École Nationale des Sciences Appliquées de Marrakech

Rapport du Mini-Projet CI/CD – Voting CI Demo

Étudiants : RAMI Anas, RAZI Mouad, TAHOUN Brahim Khalil
Année universitaire : 2025–2026
Enseignant : *Pr. BOUARIFI*

Marrakech, 12 décembre 2025

Table des matières

1. Introduction

Le développement logiciel contemporain repose sur des itérations courtes où la qualité et la rapidité de livraison sont des impératifs simultanés. L'intégration continue et le déploiement continu (CI/CD) automatisent la construction, les tests, l'analyse de qualité et la livraison, réduisant drastiquement les risques d'erreur humaine. Ce mini-projet a pour objectif de bâtir une chaîne CI/CD complète autour de l'application *Voting CI Demo* afin d'illustrer la valeur de l'automatisation. Les outils mobilisés sont Java, Maven, JUnit pour les tests, JaCoCo pour la couverture, SonarQube pour l'analyse statique et Jenkins pour l'orchestration.

2. Présentation de l'application Voting CI Demo

2.1. Description fonctionnelle

Voting CI Demo est une application Java en ligne de commande permettant de gérer un vote simple : enregistrement de candidats, collecte des suffrages, application de différentes stratégies de comptage et remise à zéro du scrutin. L'objectif pédagogique est de disposer d'un socle clair pour démontrer la valeur des pratiques CI/CD.

2.2. Interface CLI et commandes principales

L'interface CLI offre les commandes suivantes : ajout de candidats, dépôt d'un vote, choix de stratégie de comptage, affichage du résultat et réinitialisation complète. Les interactions détaillées sont illustrées au chapitre 9.

2.3. Organisation en packages

— **model** : entités `Candidate`, `Vote` et structures de résultat.

— **repo** : repositories en mémoire pour les candidats et les votes.

— **service** : logique métier orchestrant enregistrement, validation et comptage.

— **observer** : écouteurs notifiés lors des événements de vote ou de réinitialisation.

— **strategy** : implémentations interchangeables de stratégies de comptage.

— **factory** : création centralisée des repositories et des stratégies.

2.4. Stratégies de vote

— **Plurality** : le candidat obtenant le plus grand nombre de voix l'emporte, sans exigence de majorité absolue.

— **Majority** : la victoire n'est acquise que si un candidat dépasse 50% des voix, sinon aucun vainqueur n'est proclamé.

3. Architecture du projet

3.1. Organisation interne

Le projet applique une séparation nette des responsabilités : le modèle structure les données, les services encapsulent la logique métier, les repositories assurent la persistance en mémoire, les stratégies rendent le comptage extensible, les observateurs gèrent les effets secondaires (journalisation) et la CLI constitue le point d'entrée utilisateur.

3.2. Description textuelle du diagramme UML

La classe `VoteService` orchestre le flux principal et dépend des repositories `VoteRepository` et `CandidateRepository`. Elle agrège une `CountingStrategy` injectée (`PluralityCountingStrategy` ou `MajorityCountingStrategy`). Les événements émis par `VoteService` sont observés par `LoggingVoteListener`. Les relations couvrent des dépendances de service vers les repositories, une composition de stratégie dans le service et une relation d'observation entre le service et les listeners.

3.3. Design patterns utilisés

- **Factory** : centralise l'instanciation des repositories et des stratégies, facilitant le remplacement d'implémentation.
- **Strategy** : permet de changer le mode de comptage sans modifier le cœur métier.
- **Observer** : notifie des listeners (ex. logs) lors des opérations de vote ou de remise à zéro.

3.4. Rôle des couches

- **Modèle** : description des entités métier et de leurs attributs.
- **Service** : logique de validation, d'enregistrement et de calcul des résultats.
- **Repository** : persistance en mémoire pour un accès rapide et testable.
- **CLI** : interface utilisateur textuelle qui orchestre les commandes et délègue au service.

4. Gestion du projet avec Maven

4.1. Structure du `pom.xml`

Le `pom.xml` définit l'identité du projet, la version de Java, les dépendances (JUnit, SLF4J, etc.) et les plugins qui automatisent compilation, tests, couverture et analyse.

4.2. Plugins principaux

— **maven-surefire-plugin** : exécute les tests unitaires JUnit.

— **jacoco-maven-plugin** : génère le rapport de couverture.

— **exec-maven-plugin** : lance l'application CLI depuis Maven.

— **sonar-maven-plugin** : envoie l'analyse vers SonarQube.

4.3. Commandes Maven essentielles

— `mvn clean test` : compilation et exécution des tests.

— `mvn jacoco:report` : génération du rapport JaCoCo.

— `mvn sonar:sonar` : analyse statique SonarQube.

— `mvn exec:java -Dexec.mainClass="cli.Main"` : lancement de la CLI.

5. Tests unitaires (JUnit)

5.1. Jeu de tests

Le jeu de tests couvre le service, les stratégies et les observateurs :

— **VoteServiceTest** : vérifie l'ajout des candidats, l'enregistrement des votes et le calcul des résultats.

— **VoteServiceResetTest** : contrôle la remise à zéro des données de vote.

— **RepositoryFactoryTest** : valide la construction des repositories par la factory.

— **MajorityCountingStrategyTest** : teste les scénarios de majorité absolue et l'absence de vainqueur.

— **LoggingVoteListenerTest** : assure la bonne émission et capture des événements.

5.2. Tableau récapitulatif

Suite de tests	Nombre de tests	Succès	Échecs
VoteServiceTest	6	6	0
VoteServiceResetTest	3	3	0
RepositoryFactoryTest	2	2	0
MajorityCountingStrategyTest	4	4	0
LoggingVoteListenerTest	2	2	0
Total	**17**	**17**	**0**

TABLE 5.1 – Synthèse des tests unitaires JUnit

5.3. Rôle dans la pipeline CI/CD

Les tests automatisés protègent contre les régressions à chaque commit et fournissent des preuves objectives de qualité. Ils conditionnent l'étape de couverture (JaCoCo) et la validation SonarQube pour autoriser ou bloquer la livraison.

6. Analyse de couverture (JaCoCo)

6.1. Principe de la couverture

La couverture mesure la proportion d'instructions et de branches exécutées par les tests. Elle permet d'identifier les zones non testées et d'orienter l'effort de validation.

6.2. Génération via Maven

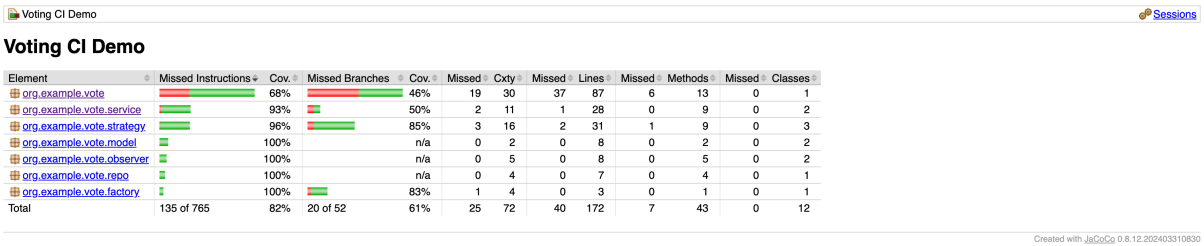
La commande `mvn jacoco:report` produit un rapport HTML dans `target/site/jacoco`, incluant cartes thermiques et détails par classe.

6.3. Résultats obtenus

— Couverture des instructions : 82%

— Couverture des branches : 61%

— Nombre de classes testées : 12


Voting CI Demo

Sessions

Voting CI Demo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.example.vote		68%		46%	19	30	37	87	6	13	0	1
org.example.vote.service		93%		50%	2	11	1	28	0	9	0	2
org.example.vote.strategy		96%		85%	3	16	2	31	1	9	0	3
org.example.vote.model		100%		n/a	0	2	0	8	0	2	0	2
org.example.vote.observer		100%		n/a	0	5	0	8	0	5	0	2
org.example.vote.repo		100%		n/a	0	4	0	7	0	4	0	1
org.example.vote.factory		100%		83%	1	4	0	3	0	1	0	1
Total	135 of 765	82%	20 of 52	61%	25	72	40	172	7	43	0	12

Created with JaCoCo 0.8.12.202403310830

FIGURE 6.1 – Rapport JaCoCo (placeholder)

7. Analyse statique avec SonarQube

7.1. Rôle de SonarQube

SonarQube réalise une analyse statique continue, détecte les bugs et vulnérabilités et impose un *Quality Gate* pour sécuriser la livraison. Son intégration dans la pipeline Jenkins permet d'arrêter automatiquement le processus en cas d'échec.

7.2. Lecture du tableau de bord

— Quality Gate : Passed

— Bugs critiques : 0

— Vulnérabilités : 0

— Maintenabilité : A

— Fiabilité : C (1 issue mineure)

— Couverture : 73.2%

— Duplications : 4.7%

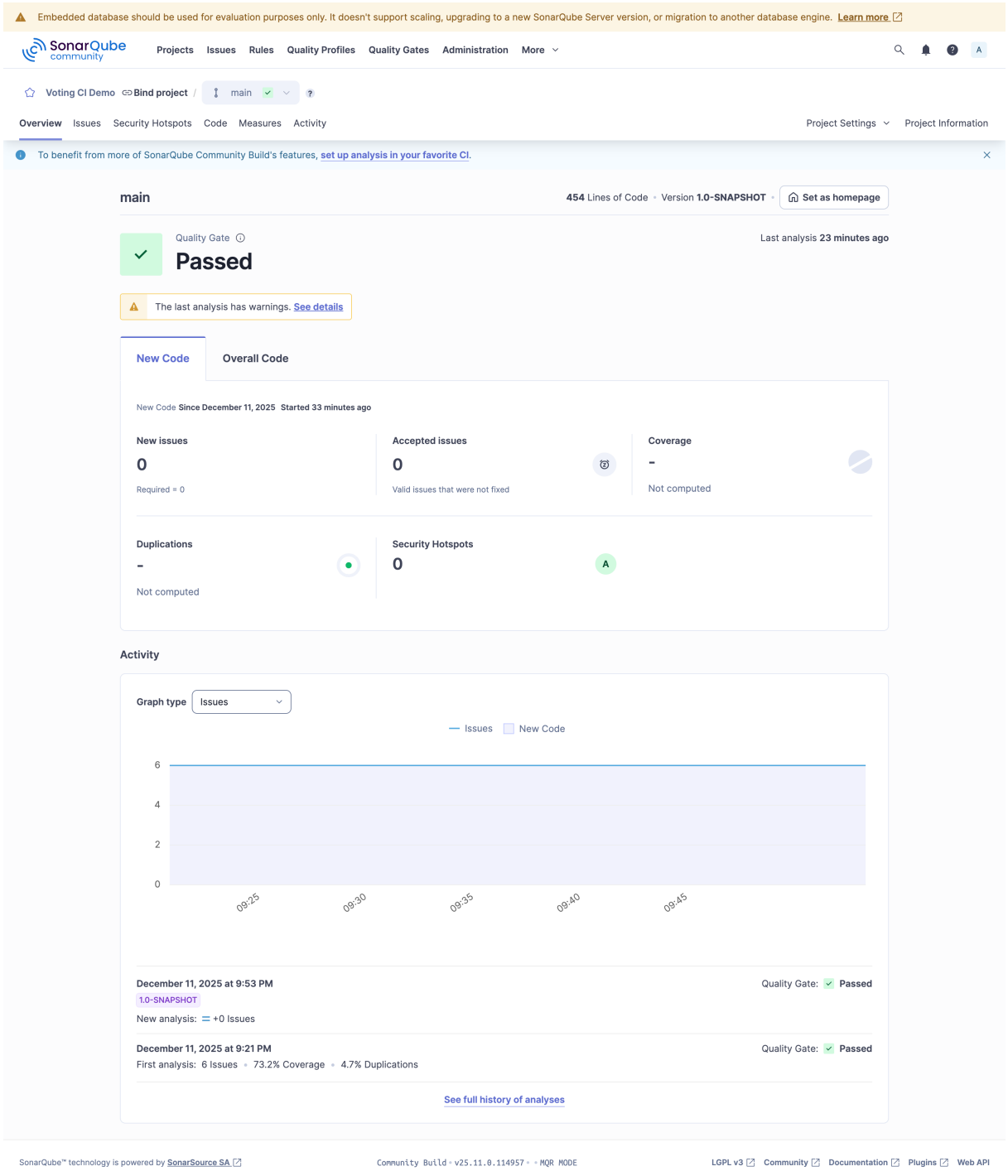


FIGURE 7.1 – Dashboard SonarQube (placeholder)

8. Pipeline CI/CD avec Jenkins

8.1. Étapes du Jenkinsfile

— **Checkout** : récupération du code source.

— **Build Maven** : compilation et résolution des dépendances.

— **Tests unitaires** : exécution via Surefire et publication des rapports JUnit.

— **Rapport JaCoCo** : génération du rapport et publication HTML.

— **Analyse SonarQube** : exécution de `sonar:sonar`.

— **Quality Gate** : attente du verdict SonarQube, blocage en cas d'échec.

— **Archivage des artefacts** : conservation des `jar` et rapports.

8.2. Utilisation des credentials

Les secrets `SONAR_HOST` et `SONAR_TOKEN` sont injectés comme variables d'environnement sécurisées pour l'étape SonarQube, évitant la divulgation des paramètres sensibles dans le `Jenkinsfile`.

8.3. Jenkinsfile

```
pipeline {
    agent any

    tools {
        maven 'Maven3'
        jdk 'JDK21'
    }

    environment {
        SONAR_TOKEN = credentials('SONAR_TOKEN')
    }
```

```
stages {

    /* -----------------------------------------------------
       BUILD MAVEN
    ----------------------------------------------------- */
    stage('Build') {
        steps {
            sh 'mvn -B clean package -DskipTests'
        }
        post {
            success {
                archiveArtifacts artifacts: 'target/*.jar', fingerprint: true
            }
        }
    }

    /* -----------------------------------------------------
       UNIT TESTS
    ----------------------------------------------------- */
    stage('Unit Tests') {
        steps {
            sh 'mvn -B test'
        }
        post {
            always {
                junit '**/target/surefire-reports/*.xml'
            }
        }
    }

    /* -----------------------------------------------------
       JACOCO COVERAGE
    ----------------------------------------------------- */
    stage('Code Coverage') {
        steps {
            sh 'mvn -B jacoco:report'
        }
        post {
            success {
                publishHTML(target: [
```

```
                reportName: 'JaCoCo Coverage',
                reportDir: 'target/site/jacoco',
                reportFiles: 'index.html',
                keepAll: true
            ])
        }
    }
}

/* ----------------------------------------------------
   SONARQUBE ANALYSIS
---------------------------------------------------- */
stage('SonarQube Analysis') {
    steps {
        withSonarQubeEnv('SonarQube') {
            sh """
                mvn sonar:sonar \
                  -Dsonar.projectKey=voting-ci-demo \
                  -Dsonar.token=$SONAR_TOKEN
            """
        }
    }
}

/* ----------------------------------------------------
   QUALITY GATE BLOCKER
---------------------------------------------------- */
stage('Quality Gate') {
    steps {
        timeout(time: 3, unit: 'MINUTES') {
            waitForQualityGate abortPipeline: true
        }
    }
}

stage('Deliver') {
    steps {
        echo "Delivery step (optional)"
    }
}
```

```
    }

    post {
        always {
            echo "Cleaning workspace..."
            cleanWs()
        }
    }
}
```

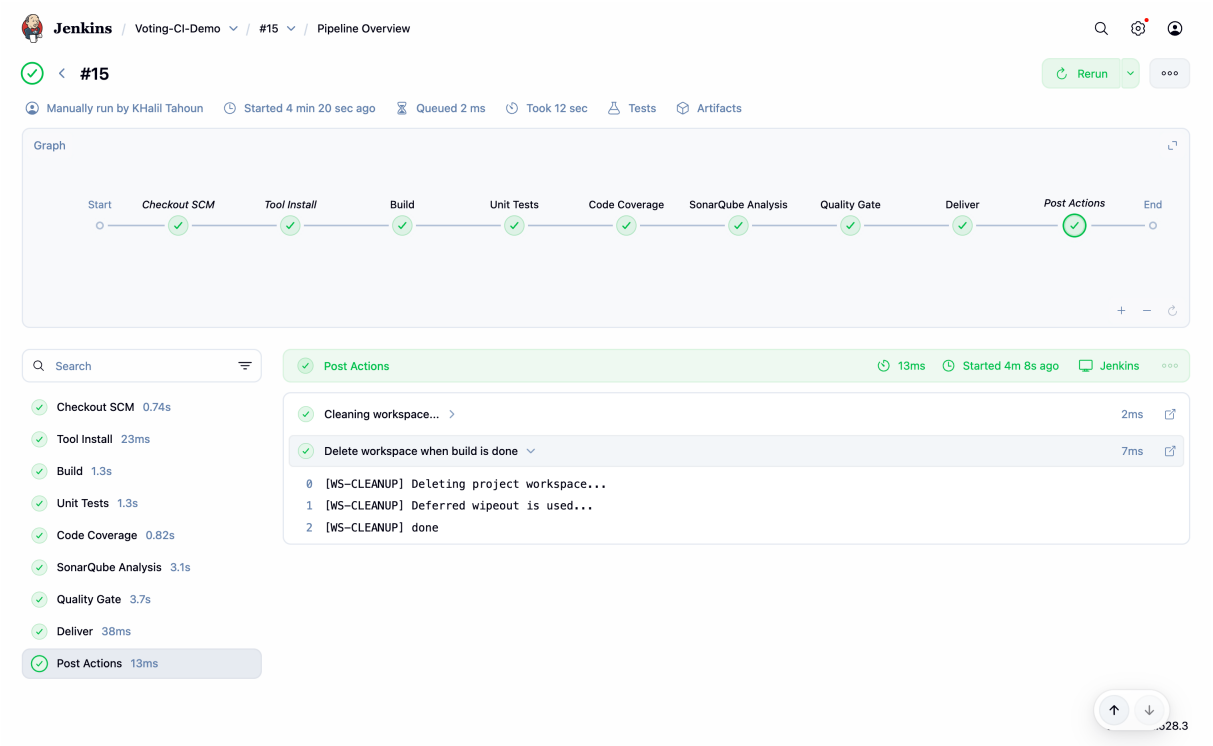


FIGURE 8.1 – Pipeline Jenkins

9. Démonstration d'exécution

9.1. Extraits de session CLI

```
# Ajout de candidats
add-candidate "Alice"
add-candidate "Bob"

# Votes enregistrés
vote "Alice"
vote "Bob"
vote "Alice"

# Comptage avec pluralité
count --strategy plurality
# Résultat: Alice (2), Bob (1)

# Comptage avec majorité
count --strategy majority
# Résultat: Alice obtient 66.7% (>50%), vainqueur.

# Réinitialisation
reset
```

10. Conclusion

Ce travail a permis de démontrer la valeur d'une chaîne CI/CD complète : compilation automatisée, tests systématiques, mesure de couverture, analyse statique et contrôle qualité par Jenkins. La pipeline renforce la fiabilité du code et réduit le temps de livraison en apportant des indicateurs objectifs à chaque étape. Les étudiants ont consolidé leurs compétences en conception (patterns Factory, Strategy, Observer), en écosystème Java/Maven, en tests unitaires, en couverture JaCoCo et en intégration SonarQube/Jenkins, compétences essentielles pour l'industrie logicielle moderne.