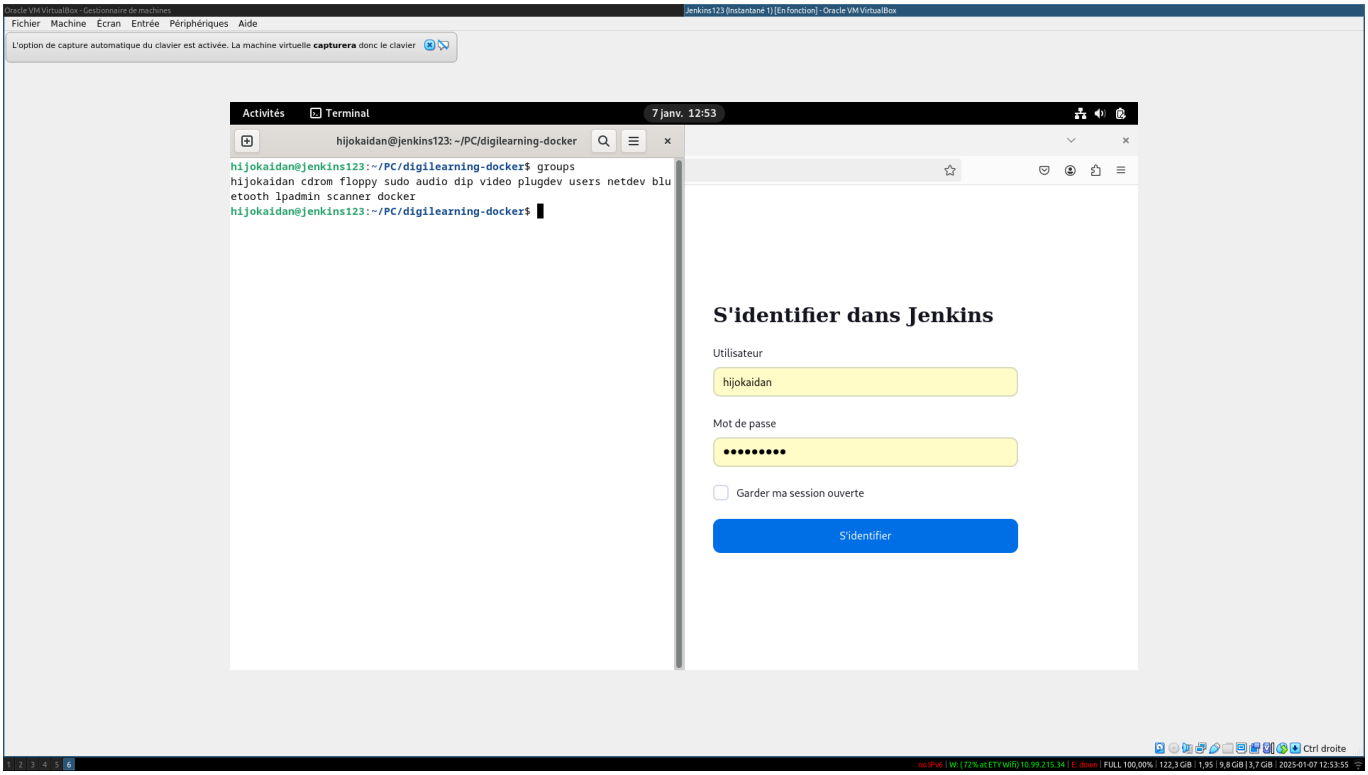
Documentation de déploiement V1

1. Mise en place du serveur Jenkins.

La première étape consiste à mettre en place une VM Debian 12 pour pouvoir accueillir le serveur Jenkins.

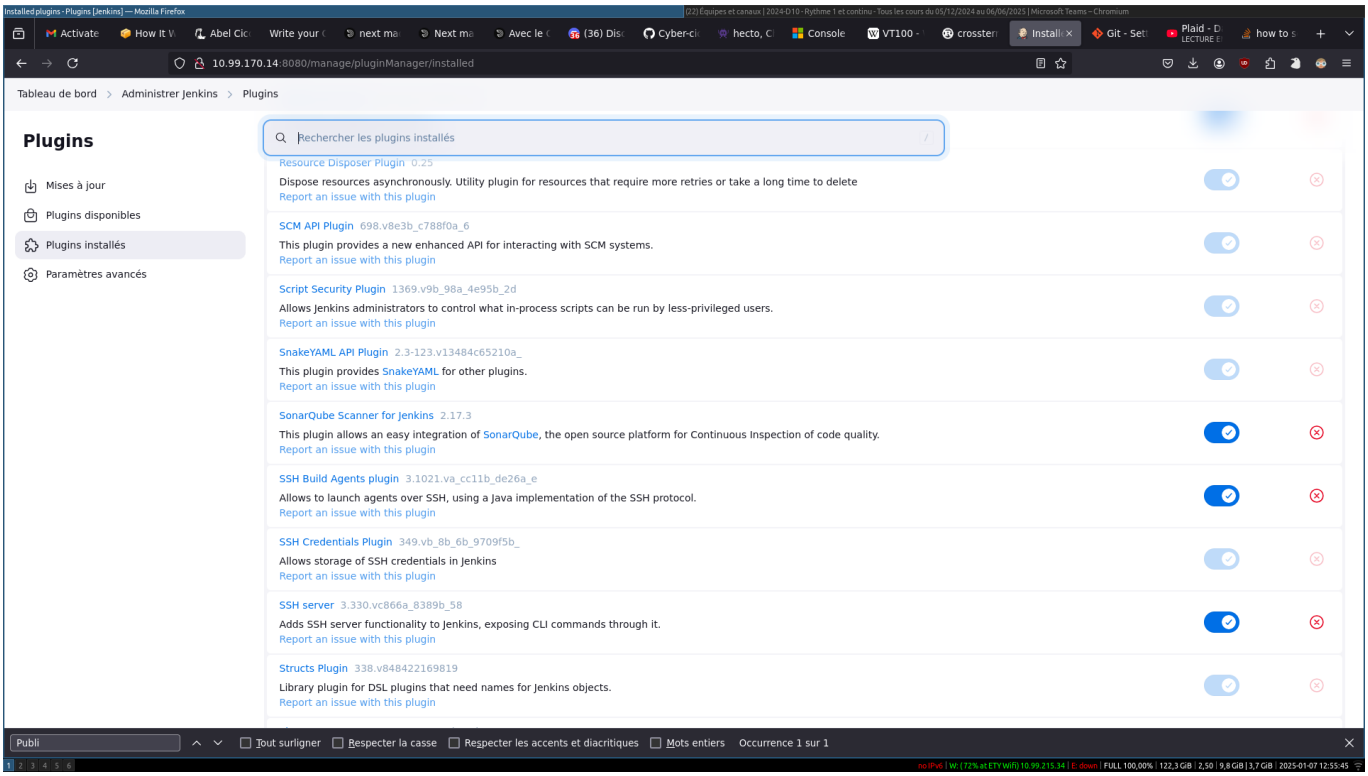
Ensuite, il faut ajouter l'utilisateur principal dans la liste des sudoers, récupérer les packets nécessaires à l 'installation de docker, et ajouter le groupe "docker" à l'utilisateur principal.



Une fois ceci fait, il suffit alors d'aller récupérer l'image docker de Jenkins, faire une redirection de port, configurer le mot de passe de l'administrateur, et récupérer les plugins nécessaires.

Pour cela, on va dans `Administrer Jenkins > Plugins`, on récupère les plugins pour Sonarqube et JaCoCo, et on redémarre Jenkins.

Ensuite, on ajoute l'outil `Maven 3.9.9` à Jenkins, pour pouvoir l'utiliser dans la pipeline.

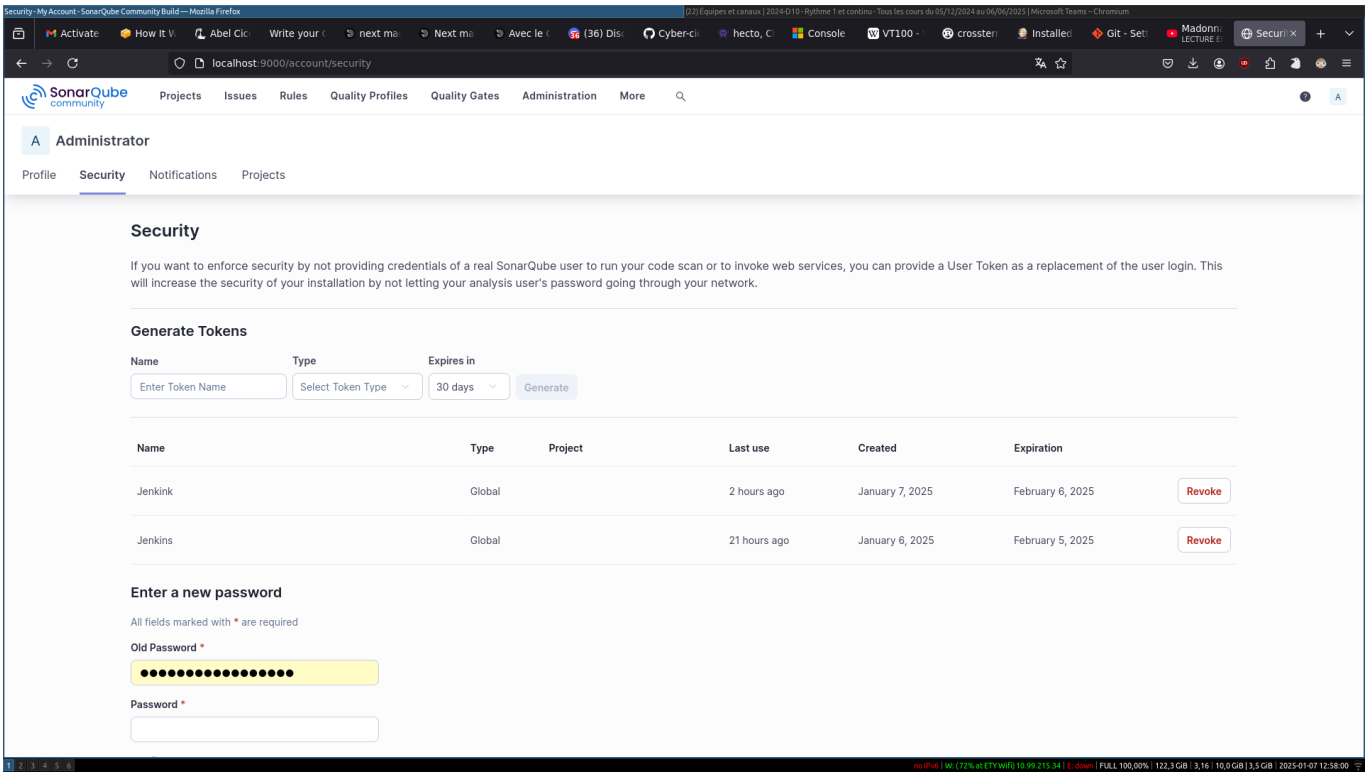


Enfin, dernière étape de la mise en place de notre serveur vient après avoir mis en place le serveur Sonarqube : on doit ajouter les credentials pour se connecter à Sonarqube, ainsi que l'URL du serveur Sonarqube

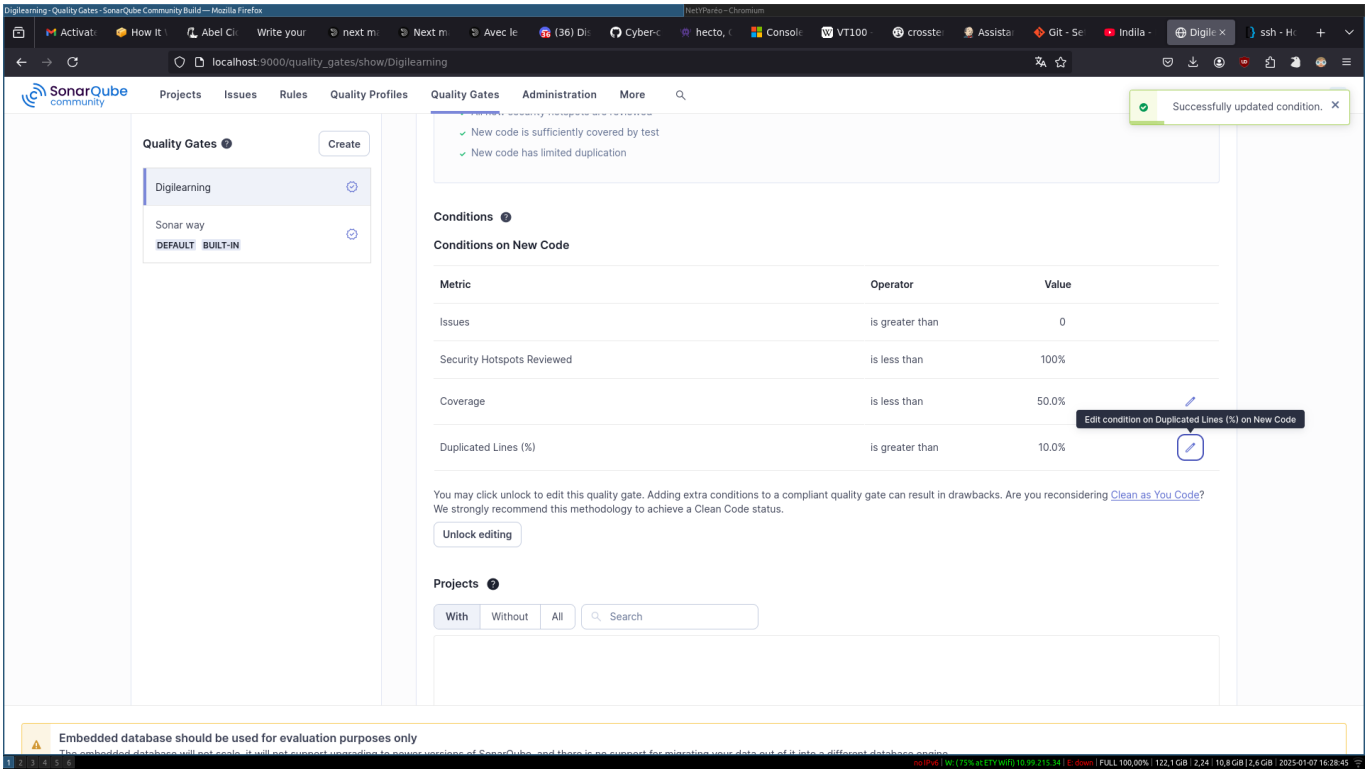
2. Mise en place du serveur Sonarqube.

Sur un système d'exploitation Debian contenant Docker dans le même réseau que le serveur Jenkins, on lance le docker Sonarqube.

Une fois le nouveau mot de passe ajouté, on peut maintenant configurer un nouveau projet local, dont le nom et la clé serviront dans le script présent dans le Jenkinsfile pour récupérer des données à propos du build.



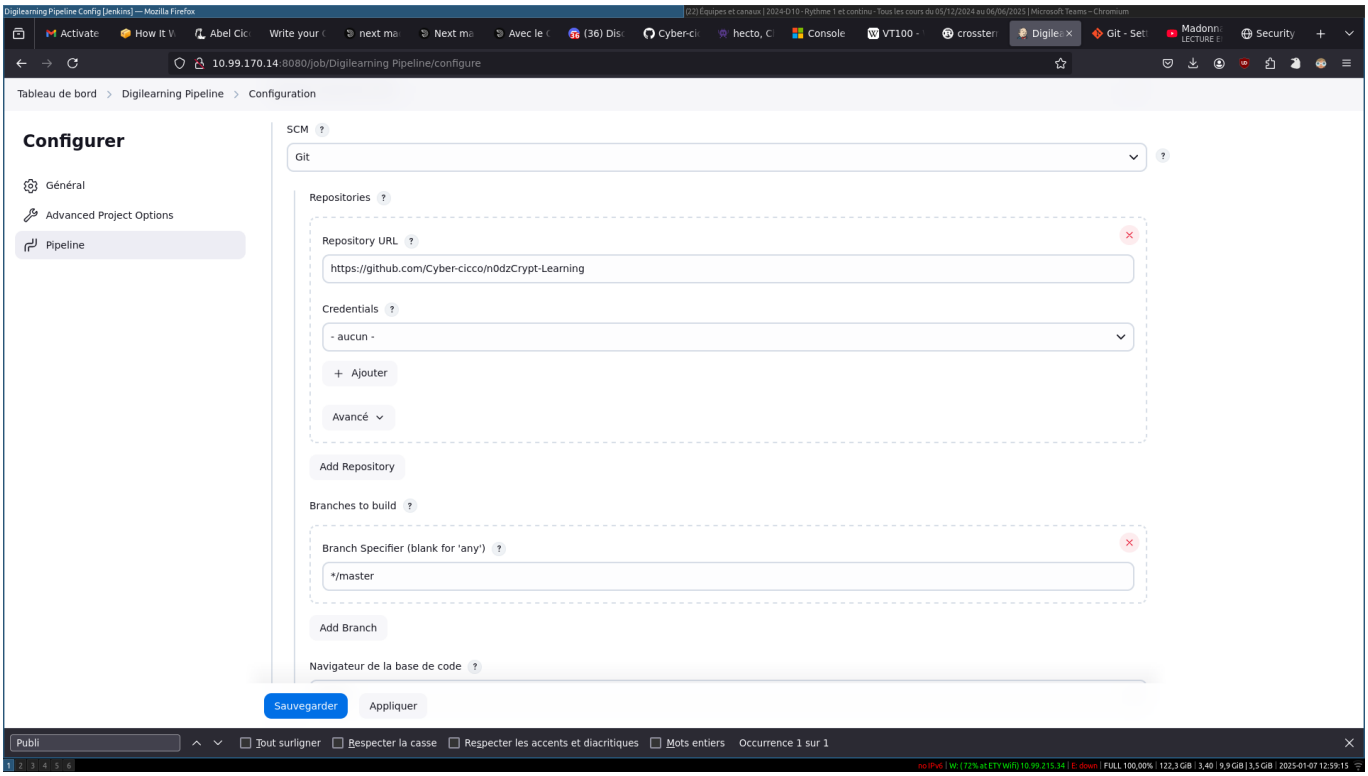
On ajoute également une quality gate que l'on associe au projet pour s'assurer que de renvoyer une erreur si le projet est dans un état non satisfaisant en terme de qualité de code.



3. Mise en place du build.

3.1 Ajout du repo.

Pour mettre en place la pipeline, j'ai décidé d'utiliser un repo public appelé "Nodzcrypt-learning", une de mes applications Java. J'ai donc ajouté l'URL du repo et mis en place un token d'authentification sur github.



3.2 Travail préparatoire dans le Jenkinsfile.

Pour pouvoir effectuer mes actions, j'avais besoin de définir l'outil Maven dans le script, ainsi que le repo à récupérer

```
pipeline {
    agent any
    tools {
        maven 'Maven 3.9.9'
    }
    stages {
        // Récupération du repo distant.
        stage('Checkout') {
            steps {
                git 'https://github.com/Cyber-cicco/n0dzCrypt-Learning.git'
            }
        }
        // Les différents éléments de ma pipeline
    }
}
```

3.3 Ajout des tests et du build

La première étape la plus facile consiste à ajouter les deux éléments ne demandant que de configurer maven.

Dans le repo, il y a un fichier de test unitaire pour des utilitaires permettant de manipuler des chaines de caractères.

```
package fr.diginamic.digilearning.utils;

import org.junit.jupiter.api.Test;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.*;

class StringUtilsTest {

    @Test
    void isDigits() {
        assertTrue(StringUtils.isDigits("123345"));
        assertTrue(StringUtils.isDigits("1"));
        assertFalse(StringUtils.isDigits(""));
        assertFalse(StringUtils.isDigits("12 3"));
    }

    @Test
    void getIndexOfCounterIfPresent() {
        assertEquals(4,
StringUtils.getIndexOfCounterIfPresent("test_1").get());
        assertEquals(5,
StringUtils.getIndexOfCounterIfPresent("test__1").get());
        assertEquals(5,
StringUtils.getIndexOfCounterIfPresent("test__11111111").get());
        assertEquals(Optional.empty(),
StringUtils.getIndexOfCounterIfPresent("test-1"));
    }

}
```

Ces tests utilisent le framework de test proposé par Spring Boot, ainsi que junit. Pour cela, il suffisait de rajouter l'entrée suivante au pom.xml :

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
```

Enfin, il fallait simplement lancer les tests via maven dans la pipeline en ajoutant ce stage :

```
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
```

Pour le build, il suffit d'ajouter ce stage à la pipeline:

```
        stage('Build') {
            steps {
                sh 'mvn clean package -DskipTests'
            }
        }
```

Et l'on peut également, comme pour le rapport de JaCoCo, récupérer un artifact du build via cette commande :

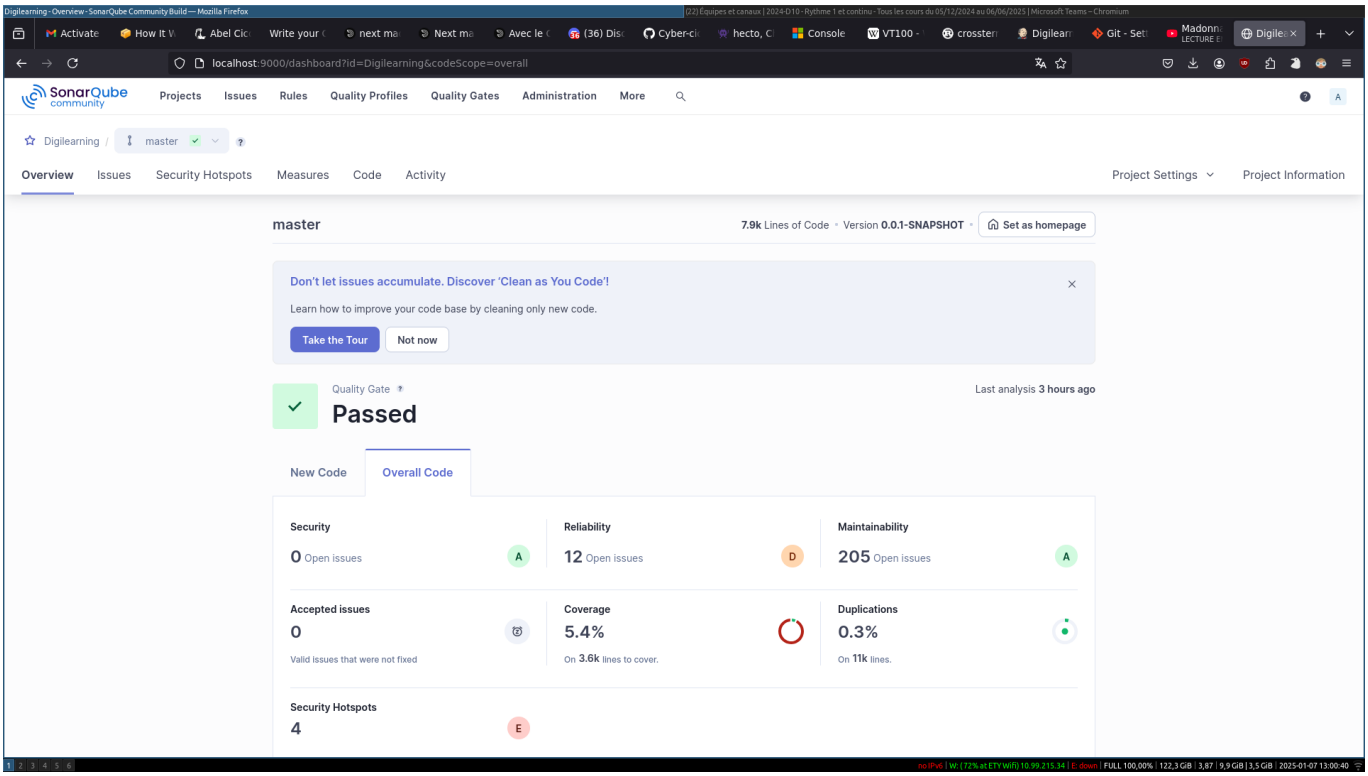
```
        stage('Archive Build') {
            steps {
                archiveArtifacts artifacts: 'target/digi-learning-0.0.1-
SNAPSHOT.jar', allowEmptyArchive: true
            }
        }
```

3.4 Ajout de Sonarqube

Après avoir créé l'installation Sonarqube ayant pour identifiant 'SonarQ', et après avoir créer sur Sonarqube un projet local avec pour nom et clé "Digilearning", on pouvait simplement ajouter le stage suivant pour lancer le scan Sonarqube via Maven:

```
        stage('SonarQube Analysis') {
            steps {
                 script {
                     def mvnHome = tool 'Maven 3.9.9'
                     withSonarQubeEnv('SonarQ') {
                         sh "${mvnHome}/bin/mvn clean verify sonar:sonar -
Dsonar.projectKey='Digilearning' -Dsonar.projectName='Digilearning'"
                     }
                 }
             }
        }
```

On obtiens alors ce résultat:



3.5 Ajout de JaCoCo.

On ajoute le plugin JaCoCo à la fois dans Jenkins (comme vu dans la mise en place du serveur) et dans le pom.xml comme ceci :

```
            <plugin>
                <groupId>org.jacoco</groupId>
                <artifactId>jacoco-maven-plugin</artifactId>
                <version>0.8.10</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>prepare-agent</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>report</id>
                        <phase>verify</phase>
                        <goals>
                            <goal>report</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
```

Ensuite, on ajoute un stage exécutant la commande maven de vérification:

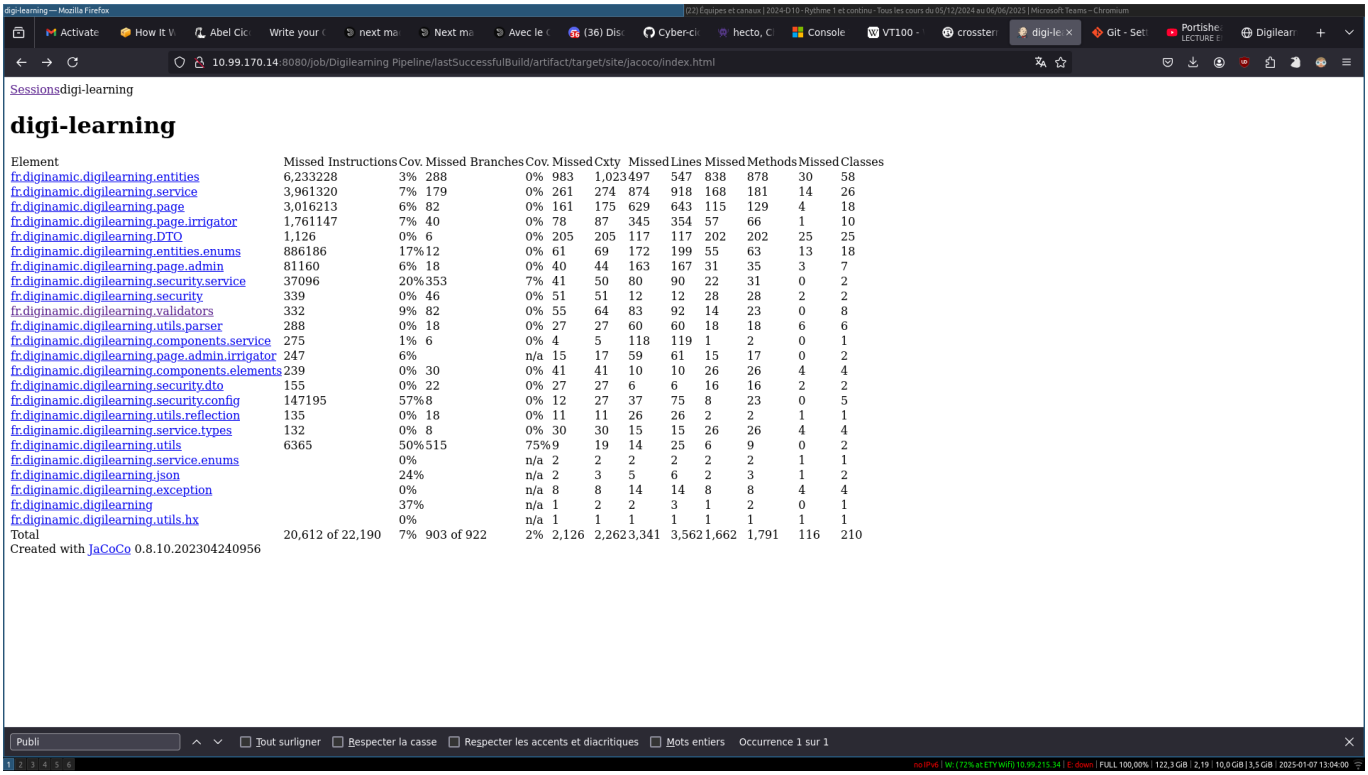
```
        stage('Code Coverage') {
            steps {
```

```
                sh 'mvn verify'
            }
        }
```

Ainsi que la commande permettant de récupérer le rapport HTML de JaCoCo sous forme d'archive du build:

```
        stage('Archive Coverage Report') {
            steps {
                // Archive the JaCoCo coverage report
                archiveArtifacts artifacts: 'target/site/jacoco/*.html',
allowEmptyArchive: true
            }
        }
```

Et voici le résultat:



Sessions digi-learning

digi-learning

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.diginamic.digilearning.entities	6,233	228 3%	288	0%	983	1,023	497	547	838	878	30	58
fr.diginamic.digilearning.service	3,961	320 7%	179	0%	261	274	874	918	168	181	14	26
fr.diginamic.digilearning.page	3,016	213 6%	82	0%	161	175	629	643	115	129	4	18
fr.diginamic.digilearning.page.irrigator	1,761	147 7%	40	0%	78	87	345	354	57	66	1	10
fr.diginamic.digilearning.DTO	1,126	0%	6	0%	205	205	117	117	202	202	25	25
fr.diginamic.digilearning.entities.enums	886	186 17%	12	0%	61	69	172	199	55	63	13	18
fr.diginamic.digilearning.page.admin	811	60 6%	18	0%	40	44	163	167	31	35	3	7
fr.diginamic.digilearning.security.service	370	96 20%	35	3 7%	41	50	80	90	22	31	0	2
fr.diginamic.digilearning.security	339	0%	46	0%	51	51	12	12	28	28	2	2
fr.diginamic.digilearning.validators	332	9%	82	0%	55	64	83	92	14	23	0	8
fr.diginamic.digilearning.utils.parser	288	0%	18	0%	27	27	60	60	18	18	6	6
fr.diginamic.digilearning.components.service	275	1%	6	0%	4	5	118	119	1	2	0	1
fr.diginamic.digilearning.page.admin.irrigator	247	6%		n/a	15	17	59	61	15	17	0	2
fr.diginamic.digilearning.components.elements	239	0%	30	0%	41	41	10	10	26	26	4	4
fr.diginamic.digilearning.security.dto	155	0%	22	0%	27	27	6	6	16	16	2	2
fr.diginamic.digilearning.security.config	147	195 57%	8	0%	12	27	37	75	8	23	0	5
fr.diginamic.digilearning.utils.reflection	135	0%	18	0%	11	11	26	26	2	2	1	1
fr.diginamic.digilearning.service.types	132	0%	8	0%	30	30	15	15	26	26	4	4
fr.diginamic.digilearning.utils	636	5 50%	51	5 75%	9	19	14	25	6	9	0	2
fr.diginamic.digilearning.service.enums		0%		n/a	2	2	2	2	2	2	1	1
fr.diginamic.digilearning.json		24%		n/a	2	3	5	6	2	3	1	2
fr.diginamic.digilearning.exception		0%		n/a	8	8	14	14	8	8	4	4
fr.diginamic.digilearning		37%		n/a	1	2	2	3	1	2	0	1
fr.diginamic.digilearning.utils.hx		0%		n/a	1	1	1	1	1	1	1	1
Total	20,612 of 22,190	7%	903 of 922	2%	2,126	2,262	3,341	3,562	1,662	1,791	116	210

Created with JaCoCo 0.8.10.202304240956

3.6 Déploiement

Pour déployer, on va copier le Jar via ssh vers la machine remote.

```
        stage('Deploy') {
            steps {
                sshagent(['SSH-1']) {
                    sh 'scp target/digi-learning-0.0.1-SNAPSHOT.jar
hijokaidan@10.99.215.34:/home/hijokaidan/sites/digi-learning/'
                    sh 'ssh hijokaidan@10.99.215.34
./home/hijokaidan/sites/digi-learning/setup.sh'
                }
```

```
            }
        }
```

L'adresse IP doit correspondre à un serveur ayant un port SSH ouvert et correctement configuré pour recevoir la clé privée.

3.7 Version finale

Voici donc la version finale du Jenkinsfile

```
pipeline {
    agent any
    tools {
        maven 'Maven 3.9.9'
    }
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/Cyber-cicco/n0dzCrypt-Learning.git'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Code Coverage') {
            steps {
                sh 'mvn verify'
            }
        }
        stage('Archive Coverage Report') {
            steps {
                // Archive the JaCoCo coverage report
                archiveArtifacts artifacts: 'target/site/jacoco/*.html',
allowEmptyArchive: true
            }
        }
        stage('Build') {
            steps {
                sh 'mvn clean package -Dskiptests'
            }
        }
        stage('Archive Build') {
            steps {
                archiveArtifacts artifacts: 'target/digi-learning-0.0.1-
SNAPSHOT.jar', allowEmptyArchive: true
            }
        }
        stage('SonarQube Analysis') {
            steps {
```

```
                script {
                    def mvnHome = tool 'Maven 3.9.9'
                    withSonarQubeEnv('SonarQ') {
                        sh "${mvnHome}/bin/mvn clean verify sonar:sonar -
Dsonar.projectKey='Digilearning' -Dsonar.projectName='Digilearning'"
                    }
                }
            }
        }
        stage('Deploy') {
            steps {
                sshagent(['SSH-1']) {
                    sh 'scp target/digi-learning-0.0.1-SNAPSHOT.jar
hijokaidan@10.99.215.34:/home/hijokaidan/sites/digi-learning/'
                    sh 'ssh hijokaidan@10.99.215.34
./home/hijokaidan/sites/digi-learning/setup.sh'
                }
            }
        }
    }
}
```

4. Configuration du serveur de déploiement.

Nous avons tester deux solutions pour déployer notre code sur le serveur.

La première copie le résultat du build via SSH.

La seconde, plus expérimentale, utilise les fonctionalité de git pour copier le code source sur le serveur et utiliser les git hooks pour build le code source via Docker

4.1 Avec SSH

Il faut tout d'abord configuré un serveur SSH sur le serveur de destination.

Pour cela, on utilise `ssh-keygen` pour créer une clé privé et publique.

On ajoute ensuite la clé privée dans les credentials de Jenkins.

Enfin, on met la clé publique dans les "authorized_keys" de l'utilisateur avec lequel on souhaite se connecter au serveur.

Et en ayant un DockerFile tel que celui-ci dans le dossier `/home/hijokaidan/sites/digilearning/` sur le serveur :

```
#
# Package
#
FROM azul/zulu-openjdk-alpine:17-latest
WORKDIR /digi-learning
COPY ./*.jar app.jar
```

```
EXPOSE 8090
ENTRYPOINT ["java","-jar","app.jar"]
```

on peut, avec ce script nommé /home/hijokaidan/sites/digi-learning/setup.sh exécutable sur le serveur :

```
#!/bin/bash

docker compose down
docker stop digilearning || true
docker rm digilearning || true
docker build -t digilearning .
docker compose up
```

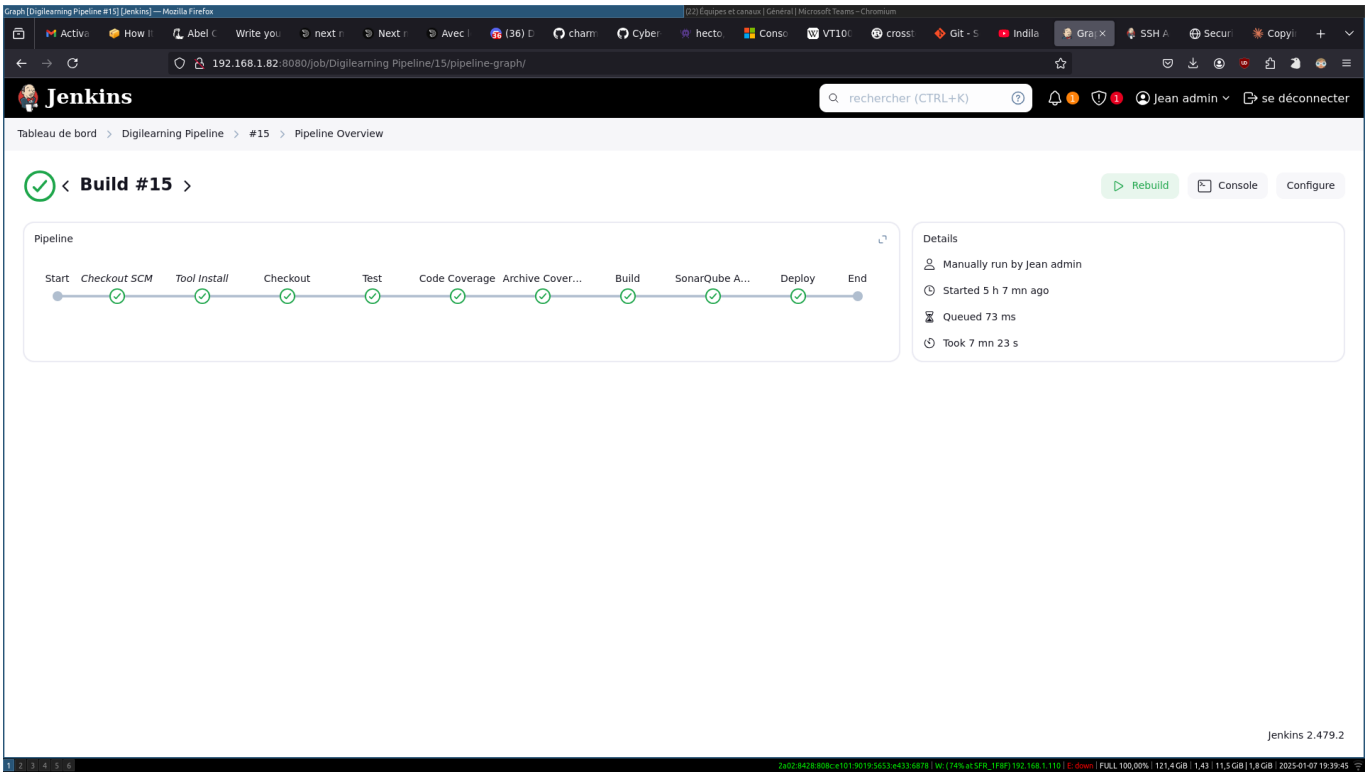
Voici le contenu du docker-compose.yml:

```
services:
    db:
        image: mariadb:latest
        restart: always
        environment:
            MARIADB_ROOT_PASSWORD: root
        ports:
            - 3306:3306
        volumes:
            - ./init-db:/docker-entrypoint-initdb.d

    phpmyadmin:
        image: phpmyadmin
        restart: always
        ports:
            - 8083:80
        depends_on:
            - db

    app:
        image: digilearning
        ports:
            - 8090:8090
        environment:
            SPRING_DATASOURCE_URL: jdbc:mariadb://db:3306/sid
            SPRING_DATASOURCE_USERNAME: root
            SPRING_DATASOURCE_PASSWORD: root
        volumes:
            - ./ressources:/digi-learning/ressources
```

On peut ainsi lancer le groupe de docker (base de données, phpmyadmin, application) permettant de déployer l'application.



4.2 Avec Git

L'idée était la suivante: on ne va pas build l'artifact sur jenkins et l'envoyer au serveur, on va simplement envoyer le code source au serveur, et build le code une une fois sur le serveur.

Concrètement, cela n'a pas grand intérêt de procéder de cette façon, mais j'avais envie de jouer avec les hooks de git.

Le serveur de déploiement doit être un Linux possédant git et ayant configuré git pour en faire un serveur.

Pour cela, on commence, par générer un couple de clés privée / publique.

Ensuite, sur le serveur , il faut générer un utilisateur nommé git, ajouter la clé publique que l'on vient de générer dans le fichier "authorized_keys" de son dossier .ssh, et ajouter le clé privée dans Jenkins. Pour cela, on utilise le même plugin SSH Agent.

Une fois cela fait, il faut configurer un hook pour lancer un script de déploiement chaque fois qu'un push est reçu sur la branche main.

L'idée est donc de créer un DockerFile et un docker-compose.yml mettant en place le stack nécessaire au déploiement de l'application, comme ceci :

```
#
# Build
#
FROM maven:3-eclipse-temurin-17-alpine AS build
RUN mkdir -p /workspace
WORKDIR /workspace
COPY pom.xml /workspace
COPY src /workspace/src
RUN mvn -B -f pom.xml clean package -DskipTests
```

```
#
# Package
#
FROM azul/zulu-openjdk-alpine:17-latest
WORKDIR /digi-learning
COPY --from=build /workspace/target/*.jar app.jar
EXPOSE 8090
ENTRYPOINT ["java","-jar","app.jar"]
```

Pour cela, on va créer un fichier bash exécutable au chemin suivant `/home/hijokaidan/PC/Digilearning/.git/hooks/post-receive`

```
#!/bin/bash

GIT_DIR="/home/hijokaidan/PC/digi-learning.git"
WORK_DIR="/home/hijokaidan/PC/digi-learning"
BRANCH="master"

while read oldrev newrev ref
do
    # only checking out the master (or whatever branch you would like to
deploy)
    if [[ $ref = refs/heads/$BRANCH ]];
    then
        echo "Ref $ref received. Deploying ${BRANCH} branch to
production..."
        cd "$WORK_DIR" || { echo "Failed to navigate to $WORK_DIR"; exit 1;
}
        docker compose down
        docker stop digilearning || true
        docker rm digilearning || true
        docker build -t digilearning .
        docker compose up
    else
        echo "Ref $ref received. Doing nothing: only the ${BRANCH} branch
may be deployed on this server."
    fi
done
```

Ce script s'active sur chaque push reçu, et effectue les actions suivantes :

- Vérification du fait qu'il s'agisse d'un push sur la branche master
- Stoppe les éléments lancés par le docker compose précédent
- Supprime la version précédente du build de l'application
- Rebuild l'application selon le docker file.
- Relance le stack selon le docker-compose

Et permet donc de lancer la nouvelle application en production

Pour l'instant, il n'est pas fonctionnel

5. Configuration du déploiement continu (non terminé).

Nous sommes actuellement en possession d'un VPS à l'adresse abel-ciccoli.fr / leeveen.com

Pour déployer en continu sur ce serveur, il faudrait:

- Installer jenkins sur le serveur via docker.
- Refaire toute la configuration
- Configurer le reverse proxy pour que jenkins.leeveen.com redirige vers Jenkins
- Configurer le reverse proxy pour que learning.leeveen.com redirige vers le docker du nom de digilearning.
- Configurer des github actions sur le repo pour envoyer le job Jenkins

L'idée est d'utiliser les github actions pour trigger les jobs Jenkins à chaque fois que l'on push sur main.

6. Conclusion

Il manque encore un bon nombre de choses pour rendre la pipeline viable.

Par exemple, il faudrait pouvoir variabiliser les URLs, permettre de déclencher le build en mode dev ou production, et avoir un serveur de test qui n'implique pas de mettre une application directement accessible par internet.

Toutefois, cela reste une base qui pourra être utilisé dans un contexte de mise en production d'une application, sans toutefois en être un exemple en terme de bonnes pratiques.