



LOG8371

Ingénierie de la qualité logicielle

Travail Pratique #1

Équipe: Sherbrooke

2002127 – Chebbi, Aymen
1621855 – Gafsi, Ahmed
2089012 – Nde Tsakou, Homer
2091172 – Potvin, David

Remis à : Patrick Loic Foalem

Hiver 2024

Table de matières

Régime d'Assurance Qualité.....	3
Introduction	3
Critères de qualité	4
Stratégie de Validation.....	5
Cas de tests	5
Rapports des tests.....	7
Complétude	7
Exactitude.....	8
Adéquation.....	9
Recensement des méthodes.....	9
Tolérance aux pannes.....	9
Récupérabilité	9
Maturité.....	10
Testabilité	11
Réutilisabilité.....	11
Possibilité de modification.....	11
Conclusion	12
Intégration continue	13
Installation de Jenkins:	13
Nouvelle fonction.....	21
Mise à jour du plan de qualité.....	26
Vidéo démontrant l'intégration et le déploiement en continu	26

Régime d'Assurance Qualité

Introduction

Aujourd'hui, il n'est pas rare que plusieurs développeurs travaillent ensemble pour développer un logiciel et pour ce faire, de nombreux outils sont utilisés pour faciliter la gestion du code source. Deux de ces outils sont Gerrit et Stash. Bien que l'utilisation de ces logiciels permette aux développeurs de créer plus facilement des logiciels, il arrive que du code de mauvaise qualité ou du code contenant des défauts soit introduit sans le vouloir dans le logiciel. Dans ce contexte, le logiciel Sputnik est d'une grande utilité, car c'est un outil permettant d'effectuer une révision statique du code qui provient d'un ensemble de changements proposés pour Gerrit ou Stash. En effet, le logiciel analyse le code source à l'aide de divers outils d'analyse statique du code tel que des linters pour divers langages de programmation ou encore d'autres outils qui permettent de détecter des vulnérabilités et des bogues dans le code. Sputnik a été conçu pour être exécuté après la phase « build » du serveur d'intégration continue (IC) de Jenkins. Ce faisant, la revue du code est effectuée de manière automatisée.

Le logiciel Sputnik est utilisé pour améliorer la qualité du code et réduire la possibilité d'intrusion de défauts dans le code source, alors l'importance de la qualité du projet Sputnik est primordiale, car le logiciel doit pouvoir détecter les problèmes lorsqu'il y en a. Ainsi, il faut que le nombre de faux négatifs soit minimisé. De plus, le fait que l'outil peut être utilisé pour une grande variété de projets allant du développement de jeux vidéo aux développements d'applications aéronautiques où la qualité est cruciale renforce d'autant plus l'importance de la qualité.

Les parties prenantes du logiciel Sputnik sont les développeurs du logiciel (contributeurs du projet qui est sur GitHub), les développeurs qui utilisent Gerrit ou Stash en conjonction avec Sputnik, les ingénieurs en assurance qualité et le gestionnaire de projet.

Les 5 critères de qualité utilisés sont la fonctionnalité, la fiabilité, la maintenabilité, l'efficacité et la sécurité.

La **fonctionnalité** est un critère de qualité qui décrit la capacité d'un système à satisfaire aux exigences spécifiées par les clients et aux exigences des utilisateurs du système.

La **fiabilité** est un critère de qualité qui décrit la capacité d'un système à fonctionner de manière stable et prévisible, à éviter les fautes et les pannes, à être disponible pour les utilisateurs ou les autres systèmes qui l'utilisent.

La **maintenabilité** est un critère de qualité qui décrit la facilité et la capacité d'un système à être réparé, mis à jour ou amélioré.

L'**efficacité** est un critère de qualité qui décrit la capacité d'un système à exercer ses fonctions de manière à utiliser les ressources disponibles de façon optimale.

La **sécurité** est un critère de qualité qui décrit la capacité d'un système à prévenir et à se protéger des accès non autorisés et à maintenir l'intégrité, la disponibilité et la confidentialité des données.

Critères de qualité

Tableau 1 : Objectif et méthode de validation des sous-critères de qualité

Critère	Sous critère	Objectifs	Mesures et méthode de validation
Fonctionnalité	Complétude	Le système doit fournir tous les résultats nécessaires	Le taux d'exécution des tests unitaires doit être de 95% (Utiliser le TDD, si le test est ignoré, la fonctionnalité du test n'est pas implémentée)
	Exactitude	Le système doit fournir des résultats exacts	Avoir un taux d'erreurs maximum de 10%
	Adéquation	Couverture de l'implémentation fonctionnelle = 1 - (Nombre de fonctions manquantes ou non exécutables / Nombre de fonctions spécifiées)	Ratio entre le nombre de fonctionnalités manquantes ou non exécutables et le nombre de fonctions spécifiées doit être inférieur à 10%
Fiabilité	Tolérance aux pannes	Éviter les crashes du logiciel à la suite d'un bogue.	Avoir un ratio inférieur à 0.10 entre le nombre de bogues qui font crasher le système et le nombre de bogues découverts
	Récupérabilité	Le système doit pouvoir être récupéré de façon rapide et efficace en cas de défaillance.	Le temps de récupération doit être inférieur à une heure de temps
	Maturité	Avoir la majorité des bogues corrigés.	Avoir un ratio minimum de 0.75 entre le nombre de défaillances corrigées et le nombre de défaillances détectées
Critère	Sous critère	Objectifs	Mesures et méthode de validation
Maintenabilité	Testabilité	Avoir des tests unitaires pour la majorité du code et chaque fonctionnalité requise.	Avoir un ratio d'un minimum de 0.75 entre le nombre de méthodes testées et le nombre de méthodes du code
	Réutilisabilité	Pouvoir réutiliser des composantes du code ailleurs	Avoir un ratio minimum de 0.3 entre le nombre de méthodes utilisées plus d'une fois et le

			nombre de méthodes totales
	Possibilité de modification	Le système doit pouvoir être modifié de manière facile et efficace	Le système doit être modifié sans perturber son fonctionnement (le nombre de bogues qui apparaît à la suite d'une modification inférieure à 1)

Stratégie de Validation

Afin de valider les objectifs sur les critères de qualité définis dans le plan de qualité et s'assurer du bon fonctionnement du logiciel dans des cas d'utilisations normales et d'autres cas extrêmes possibles, il est important de tester le logiciel de façon rigoureuse avec des tests pertinents. C'est pour quoi l'élaboration d'une stratégie de tests / validation est une nécessité.

Cas de tests

Le tableau ci-dessous représente donc les différents cas de tests pour les fonctionnalités clefs de sputnik lié à chacun des sous-critères de qualité définis plus haut.

Tableau 2 : Information sur les cas de tests

Objectif	Fonctionnalités concernées	Description	Test ou diagnostic	Logiciel/utilitaire mis à profit
Complétude	Ensemble des fonctionnalités	Faire du développement dirigé par les tests où tout test unitaire implémenté correspond à une fonctionnalité requise (TDD). Un test unitaire ignoré est un indicateur d'une fonctionnalité non implémentée.	Tests unitaires	IntelliJ
Exactitude	Ensemble des fonctionnalités	Exécuter les tests unitaires. Si un test unitaire d'une fonctionnalité implémentée échoue, cela indique une fonctionnalité mal implémentée.	Tests unitaires	IntelliJ
Adéquation	ShellCheck	Faire des tests unitaires et	Tests	IntelliJ

	Processor	s'assurer que le ratio de tests unitaires ignorés pour ShellCheck avec le nombre de méthodes doit être inférieur à 10%	unitaires	
Tolérance aux pannes	Configuration, Connector et Processor	L'équipe chargée du test identifie les scénarios de chaos potentiels et met en place une architecture pour simuler ces différents scénarios. Cela peut inclure l'utilisation d'outils de test de charge, de générateurs de faux trafic, de logiciels de simulation de pannes, etc.	Ingénierie du Chaos	Chaos Monkey
Récupérabilité	Connector Github	Forcer un échec de connexion à Github avec des « mocks » et des « spies » et s'assurer que cela ne cause pas une perte de service d'une durée supérieure à 60 secondes.	Test de fonction	Github CLI
Maturité	Ensemble des fonctionnalités	Vérification manuelle pour s'assurer qu'au moins 75% des issues sur Github sont résolues.	Issue Tracking	Github
Testabilité	Configuration	Compter le nombre de fonctions de la classe Configuration Builder et le nombre de tests unitaires associés.	Tests unitaires	IntelliJ
Réutilisabilité	Connector	Tester manuellement que les classes dépendantes de requêtes HTTP utilisent les méthodes des classes HTTP connector, helper et Exception pour au moins 30% de leurs méthodes.	Recensement de fonctions utilisées	—
Possibilité de modification	Connector Configuration	Utilisation de tests de régression et de fuzzing pour s'assurer que la modification de certaines fonctions ou certains attributs ne causent pas de bogues supplémentaires.	Régression et fuzzing	CIFuzz

Rapports des tests

Pour l'ensemble des tests réalisés à l'aide des tests unitaires, les résultats sont obtenus en exécutant les tests unitaires à l'aide de l'IDE IntelliJ qui fournit un résumé de l'exécution des tests avec des statistiques sur le nombre de tests lancés, ignorés, réussis et échoués.

Complétude

Test Summary

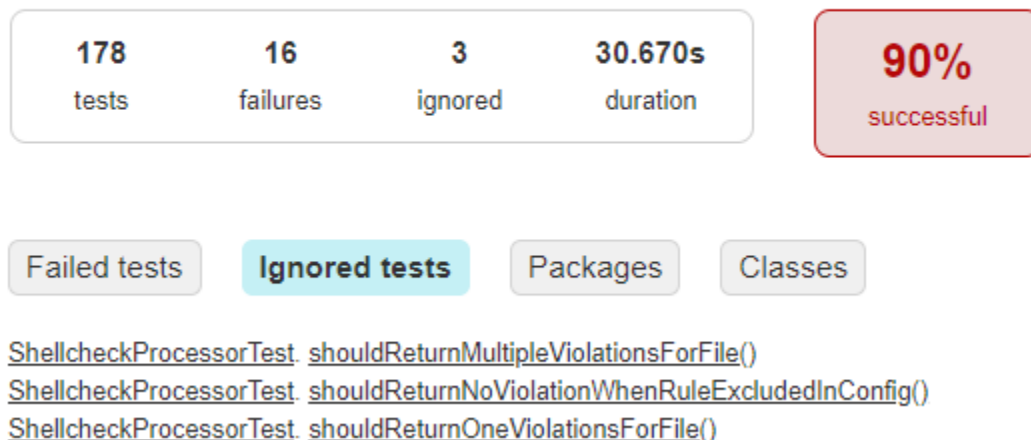


Figure 1. Résumé des tests

Nous remarquons trois tests unitaires ignorés sur 178 tests, donc un taux d'exécution de 98.3%. Le critère de complétude est validé.

Nous souhaitons souligner que malgré que le critère soit validé, une attention particulière devrait être fournie pour les tests ignorés, car ces derniers pourraient cacher une ou plusieurs vulnérabilités.

Ainsi notre recommandation est de placer une priorité élevée sur le rétablissement de ces tests.

Exactitude

Test Summary

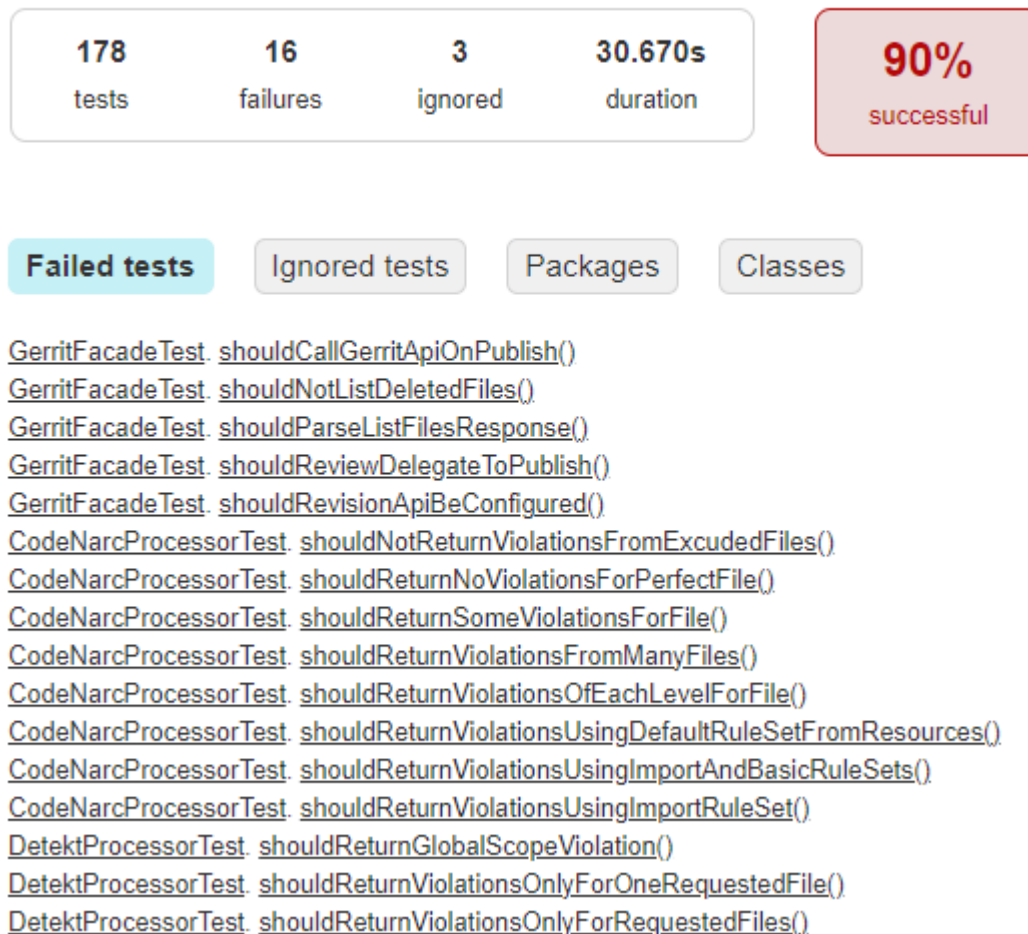


Figure 2. Résumé des tests

Sur 175 tests exécutés, nous obtenons 16 échecs pour un taux de réussite de 90.08% et donc un taux d'erreurs plus bas que 10%. Le critère d'exactitude est validé.

Notre recommandation est d'effectuer une analyse approfondie des 16 tests qui échouent pour les classer en fonction de leur criticité pour s'assurer qu'aucune vulnérabilité grave n'est présente.

Aussi, nous tenons à souligner que le taux de réussite dépasse tout juste le seuil minimal imposé. Or ce seuil est un minimum indicatif, et non un idéal. Nous recommandons donc aussi de chercher à atteindre un taux de réussite de 100%.

Adéquation

Test Summary

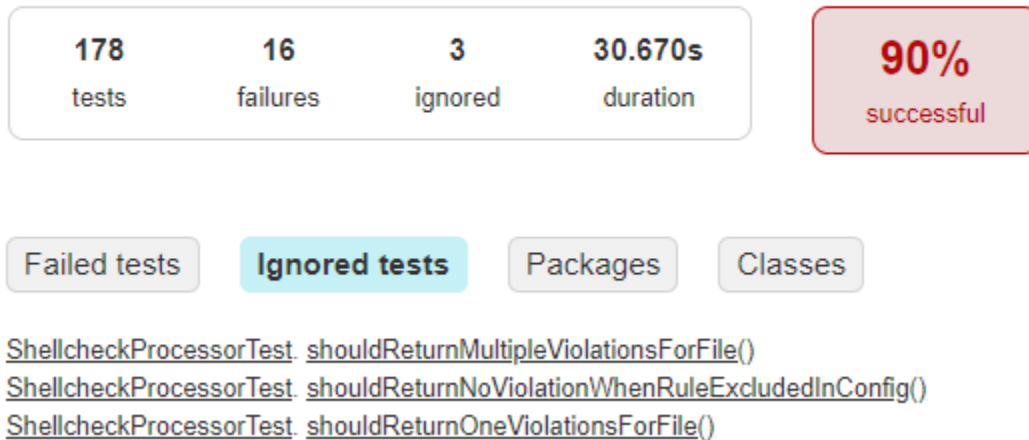


Figure 3. Résumé des tests

Recensement des méthodes

En parcourant minutieusement la classe ShellCheckProcessour, nous avons observé qu'elle implémente six méthodes en comptant le constructeur. Il y a donc $3/6 = 50\%$ de fonctionnalités manquantes. Le critère d'adéquation n'est pas validé.

Cela fait écho aux observations que nous avons réalisées pour le critère de complétude.

Il est important de faire en sorte que les tests de cette classe ne soient plus ignorés, sous peine de s'exposer à des vulnérabilités ou défauts sévères.

Tolérance aux pannes

Le système actuel ne permet pas l'exécution de ce test. Nous souhaitons justement insister sur l'importance de mettre en place des tests de l'ingénierie du chaos pour réellement mettre à l'épreuve la tolérance aux pannes du système en production. Nous recommandons l'usage d'un logiciel tel que Chaos Monkey pour réaliser cela, en prenant soin de choisir l'outil qui serait le plus facile à implémenter (il ne faut pas que l'outil de test cause des transformations radicales au système).

Récupérabilité

Nous recommandons d'implémenter une série de tests fonctionnels à l'aide de « mocks » et de « spies » pour simuler une erreur de connexion à GitHub et monitorer le temps de récupération du système.

Maturité

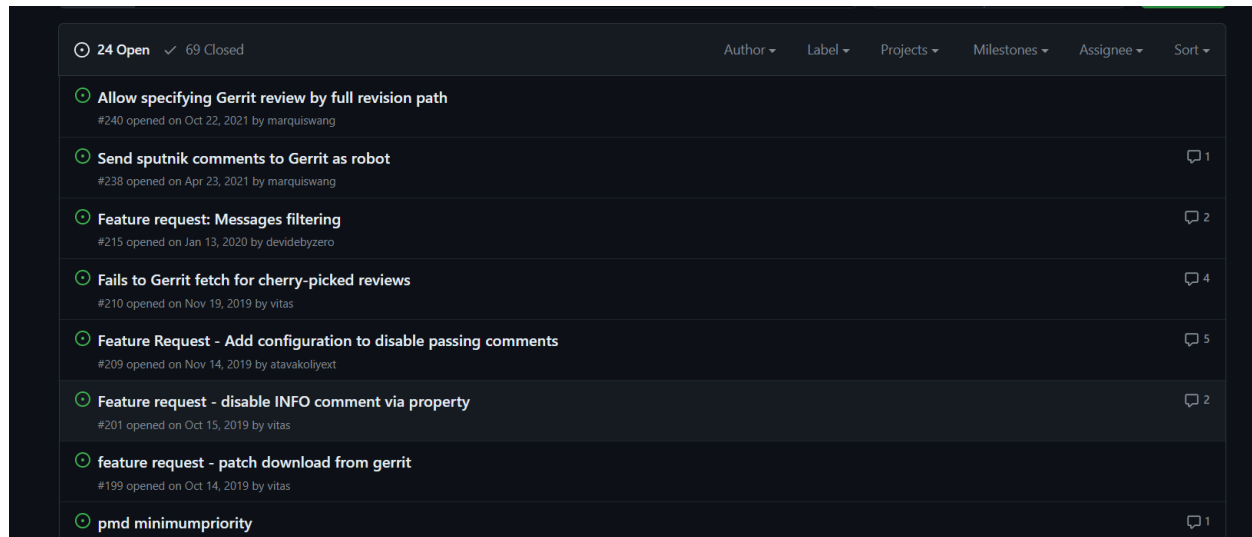


Figure 4. Issues ouvertes

La page GitHub du projet offre les statistiques suivantes sur les issues ouvertes pour le projet>

Tableau 3 : Résultat du test de maturité

Issues totales	Issues ouvertes	Issues fermées	Taux d'issues ouvertes
93	24	69	25.8%

Nous obtenons donc un taux d'issues ouvertes supérieur au seuil maximal imposé. Ainsi, le critère de maturité n'est pas satisfait.

Nous tenons cependant à fortement nuancer ce résultat.

Premièrement, des issues ouvertes, seul un est classifié en tant que bogue. De plus, plusieurs des issues ouvertes sont des demandes de nouvelles fonctionnalités.

Notre recommandation est donc de suspendre les issues ouvertes qui sont des demandes de nouvelles fonctionnalités non critiques pour concentrer les efforts de développement sur les fonctionnalités clefs et la correction des bogues.

Testabilité

Nous avons compté trois fonctions dans la classe ConfigurationBuilder, chacune permettant d'initialiser la classe à partir d'une source différente : depuis une ressource, un fichier ou des propriétés.

Nous avons dénombré 7 tests unitaires pour cette classe, dont au moins un test par méthode. Ainsi, nous jugeons que le critère de testabilité est satisfaisant.

Notre seule recommandation est de maintenir cette base de test à jour avec l'évolution de la classe.

Réutilisabilité

Nous avons réalisé une analyse approfondie des classes utilisant des méthodes HTTP qui révèle que les méthodes des classes HTTP sont toujours utilisées quand les connecteurs ne peuvent pas avoir recours à des SDK propres à la plateforme cible telle que pour Saas et Stash et représentent au moins 30% des appels et méthodes des classes les exploitant.

Nous concluons que les méthodes HTTP sont versatiles et atteignent le critère de réutilisabilité.

Possibilité de modification

Tel que pour l'ingénierie du chaos, le système actuel ne permet pas de facilement implémenter ce test de façon déterministe et consistante.

Notre recommandation est de mettre en place des tests unitaires avec du Fuzzing pour observer comment l'introduction de modification minimale impacte le système.

Conclusion

Tableau 4: Résultats des cas de tests et contre-mesure

Objectifs/tests associés	Résultats des cas de tests correspondants	Contre-mesure
Complétude (Tests unitaires)	Succès	—
Exactitude (Tests unitaires)	Succès	—
Adéquation (Tests unitaires)	Succès	—
Tolérance aux pannes (Ingénierie du Chaos)	Non implémenté	Identifier les scénarios de chaos potentiels et mettre en place une architecture pour simuler ces différents scénarios.
Récupérabilité (Test de fonction)	Connector Github	Implémenter une série de tests fonctionnels à l'aide de « mocks » et de « spies » pour simuler une erreur de connexion à GitHub et monitorer le temps de récupération du système.
Maturité (Issue Tracking)	Échec	Suspendre les issues ouvertes qui sont des demandes de nouvelles fonctionnalités non critiques pour concentrer les efforts de développement sur les fonctionnalités clefs et la correction des bogues.
Testabilité (Tests unitaires)	Succès	—
Réutilisabilité (Recensement de fonctions utilisées)	Succès	—
Possibilité de modification (Régression et fuzzing)	Non Implémenté	Implémenter des tests de fuzzing avec cifuzz et de régression.

Intégration continue

Notre choix s'est porté sur l'outil d'intégration Jenkins en raison de sa réputation d'excellence dans le domaine de l'intégration continue. Jenkins est reconnu pour sa fiabilité et sa capacité à automatiser l'ensemble du processus de développement logiciel de manière efficace. Avec Jenkins, nous bénéficions d'une plateforme robuste et hautement adaptable, offrant une grande flexibilité pour personnaliser notre processus d'automatisation des tests. Dans notre cas, nous avons mis en place un pipeline d'automatisation qui se déclenche automatiquement à chaque push effectué sur la branche master, assurant ainsi une intégration continue fluide et fiable.

Installation de Jenkins:

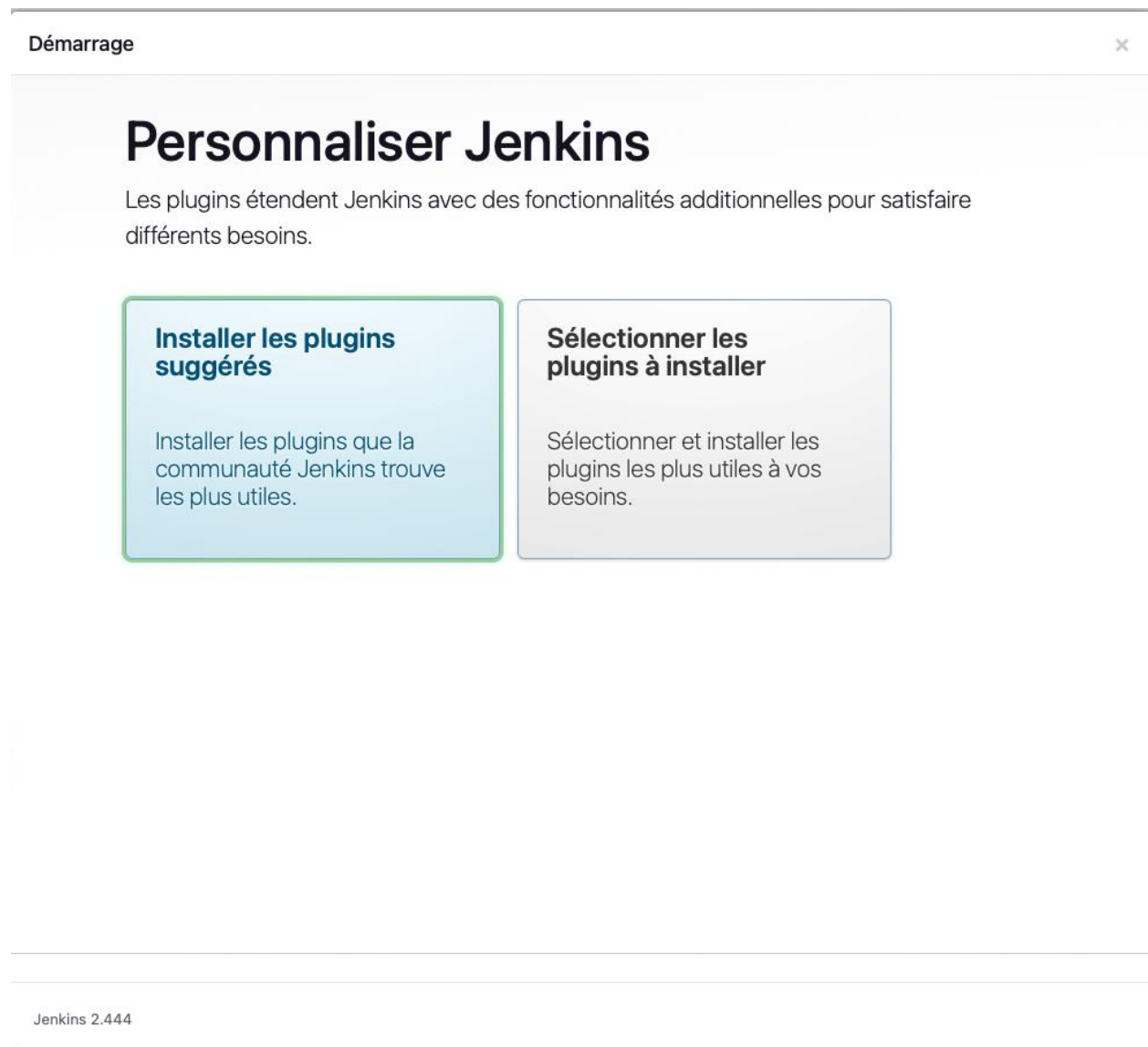


Figure 5. Installation de Jenkins 2.444

Installation en cours...

Installation en cours...

✓ Folders	✓ OWASP Markup Formatter	✓ Build Timeout	✓ Credentials Binding	Folders OWASP Markup Formatter ** JSON Path API ** Structs ** Pipeline: Step API ** Token Macro Build Timeout ** bouncycastle API ** Instance Identity ** JavaBeans Activation Framework (JAF) API ** JavaMail API ** Credentials ** Plain Credentials ** Gson API ** Trilead API ** SSH Credentials Credentials Binding ** SCM API ** Pipeline: API ** commons-lang3 v3.x Jenkins API Timestampers ** Caffeine API ** Script Security ** JAXB ** SnakeYAML API ** Jackson 2 API ** commons-text API ** Pipeline: Supporting APIs ** Plugin Utilities API ** Font Awesome API ** Bootstrap 5 API ** JQuery3 API ** ECharts API ** - dépendance requise
✓ Timestampers	Workspace Cleanup	Ant	Gradle	
Pipeline	GitHub Branch Source	Pipeline: GitHub Groovy Libraries	Pipeline: Stage View	
Git	SSH Build Agents	Matrix Authorization Strategy	PAM Authentication	
LDAP	Email Extension	Mailer	Dark Theme	
GitHub	xUnit	JUnit	HTML Publisher	
Coverage	NodeJS	MSBuild		

Jenkins 2.444

Figure 6. Installation des « plugins »

```
ahmedgafsi — zsh — zsh (figterm) › zsh — 80x24

[ahmedgafsi@Mac ~ % brew services
Name          Status User File
jenkins        none
mongodb-community none
[ahmedgafsi@Mac ~ % brew services start jenkins
==> Successfully started `jenkins` (label: homebrew.mxcl.jenkins)
[ahmedgafsi@Mac ~ % brew services
Name          Status User File
jenkins        started ahmedgafsi ~/Library/LaunchAgents/homebrew.mxcl.jenkins.plist
mongodb-community none
ahmedgafsi@Mac ~ %
```

Figure 7. Ouverture de Jenkins à l'aide de la commande *brew services*

Après avoir installé Jenkins, on va le configurer avec Java 8

Tableau de bord > Administrer Jenkins > Tools

Utiliser les réglages globaux, modifier par utilisateur

Installations JDK

Installations JDK ^ Edited

Ajouter JDK

≡ JDK

Nom

temurin-8.jdk

JAVA_HOME

/Library/Java/JavaVirtualMachines/temurin-8.jdk/Contents/Home

☒ Install automatically ?

Ajouter un installateur

Ajouter JDK

Git installations

≡ Git

Name

Enregistrer Appliquer

Figure 8. Configurations de Java 8

Par la suite, on va configurer ngrok.

```
ahmedgafsi@Mac ~ % ngrok http http://localhost:8080
```

Figure 9. Lancement de ngrok

```
ngrok
Build better APIs with ngrok. Early access: ngrok.com/early-access

Session Status      online
Account             Ahmed (Plan: Free)
Update              update available (version 3.6.0, Ctrl-U to update)
Version             3.5.0
Region              United States (us)
Latency              30ms
Web Interface        http://127.0.0.1:4040
Forwarding           https://11a0-132-204-251-253.ngrok-free.app -> http://localhost:8080

Connections          ttl    opn    rt1    rt5    p50    p90
                    5      0      0.01   0.00   30.28  30.39
```

Figure 10. Ouverture de ngrok

À partir du lien <https://11a0-132-204-251-253.ngrok-free.app> fournis par ngrok (section “Forwarding”), on va créer un webhook sur Github.

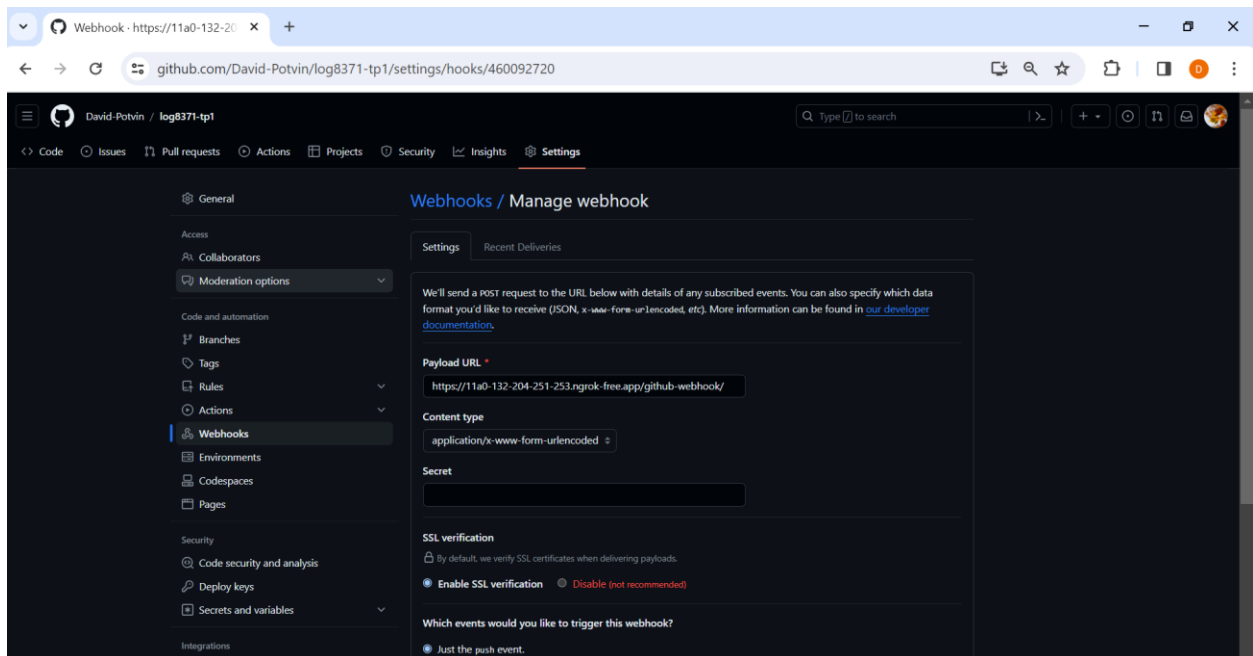


Figure 11. Ajout d'un webhook à notre répertoire Github


```
ahmedgafsi — zsh — zsh (figterm) • ngrok — 98x30
ngrok (Ctrl+C to quit)
Build better APIs with ngrok. Early access: ngrok.com/early-access

Session Status      online
Account             Ahmed (Plan: Free)
Update              update available (version 3.6.0, Ctrl-U to update)
Version             3.5.0
Region              United States (us)
Latency             30ms
Web Interface        http://127.0.0.1:4040
Forwarding           https://11a0-132-204-251-253.ngrok-free.app -> http://localhost:8080

Connections
  ttl   opn   rt1   rt5   p50   p90
    5    0    0.01  0.00  30.28  30.39

HTTP Requests
-----
POST /github-webhook/ 200 OK
```

Figure 12. Ngrok est désormais configuré avec le webhook de GitHub

On va désormais créer le projet Jenkins.

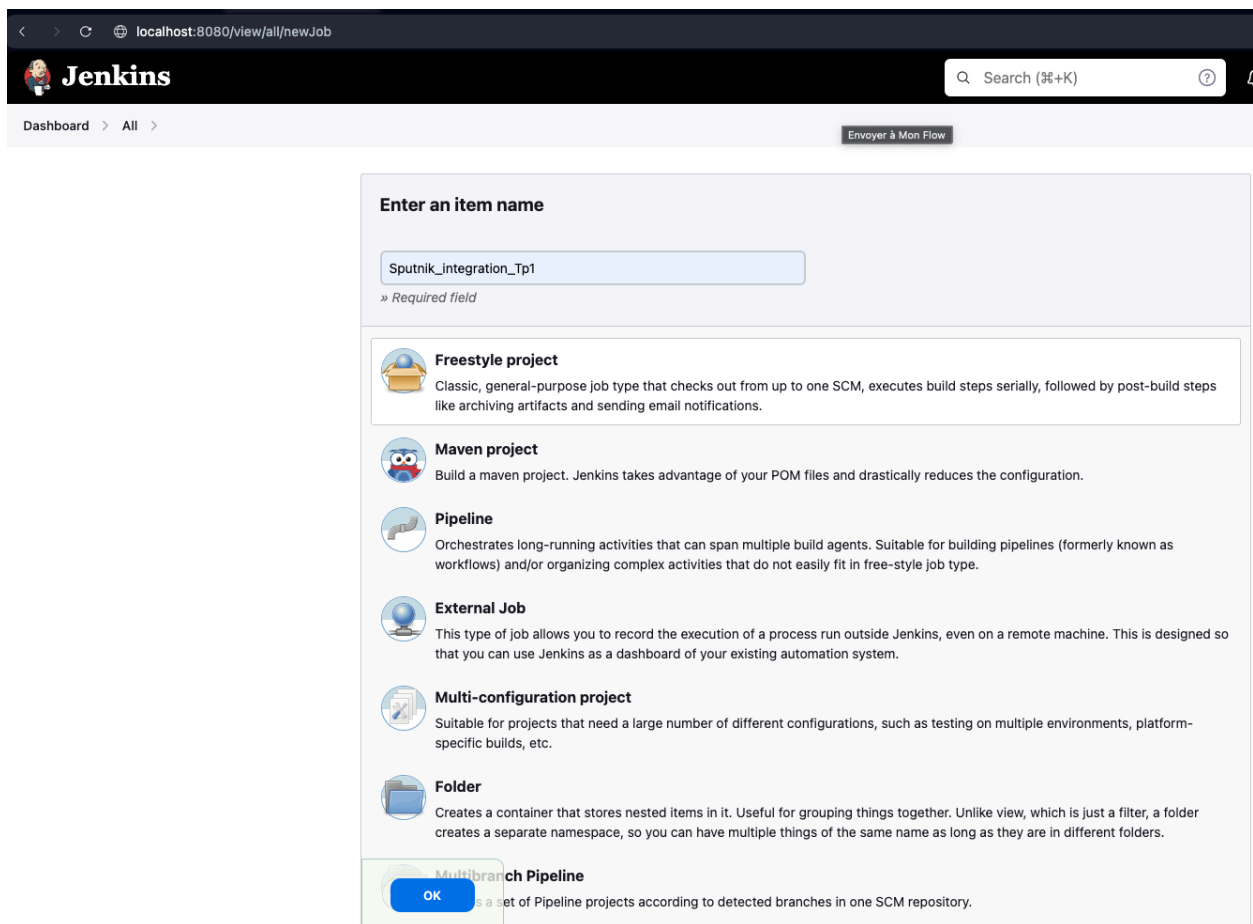


Figure 13. Création du projet Sputnik_integration_Tp1

The screenshot shows the Jenkins configuration interface for a project named 'Sputnik_integration_Tp1'. The 'General' tab is selected in the left sidebar. The main area contains the following fields and options:

- Description:** A text area containing 'Intégration continue du projet sputnik'.
- Plain text:** A link labeled 'Preview'.
- Jira site:** A dropdown menu.
- Options:** A list of checkboxes for various build settings:
 - ☐ Discard old builds ?
 - ☐ GitHub project
 - ☐ Disable Automated Maven Repository Cleanup
 - ☐ This project is parameterised ?
 - ☐ Throttle builds ?
 - ☐ Execute concurrent builds if necessary ?
- Advanced:** A dropdown menu.

At the bottom, there are 'Save' and 'Apply' buttons.

Figure 14. Configurations générales du projet

Après avoir créé le projet, on le configure pour qu'il fonctionne avec notre répertoire hébergé par GitHub.

The screenshot shows the Jenkins configuration interface for the same project, but with the 'Source Code Management' tab selected. The configuration is as follows:

- Source Code Management:** A radio button selection where 'Git' is chosen over 'None'.
- Repositories:** A dashed box containing:
 - Repository URL:** A text field with 'https://github.com/David-Potvin/log8371-tp1.git'.
 - Credentials:** A dropdown menu showing 'MDXZ-cyber/***** (Auth2)'.
 - + Add:** A button to add more repositories.
- Advanced:** A dropdown menu.
- Add Repository:** A button.
- Branches to build:** A dashed box containing:
 - Branch Specifier (blank for 'any'):** A text field with '*/master'.
- Add Branch:** A button.
- Repository browser:** A dropdown menu.

At the bottom, there are 'Save' and 'Apply' buttons.

Figure 15. Configurations Git du projet

Ensuite, on configure le projet pour notifier Jenkins lorsqu'il y a des changements dans l'entrepôt de code.

Build Triggers

- ☐ Trigger builds remotely (e.g., from scripts) ?
- ☐ Build after other projects are built ?
- ☐ Build periodically ?
- ☐ Enable Artifactory trigger
- ☐ GitHub Branches
- ☐ GitHub Pull Request Builder
- ☐ GitHub Pull Requests ?
- ☒ GitHub hook trigger for GITScm polling ?
- ☐ Maven Dependency Mise a jour Trigger ?
- ☐ Poll SCM ?

Figure 16. Configuration d'un trigger pour GitHub

On va désormais configurer Jenkins pour qu'il rentre dans le dossier sputnik (cd sputnik) et pour qu'il puisse nettoyer le « build » avant d'exécuter les tests à l'aide du wrapper de gradle et pour qu'il puisse générer un rapport de couverture de code à la fin (./gradlew clean test jacocoTestReport).

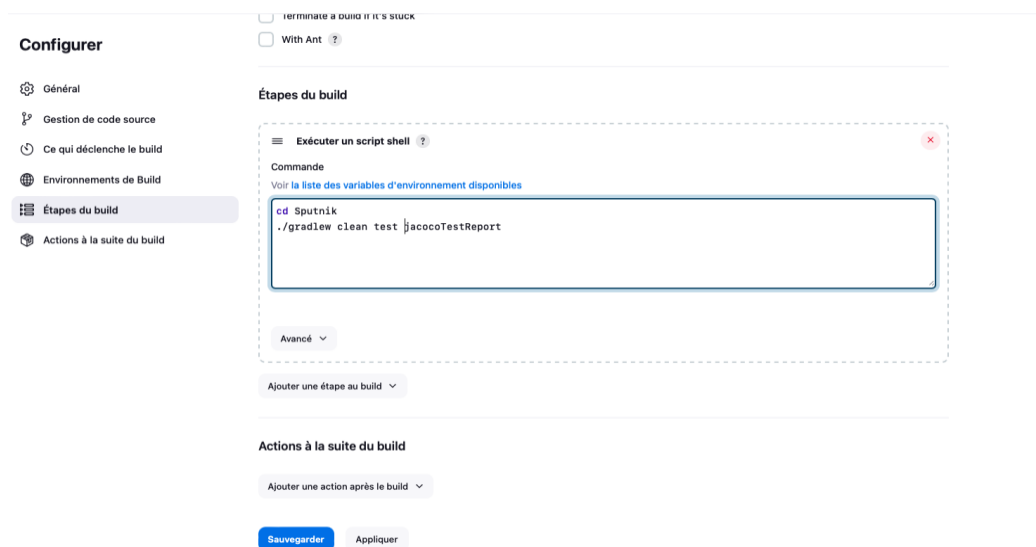


Figure 17. Étapes du « build »

```
178 tests completed, 3 failed, 3 skipped

> Task :test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at:
file:///Users/ahmedgafsi/.jenkins/workspace/Sputnik_Integration/sputnik/build/reports/tests/test/index.html

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output. Run with --scan to
get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 1m 22s
6 actionable tasks: 6 executed
Build step 'Exécuter un script shell' marked build as failure
Finished: FAILURE
```

Figure 18. Résultat de la sortie de la console Jenkins après avoir exécuté les tests

Test Summary

178 tests	3 failures	3 ignored	52.450s duration	98% successful
---------------------	----------------------	---------------------	----------------------------	--------------------------

Failed tests Ignored tests Packages Classes

CheckstyleProcessorTest. shouldConsiderSuppressionsWithConfigLocProperty()
CheckstyleProcessorTest. shouldReturnBasicSunViolationsOnSimpleClass()
SpotBugsProcessorTest. shouldReturnBasicViolationsOnEmptyClass()

Generated by Gradle 6.7.1 at 2024-02-10 17:11:31

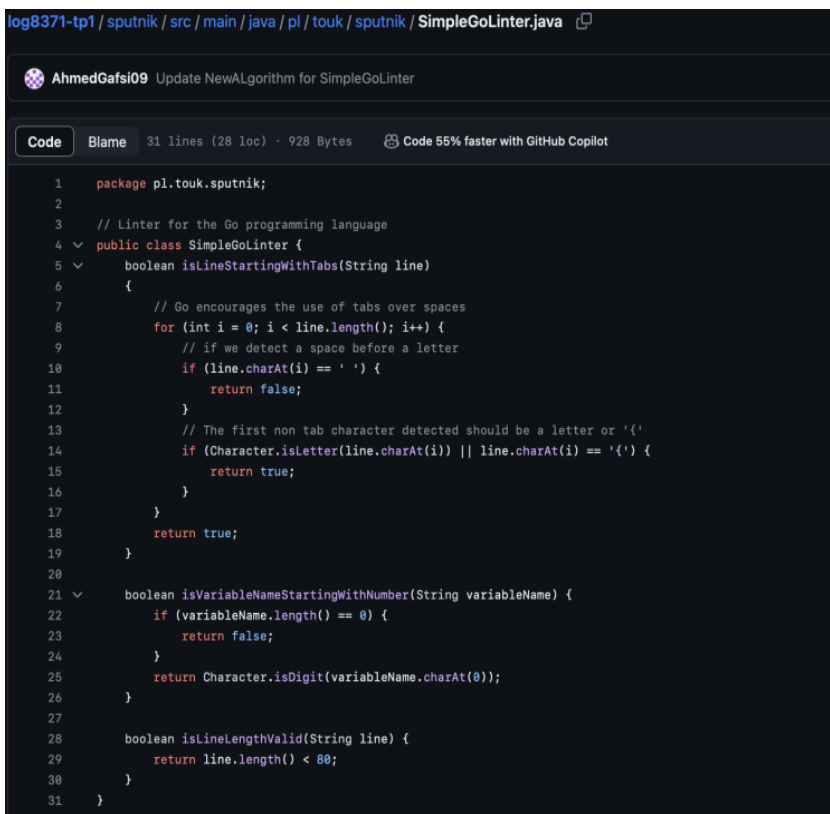
Figure 19. Résumé de l'exécution des tests fournis par Jacoco

Nouvelle fonction

La nouvelle fonctionnalité introduite à l'aide de la classe SimpleGoLinter propose une série de méthodes permettant de déterminer si du code écrit avec le langage de programmation Go respecte les normes de codage du langage. Par exemple, la première méthode s'assure que le code est indenté avec des 'tabs' au lieu des 'espaces', car c'est ce que recommande Go. La deuxième méthode permet de vérifier que le nom associé à une variable ne commence pas par un chiffre, car c'est interdit. La troisième méthode permet de vérifier si une ligne de code contient moins de 80 caractères, car cela permet d'améliorer la lisibilité. Chacune de ces méthodes est conçue pour vérifier une norme de bonne pratique spécifique du langage.

L'ajout de la nouvelle classe SimpleGoLinter impacte directement le régime d'assurance qualité du système Sputnik. En effet, avec l'introduction de nouvelles fonctionnalités, il est crucial de garantir que ces fonctionnalités fonctionnent correctement et ne perturbent pas le bon fonctionnement du système dans son ensemble. Ainsi, nous avons inclus des tests unitaires spécifiques pour chaque méthode de la classe SimpleGoLinter afin de vérifier leur fonctionnement correct et leur compatibilité avec le reste du système.

Ces tests unitaires ont été intégrés dans notre stratégie de validation, ce qui nécessite une mise à jour de notre régime d'assurance qualité pour inclure ces nouveaux tests et garantir la qualité globale du système.



```
log8371-tp1 / sputnik / src / main / java / pl / touk / sputnik / SimpleGoLinter.java
AhmedGafsi09 Update NewAlgorithm for SimpleGoLinter
Code Blame 31 lines (28 loc) · 928 Bytes Code 55% faster with GitHub Copilot

1 package pl.touk.sputnik;
2
3 // Linter for the Go programming language
4 public class SimpleGoLinter {
5     boolean isLineStartingWithTabs(String line)
6     {
7         // Go encourages the use of tabs over spaces
8         for (int i = 0; i < line.length(); i++) {
9             // if we detect a space before a letter
10            if (line.charAt(i) == ' ') {
11                return false;
12            }
13            // The first non tab character detected should be a letter or '{'
14            if (Character.isLetter(line.charAt(i)) || line.charAt(i) == '{') {
15                return true;
16            }
17        }
18        return true;
19    }
20
21    boolean isVariableNameStartingWithNumber(String variableName) {
22        if (variableName.length() == 0) {
23            return false;
24        }
25        return Character.isDigit(variableName.charAt(0));
26    }
27
28    boolean isLineLengthValid(String line) {
29        return line.length() < 80;
30    }
31 }
```

Figure 20. Code du nouveau Linter pour le langage Go

```
log8371-tp1 / sputnik / src / test / java / pl / touk / sputnik / SimpleGoLinterTest.java

AhmedGafsi09 Update NewAlgorithm for SimpleGoLinter

Code Blame 39 lines (34 loc) · 1.46 KB Code 55% faster with GitHub Copilot

1 package pl.touk.sputnik;
2 import org.junit.jupiter.api.Test;
3 import static org.junit.Assert.*;
4
5
6 public class SimpleGoLinterTest {
7     @Test
8     public void testIsLineStartingWithTabs() {
9         SimpleGoLinter linter = new SimpleGoLinter();
10        assertTrue(linter.isLineStartingWithTabs(""));
11        assertTrue(linter.isLineStartingWithTabs("{}");
12        assertTrue(linter.isLineStartingWithTabs("func main() {}));
13        assertTrue(linter.isLineStartingWithTabs("\tif number > 5 {}));
14        assertFalse(linter.isLineStartingWithTabs(" if number > 5 {}));
15    }
16    @Test
17    public void testIsVariableNameStartingWithNumber() {
18        SimpleGoLinter linter = new SimpleGoLinter();
19        assertTrue(linter.isVariableNameStartingWithNumber("iPhoneNumber"));
20        assertFalse(linter.isVariableNameStartingWithNumber(""));
21        assertFalse(linter.isVariableNameStartingWithNumber("phoneNumber"));
22    }
23
24    @Test
25    public void testIsLineLengthValid() {
26        StringBuilder stringBuilder = new StringBuilder();
27        for (int i = 0; i < 79; i++) {
28            stringBuilder.append('a');
29        }
30        String stringWith79Chars = stringBuilder.toString();
31
32        SimpleGoLinter linter = new SimpleGoLinter();
33        assertTrue(linter.isLineLengthValid(""));
34        assertTrue(linter.isLineLengthValid("func main() {}));
35        assertTrue(linter.isLineLengthValid(stringWith79Chars));
36        assertFalse(linter.isLineLengthValid(stringWith79Chars + "a"));
37    }
38
39 }
```

Figure 21. Tests unitaires associés au code du nouveau linter pour le langage Go

```

ahmedgafsi@Mac sputnik % git push
Énumération des objets: 28, fait.
Décompte des objets: 100% (28/28), fait.
Compression par delta en utilisant jusqu'à 10 fils d'exécution
Compression des objets: 100% (11/11), fait.
Écriture des objets: 100% (16/16), 2.10 Kio | 2.10 Mio/s, fait.
Total 16 (delta 3), réutilisés 0 (delta 0), réutilisés du pack 0
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/David-Potvin/log8371-tp1.git
9bb4c5e..1d6fd03 testBranch -> testBranch

```

Figure 22. Push du linter pour le langage Go

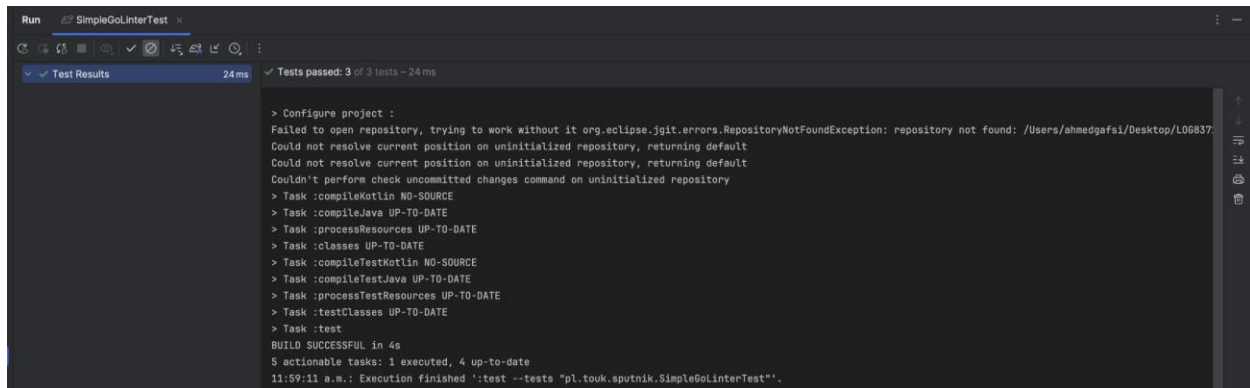


Figure 23. Résultat obtenu lors de l'exécution des tests associés à la nouvelle fonctionnalité

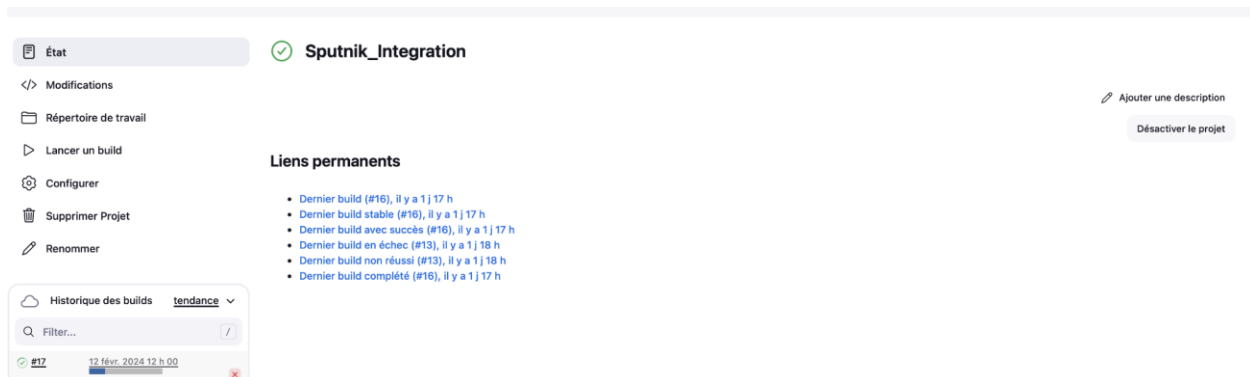


Figure 24. Lancement du « build »

```

[Sputnik_Integration] $ /bin/sh -xe /var/folders/n0/hfbrz2r50057d290cmswpfxxr0000gn/T/jenkins6092602090209160483.sh
+ cd Sputnik
+ ./gradlew clean test jacocoTestReport

> Configure project :
Failed to open repository, trying to work without it org.eclipse.jgit.errors.RepositoryNotFoundException: repository not found:
/Users/ahmedgafsi/.jenkins/workspace/Sputnik_Integration/sputnik
Could not resolve current position on uninitialized repository, returning default
Could not resolve current position on uninitialized repository, returning default
Couldn't perform check uncommitted changes command on uninitialized repository

> Task :clean
> Task :compileKotlin NO-SOURCE

> Task :compileJava
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

> Task :processResources
> Task :classes
> Task :compileTestKotlin NO-SOURCE
> Task :compileTestJava
> Task :processTestResources
> Task :testClasses
> Task :test
> Task :jacocoTestReport

BUILD SUCCESSFUL in 1m 0s
7 actionable tasks: 7 executed
Finished: SUCCESS

```

Figure 25. Sortie de la console Jenkins du « build »

Test Summary

181	0	0	40.433s	100% successful
tests	failures	ignored	duration	

Packages

Classes

Package	Tests	Failures	Ignored	Duration	Success rate
pl.touk.sputnik	9	0	0	0.029s	100%
pl.touk.sputnik.configuration	6	0	0	2.347s	100%
pl.touk.sputnik.connector	1	0	0	0.104s	100%
pl.touk.sputnik.connector.gerrit	18	0	0	0.693s	100%
pl.touk.sputnik.connector.github	2	0	0	0.529s	100%
pl.touk.sputnik.connector.http	6	0	0	0.702s	100%
pl.touk.sputnik.connector.local	4	0	0	0.761s	100%
pl.touk.sputnik.connector.saas	5	0	0	0.315s	100%
pl.touk.sputnik.connector.stash	5	0	0	2.715s	100%
pl.touk.sputnik.engine	24	0	0	0.206s	100%
pl.touk.sputnik.engine.visitor	8	0	0	0.038s	100%
pl.touk.sputnik.engine.visitor.comment	7	0	0	0.213s	100%
pl.touk.sputnik.engine.visitor.score	6	0	0	0.022s	100%
pl.touk.sputnik.processor.checkstyle	4	0	0	0.041s	100%
pl.touk.sputnik.processor.codenarc	13	0	0	5.979s	100%
pl.touk.sputnik.processor.detekt	6	0	0	3.503s	100%
pl.touk.sputnik.processor.eslint	1	0	0	0.040s	100%
pl.touk.sputnik.processor.jshint	5	0	0	2.418s	100%
pl.touk.sputnik.processor.jslint	5	0	0	2.233s	100%
pl.touk.sputnik.processor.ktlint	5	0	0	4.643s	100%
pl.touk.sputnik.processor.pmd	7	0	0	7.245s	100%
pl.touk.sputnik.processor.pylint	6	0	0	0.572s	100%
pl.touk.sputnik.processor.scalastyle	3	0	0	1.012s	100%
pl.touk.sputnik.processor.shellcheck	8	0	0	0.057s	100%
pl.touk.sputnik.processor.spotbugs	5	0	0	3.726s	100%
pl.touk.sputnik.processor.tslint	5	0	0	0.046s	100%
pl.touk.sputnik.review	4	0	0	0.237s	100%
pl.touk.sputnik.review.filter	1	0	0	0.005s	100%
pl.touk.sputnik.review.locator	2	0	0	0.002s	100%

Figure 26. Rapport de couverture de code le projet incluant la nouvelle fonctionnalité

SimpleGoLinterTest

all > pl.touk.sputnik > SimpleGoLinterTest

3	0	0	0.011s	100%
tests	failures	ignored	duration	successful

Tests

Test	Duration	Result
testIsLineLengthValid()	0s	passed
testIsLineStartingWithTabs()	0.006s	passed
testIsVariableNameStartingWithNumber()	0.005s	passed

Figure 27. Rapport de couverture de code les tests associés à la nouvelle fonctionnalité

Mise à jour du plan de qualité

Notre plan de qualité actuel couvre déjà un large éventail d'aspects concernant l'assurance qualité et les exigences du système Sputnik. Après avoir évalué l'impact de l'introduction de la nouvelle fonctionnalité fournie par la classe SimpleGoLinter, nous avons conclu que cette addition ne constitue pas un changement majeur dans le fonctionnement global du système. Par conséquent, nous estimons qu'il n'est pas nécessaire de mettre à jour notre plan de qualité logicielle. Les objectifs de qualité que nous avons définis précédemment restent pertinents et suffisamment complets pour couvrir cette nouvelle fonctionnalité sans nécessiter de modifications significatives. Bien que nous ayons ajouté des tests unitaires spécifiques pour la classe SimpleGoLinter afin de garantir son bon fonctionnement, cela ne nécessite pas de révision majeure de notre plan de qualité, car cela n'impacte pas de manière critique la fonctionnalité principale du système. Ainsi, notre stratégie d'assurance qualité demeure solide et alignée avec les objectifs globaux du projet.

Vidéo démontrant l'intégration et le déploiement en continu

https://youtu.be/O_GtBbzFusA