>.

‘Unity Asset Store - The Best Assets for Game Making’. Accessed 29 June 2022. <https://assetstore.unity.com/>.

‘AWS Elemental MediaConvert’. Accessed 29 June 2022. <https://aws.amazon.com/fr/mediaconvert/>.

Amazon Web Services, Inc. ‘AWS | Amazon S3 – Stockage de données en ligne dans le cloud’. Accessed 29 June 2022. <https://aws.amazon.com/fr/s3/>.

Amazon Web Services, Inc. ‘AWS Elemental MediaStore’. Accessed 29 June 2022. <https://aws.amazon.com/fr/mediastore/>.

‘Hébergement de Serveur de Jeu Multijoueur | Unity Multiplay’. Accessed 29 June 2022. <https://unity.com/fr/products/multiplay>.

‘Welcome to Multiplay’. Accessed 29 June 2022. <https://docs.unity.com/multiplay/shared/welcome-to-multiplay.html>.

Amazon Web Services, Inc. ‘AWS | Amazon Cognito - gestion des accès et synchronisation des données’. Accessed 29 June 2022. <https://aws.amazon.com/fr/cognito/>.

NexPlayer. ‘NexPlayer | The Premium Multiscreen Player SDK for Video Apps’. Accessed 29 June 2022. <https://nexplayersdk.com/>.

Wade, Noah. *NexPlayer_Unity_Plugin*. 2017. Reprint, NexPlayer, 2017. https://github.com/NexPlayer/NexPlayer_Unity_Plugin.