

Mini-Projet :

Qualité logicielle

Spécialité**: Génie informatique 3**

Thème**:**

**Analyse de la Qualité du Code et Validation Fonctionnelle d’une Application Logicielle (PFA)**

**Réalisé par : Sous la direction de :**

**TANAFAAT Abdessamad  
ROUCHDI Ilias Aimad QAZDAR**

Année Universitaire 2024-2025

Table des matières

[Table des matières 8](#_Toc167245265)

[Liste des figures 11](#_Toc167245266)

[Liste des tables 15](#_Toc167245267)

[Introduction générale 16](#_Toc167245268)

[Chapitre 1 17](#_Toc167245269)

[CONTEXTE DU PROJET 17](#_Toc167245270)

[1 Présentation du projet logiciel : 18](#_Toc167245271)

[2 Recensement des erreurs rencontrées: 20](#_Toc167245275)

[3 Methodes utilisées pour détecter les erreures 23](#_Toc167245281)

[4 Classification des erreurs 28](#_Toc167245285)

[Chapitre 2 31](#_Toc167245289)

[Analyse de la Qualité avec SonarQube 31](#_Toc167245290)

[1 Configuration de SonarQube 32](#_Toc167245291)

[2 Analyse du code source 33](#_Toc167245292)

[3 Rapport des résultats 33](#_Toc167245292)

[Chapitre 3 43](#_Toc167245298)

[TEST BOITE BLANCHE 43](#_Toc167245299)

[1 Développement des tests unitaires 44](#_Toc167245300)

[2 Itérations des tests : 44](#_Toc167245301)

[Chapitre 4 51](#_Toc167245308)

[TESTS BOITE NOIRE 51](#_Toc167245309)

[1 Utilisation de Selenium 52](#_Toc167245310)

[2 Tests fonctionnels 52](#_Toc167245311)

[Chapitre 5 74](#_Toc167245314)

[TEST DE PERFORMANCE 74](#_Toc167245315)

[1 Configuration de l'outil 75](#_Toc167245316)

[2 Tests de stress 75](#_Toc167245317)

[Chapitre 6 93](#_Toc167245320)

[INTEGRATION DEVOPS 93](#_Toc167245321)

[1 Mise en place de CI/CD 94](#_Toc167245322)

[2 Surveillance de la qualités 94](#_Toc167245323)

[3 Déploiement continu 95](#_Toc167245324)

**Chapitre 1**

**Contexte du Projet**

1. **Présentation du projet logiciel**

Le projet précédent visait à développer une plateforme en ligne dédiée à la gestion des enquêtes de satisfaction client. Elle comportait deux modules principaux :

**Module d'administration** : Permettait aux administrateurs de gérer les utilisateurs, les rôles, les privilèges, et de configurer les paramètres de la plateforme.

**Module pour les propriétaires d'entreprises** : Offrait des outils pour créer, personnaliser, analyser des enquêtes, et gérer des programmes de fidélité.

### Description du projet

Le projet précédent consistait en la conception et le développement d'une plateforme en ligne dédiée à la gestion des enquêtes de satisfaction client.

Notre groupe s'est concentré exclusivement sur le module d'administration de cette plateforme, qui joue un rôle clé dans la gestion et la configuration de l'application. Ce module permet de garantir une administration efficace des utilisateurs, des rôles, et des privilèges, tout en offrant une personnalisation avancée des paramètres de la plateforme.

**Fonctionnalités développées dans le module d'administration**

1. **Gestion des utilisateurs :**

Création, modification, activation/désactivation des comptes administrateurs.

Attribution de rôles spécifiques aux utilisateurs.

1. **Gestion des rôles et privilèges :**

Création de rôles personnalisés.

Attribution de privilèges spécifiques à chaque rôle.

Gestion des droits d'accès pour les différentes parties de la plateforme.

1. **Configuration et paramétrage :**

Ajout et modification des services par défaut pour les enquêtes.

Gestion des catégories d'entreprises et des zones de service.

Configuration des langues et des modèles d'enquêtes.

1. **Exportation de données :**

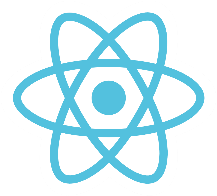
Génération et exportation des données de la plateforme sous différents formats (PDF, CSV).

**Technologies utilisées** :

* **Backend** : Spring Boot, Spring Security, Spring Data JPA.

* **Frontend** : React, Axios, Tailwind CSS.

* **Base de données** : MySQL.



* **Outils de gestion et tests** : Git, GitHub, Postman.

1. **Recensement des erreurs rencontrées dans le module**
   1. **Problèmes fonctionnels :**

**Situation 1 : Mauvaise gestion des rôles complexes**

Lors de l'attribution des privilèges, certaines erreurs dans la logique de validation ont permis de créer des rôles administratifs incohérents.

**Exemple :**

L'administrateur principal crée un nouveau rôle appelé "Superviseur Enquêtes". Ce rôle est censé permettre uniquement la consultation des résultats d’enquêtes. Cependant, lors de la configuration, le système permet d'attribuer à ce rôle à la fois :

**Privilège de consultation** : Permet de lire les enquêtes.

**Privilège de suppression** : Permet de supprimer les enquêtes.

Cela génère un conflit : un utilisateur avec ce rôle peut accidentellement supprimer des enquêtes au lieu de simplement les consulter.

**Conséquences :**

Un superviseur sans compétences techniques a supprimé plusieurs enquêtes importantes par erreur.

Les autres utilisateurs n'ont pas pu retrouver les données supprimées, ce qui a causé une perte de données critiques.

**Raison :**

L'absence de contrôle logique empêchant l'attribution simultanée de privilèges contradictoires.

**Solution proposée :**

Ajouter une validation dans le backend pour vérifier les combinaisons de privilèges et rejeter celles qui sont incohérentes. Par exemple :

Si un rôle a "lecture uniquement", désactiver automatiquement "suppression" ou "édition".

**Situation 2 : Mauvaise gestion des mots de passe réinitialisés pour les propriétaires :**

Lorsqu’un propriétaire dépasse le nombre de tentatives autorisées pour saisir son mot de passe, son compte est automatiquement désactivé pour des raisons de sécurité. Cependant, cela pose des défis lorsque ce propriétaire souhaite réactiver son compte. Le problème initial était de définir un processus fiable pour générer et communiquer un nouveau mot de passe afin de réactiver le compte.

**Problème rencontré :**

Lorsqu’un administrateur décide de réactiver le compte, il n’existe pas de processus automatisé pour générer un nouveau mot de passe.

Les administrateurs créent **manuellement** un mot de passe temporaire, mais cela peut poser des problèmes :

Certains administrateurs définissent des mots de passe faibles, mettant en danger la sécurité du compte.

**Solution proposée**

Pour résoudre ce problème, voici un processus automatisé et sécurisé pour gérer la réactivation des comptes :

Génération automatique de mots de passe sécurisés :

Lorsque l’administrateur réactive le compte, le système génère automatiquement un mot de passe temporaire sécurisé (par exemple : 12 caractères, avec des majuscules, chiffres, et caractères spéciaux).

Ce mot de passe est stocké temporairement sous forme chiffrée dans la base de données et lié à un délai d’expiration (par exemple, 24 heures).

Communication au propriétaire :

Le mot de passe temporaire est envoyé directement au propriétaire via un canal sécurisé (email ou SMS).

L’email contient des instructions pour se connecter et un lien pour changer le mot de passe immédiatement après la première connexion.

Validation des connexions :

Lors de la première connexion avec le mot de passe temporaire, le propriétaire est obligé de définir un nouveau mot de passe permanent qui respecte les règles de sécurité.

Audit et notifications :

Un journal des tentatives de connexion et de réactivation est maintenu pour permettre une surveillance par les administrateurs.

Les propriétaires reçoivent une notification confirmant que leur compte a été réactivé.

* 1. **Erreurs de conception :**

**Manque de modularité dans certaines parties du code**

**Problème 1 : Configuration de sécurité dispersée dans plusieurs fichiers :**

Dans une application, les règles de sécurité (comme les autorisations pour chaque rôle, les filtres de sécurité, ou la configuration d'authentification) sont directement définies dans différents contrôleurs ou services.

**Exemple de problème déjà rencontré :**

@PostMapping("/admin/addUser")

public ResponseEntity<?> addUser(@RequestBody User user) {

if (user.getRole().equals("ADMIN")) {

// Autoriser l'opération

} else {

throw new AccessDeniedException("Access Denied");

} }

@GetMapping("/user/viewProfile")

public ResponseEntity<?> viewProfile(Authentication authentication) {

if (authentication.getAuthorities().contains("ROLE\_USER")) {

// Autoriser l'accès

} else {

throw new AccessDeniedException("Access Denied");

} }

**Problèmes causés :**

Les règles de sécurité sont dispersées, ce qui complique leur modification ou ajout.

La sécurité dépend directement des contrôleurs, ce qui va à l’encontre du principe de séparation des préoccupations.

Les développeurs doivent chercher dans tout le projet pour modifier une règle de sécurité, augmentant le risque d’erreurs.

**Solution proposée :** Implémentation d'un **package security**

Créer un package dédié à la sécurité, **com.project.security.**

Centraliser toute la logique de sécurité dans des classes de configuration Spring Security.

@Configuration @EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/admin/\*\*").hasRole("ADMIN")

.antMatchers("/user/\*\*").hasRole("USER")

.anyRequest().authenticated()

.and()

.formLogin()

.and()

.logout(); }

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder(); } }

**Problème 2 : Couplage excessif entre la logique métier et les contrôleurs**

Ce problème se produit lorsque les contrôleurs (qui sont responsables de gérer les requêtes HTTP et d'envoyer des réponses) contiennent directement la logique métier de l'application, c’est-à-dire, les calculs, les règles de gestion, et les interactions avec les données. Cela rend le code difficile à maintenir, à tester, et à étendre.

**Problèmes causés par le couplage excessif :**

**Difficulté à tester :** Le code dans les contrôleurs est souvent difficile à tester de manière unitaire, car il inclut des dépendances directes à la logique métier, aux services, et parfois à la base de données.

**Mauvaise séparation des préoccupations :** Les contrôleurs ne devraient pas contenir de logique métier, mais uniquement des appels à des services qui encapsulent cette logique. Un code de contrôleur trop complexe va à l'encontre du principe de séparation des préoccupations.

**Maintenance difficile** : À mesure que l’application évolue, la logique métier dans les contrôleurs peut devenir obsolète, redondante, ou incohérente, nécessitant des modifications répétées dans de nombreux endroits.

**Couplage étroit :** Cela rend le système plus fragile, car toute modification dans la logique métier peut entraîner des erreurs ou des incohérences dans plusieurs contrôleurs.

**Exemple de problème déjà rencontré :**

@PostMapping("/login")

public ResponseEntity<?> login(@RequestBody LoginRequest loginRequest) {

User user = userRepository.findByEmail(loginRequest.getEmail());

if (user == null || !BCrypt.checkpw(loginRequest.getPassword(), user.getPassword())) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials"); }

return ResponseEntity.ok(new AuthResponse(token));

**Solution : Séparer la logique métier dans un service dédié :**

La solution consiste à extraire la logique métier dans des services métiers, permettant ainsi de rendre les contrôleurs légers et responsables uniquement de la gestion des requêtes et des réponses HTTP. La logique métier doit être isolée dans des services ou des composants dédiés qui gèrent l'interaction avec la base de données et l'exécution des règles métiers.

// Controlleur :

@PostMapping("/login")

public ResponseEntity<?> login(@RequestBody LoginRequest loginRequest) {

User user = userService.authenticateUser(loginRequest.getEmail(), loginRequest.getPassword());

if (user == null) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials");

}

public User authenticateUser(String email, String password) {

User user = userRepository.findByEmail(email);

if (user != null && BCrypt.checkpw(password, user.getPassword())) {

return user; }

return null; }

* 1. **Erreurs de configuration :**

**1. Problèmes de compatibilité des versions de dépendances :**

Spring Boot et Hibernate ou Jackson ne sont pas compatibles, vous pouvez rencontrer des erreurs comme **NoClassDefFoundError** ou des erreurs de **sérialisation JSON**. D’où :

La gestion des transactions peut échouer, ou Jackson peut ne pas être capable de sérialiser des objets.

**Soltuion :**

**Spring Boot Starter Parent** comme parent de votre projet. Cela fournit une configuration par défaut pour de nombreuses dépendances de Spring Boot et permet de gérer automatiquement la compatibilité des versions avec des versions recommandées de Spring Boot.

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.2.3</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

**2. Erreurs dans la configuration de sécurité (authentification et autorisation) :**

**Problème :** des erreurs dans la configuration de sécurité, notamment l'authentification et l'autorisation dans Spring Security.

Si les rôles ne sont pas correctement configurés dans Spring Security, par exemple si vous oubliez d'ajouter un rôle "ADMIN" ou de configurer un utilisateur avec ce rôle dans la base de données, l'utilisateur ne pourra pas accéder aux ressources protégées.

**Notre Solution :**

définie dans la classe SecurityConfig, est une configuration complète de sécurité pour une application Spring utilisant Spring Security. Elle inclut plusieurs aspects de la sécurité, y compris l'authentification par JWT, la gestion des accès via CORS (Cross-Origin Resource Sharing), et la configuration de la gestion des sessions pour garantir que l’application soit sans état (stateless).

**Sécurité renforcée** grâce à l'utilisation de JWT pour l'authentification sans état.

**Contrôle précis** des accès avec des filtres personnalisés.

**Prise en charge du CORS** pour gérer les requêtes provenant de différents domaines (utilisé pour les applications frontend séparées).

**Gestion des mots de passe** avec un hachage sécurisé (BCrypt).

**Pratique moderne** de gestion de la sécurité dans des architectures distribuées et évolutives (API REST).

Quelques lignes de codes de la classe Security Config :

AuthenticationFilter authenticationFilter = new AuthenticationFilter(customAuthenticationManager , adminRepository );

authenticationFilter.setFilterProcessesUrl(SecurityConstants.AUTHENTICATE\_PATH);

http.csrf(AbstractHttpConfigurer::disable)

.cors(AbstractHttpConfigurer::disable)

.authorizeHttpRequests(auth ->

auth

.requestMatchers(SecurityConstants.WHITE\_LIST)

.permitAll() // Accès libre pour certains chemins

.anyRequest()

.authenticated() // Requiert une authentification pour toutes les autres ressources

)

.addFilterBefore(new ExceptionHandlerFilter(), AuthenticationFilter.class)

.addFilter(authenticationFilter)

.addFilterAfter(new JwtAuthorizationFilter(blacklistRepository), AuthenticationFilter.class)

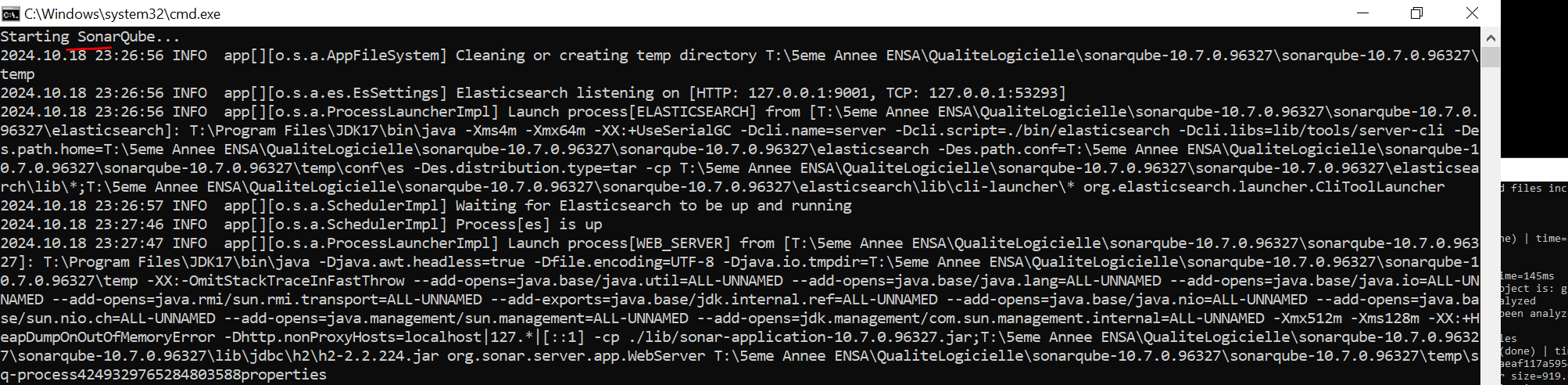
.sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS)); // Pas de session côté serveur

**Chapitre 2**

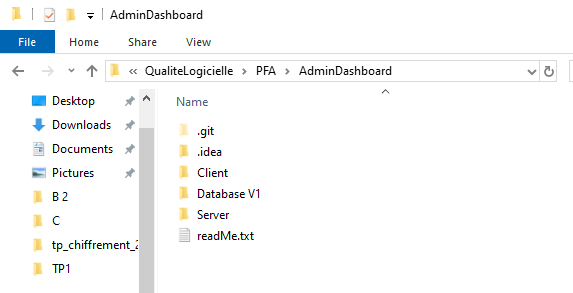
**Analyse de la Qualité avec SonarQube**

1. **Configuration de SonarQube :**
2. **Project Front-End (Client) : Java Script – CSS / frameworks : React – Tailwind CSS :**

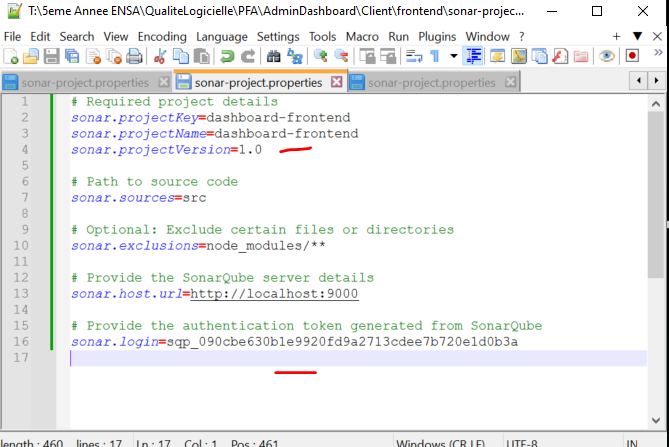
Accès à un serveur SonarQube :

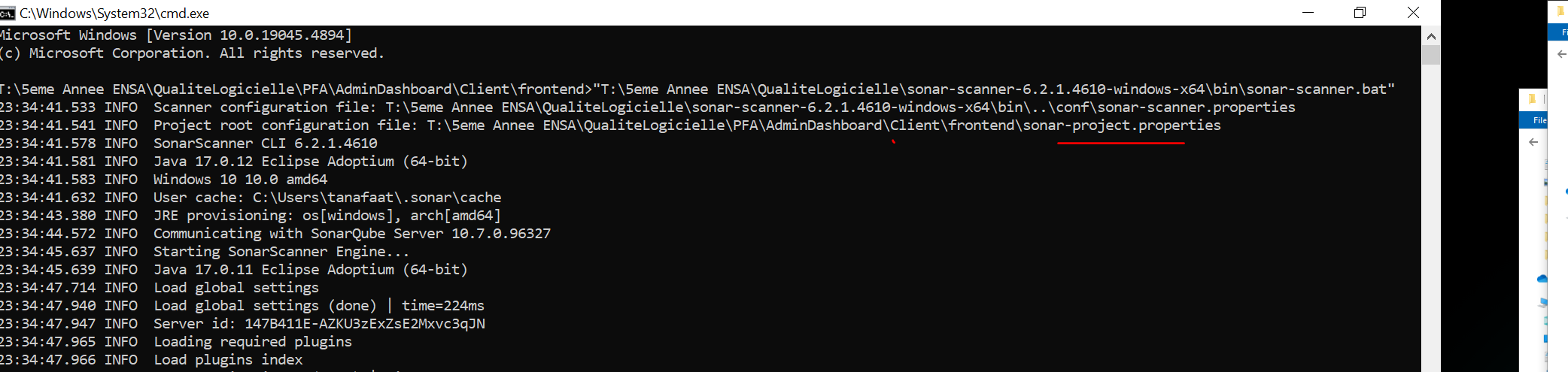


Identification du code source PFA :



Utilisation de SonarScanner pour analyser le code source :

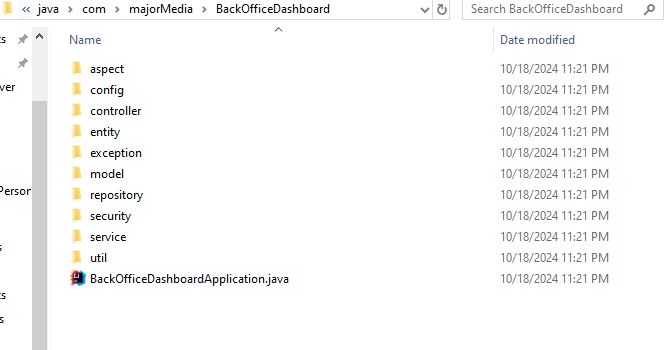




Puis dans le répertoire on fait la commande : *sonar-scanner*.

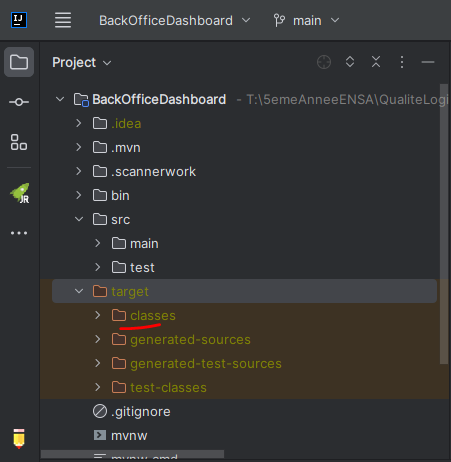
**B. Project Back-End (Server) : Java / Framework : Spring boot – Spring Security :**

Identification du code source PFA :

****

Génération de .class files : (Setting > Build Project) :

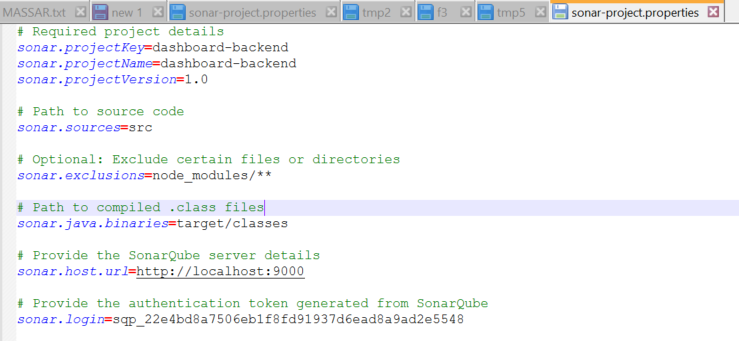
Avoir le dossier target/classes



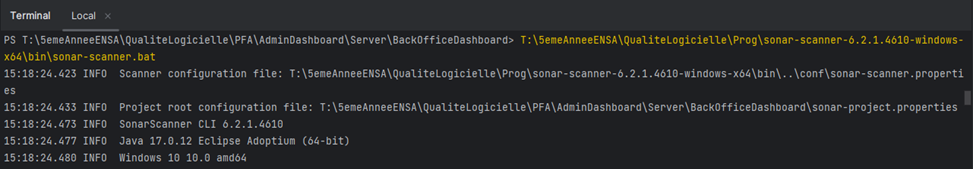
Configure SonarQube Analysis in IntelliJ IDEA :

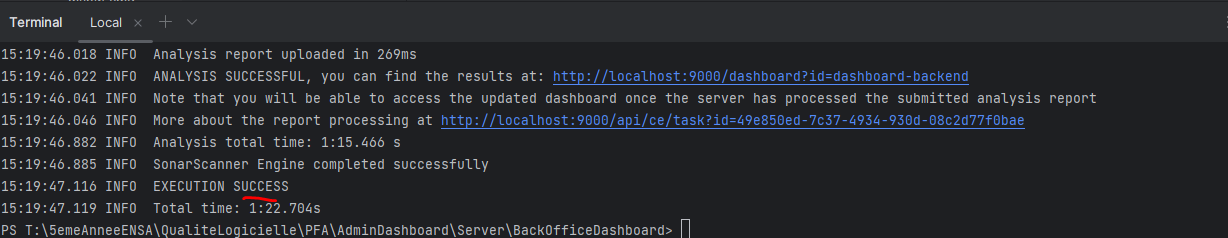
NOTE : On peut utiliser l’utile (Plugin) SonaLint pour tester le project loacalement.

Création de sonar-project.properties File:

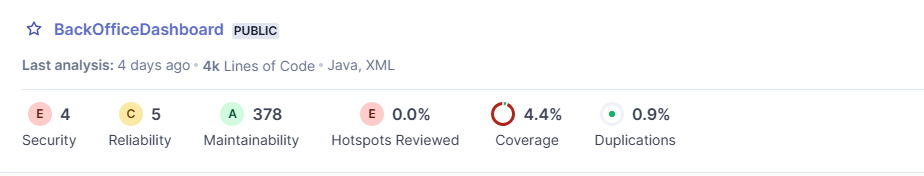


Run SonarScanner :



****

1. **Analyse du code source :**

****

Les métriques de qualité suivantes ont été analysées via SonarQube pour évaluer le code source :

|  |  |
| --- | --- |
| **Métrique** | **Description** |
| **Lignes de code** | Indique la taille totale du code base, permettant de comprendre l'échelle du projet. |
| **Bugs** | Défauts identifiés dans le code qui peuvent entraîner des pannes ou des comportements incorrects. |
| **Vulnérabilités** | Problèmes de sécurité détectés, qui peuvent exposer l'application à des attaques. |
| **Code dupliqué** | Pourcentage de code qui est dupliqué, réduisant la maintenabilité. |
| **Couverture de code** | Pourcentage de code testé par des tests unitaires, mesurant la fiabilité des tests. |
| **Complexité cyclomatique** | Complexité du code source, influençant sa lisibilité et sa maintenabilité. |
| **Dépendances** | Évaluation des dépendances tierces pour identifier des failles de sécurité ou des problèmes de compatibilité. |

**Problèmes identifiés dans le projet :**

|  |  |  |
| --- | --- | --- |
| **Catégorie** | **Problème** | **Exemple spécifique** |
| **Fiabilité** | 327 problèmes, incluant des erreurs logiques pouvant provoquer des pannes. | Une boucle for mal formée entraînant des comportements imprévisibles. |
| **Maintenabilité** | 569 problèmes, comme des noms de variables peu explicites ou des méthodes trop longues. | Une méthode processData() de 150 lignes regroupant plusieurs responsabilités. |
| **Couverture de code** | 0%, aucune couverture de tests unitaires. | Aucune méthode n'est testée, donc la fiabilité globale est inconnue. |
| **Duplications** | 11.2%, suggérant une répétition importante du code. | Dupliquer la logique de validation des formulaires dans plusieurs contrôleurs. |
| **Points chauds de sécurité** | 4 points non examinés, exposant le projet à des vulnérabilités potentielles. | Absence de validation d'entrée dans une API, rendant le projet vulnérable aux injections SQL. |

1. **Rapport des résultats :**

**Points faibles identifiés :**

* Problèmes de fiabilité :

Plusieurs bugs détectés dans les blocs de logique métier.

Des erreurs dans la gestion des boucles ou des conditions peuvent entraîner des comportements inattendus.

* Problèmes de maintenabilité :

Méthodes trop longues avec une complexité cyclomatique élevée, rendant le code difficile à comprendre et à maintenir.

Duplication du code, ce qui augmente le risque d’incohérences et rend les modifications complexes.

* Couverture de code insuffisante :

Aucune couverture de tests unitaires, ce qui limite la fiabilité des nouvelles fonctionnalités et la détection des régressions.

**Propositions pour améliorer la qualité du code :**

|  |  |
| --- | --- |
| **Problème** | **Solution proposée** |
| Bugs | 1. Examiner chaque bug identifié par SonarQube.  2. Ajouter des validations explicites et des vérifications des conditions limites. |
| Méthodes longues | Refactoriser les méthodes longues en les divisant en méthodes plus petites et spécifiques à une responsabilité. |
| Complexité cyclomatique élevée | Simplifier les structures conditionnelles et utiliser des abstractions pour réduire la complexité. |
| Code dupliqué | Extraire le code répété dans des méthodes ou des classes utilitaires réutilisables. |
| Absence de tests unitaires | Ajouter des tests unitaires pour chaque méthode critique.  Utiliser des frameworks comme JUnit et Mockito. |
| Points chauds de sécurité | 1. Examiner et résoudre les points chauds de sécurité.  2. Ajouter des validations d’entrée pour prévenir les injections SQL et XSS. |
| Couverture de code | S'assurer que les tests couvrent tous les cas possibles et suivre une couverture minimale de 80%. |

**Chapitre 3**

**Tests Boîte Blanche**

1. **Développement des tests unitaires :**

Les tests Boîte Blanche consistent à vérifier le fonctionnement interne du code source en testant les différentes branches, conditions et boucles. Ces tests permettent d'identifier les erreurs dans la logique du programme et les conditions limites. Dans ce projet, nous avons utilisé JUnit pour Java afin de développer des tests unitaires ciblant les zones critiques du code.

1. **Configuration de l’environnement :**

**L’ajoute de la dépendance JUnit dans Pom.xml**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-test</artifactId>**

**<scope>test</scope>**

**</dependency>**

1. **Écriture des tests unitaires :**

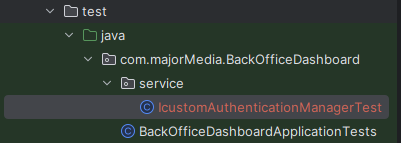
Il est essentiel de cibler celles qui sont critiques pour la logique métier, impliquent des interactions complexes, ou ont un impact sur la sécurité et les fonctionnalités principales.

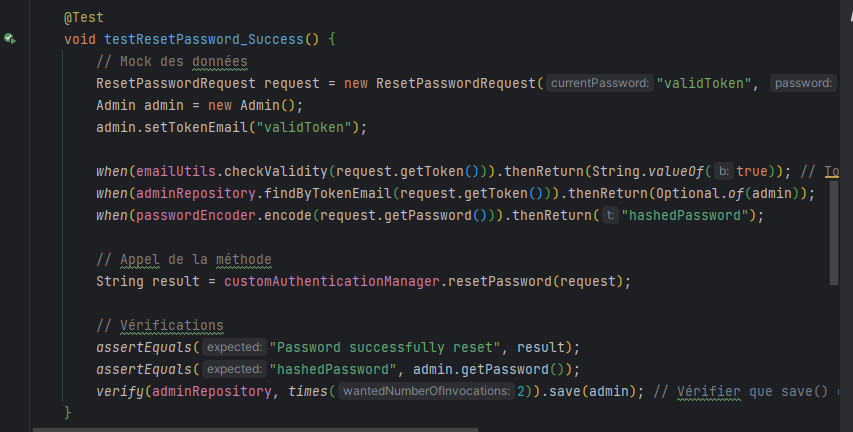
**Méthode de sécurité : resetPassword (IcustomAuthenticationManager)**

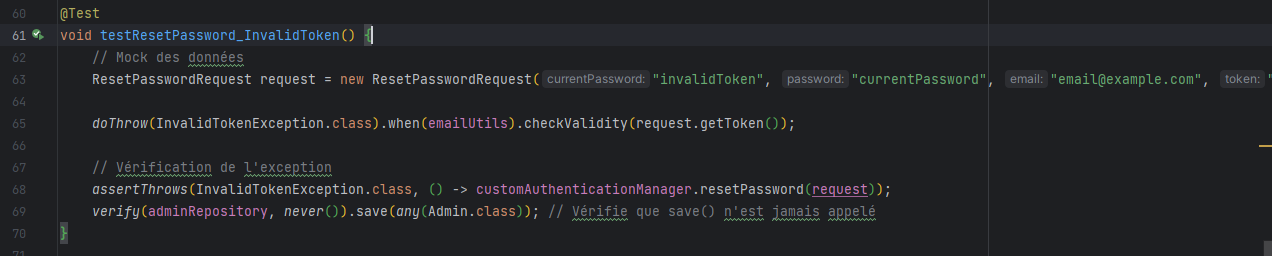


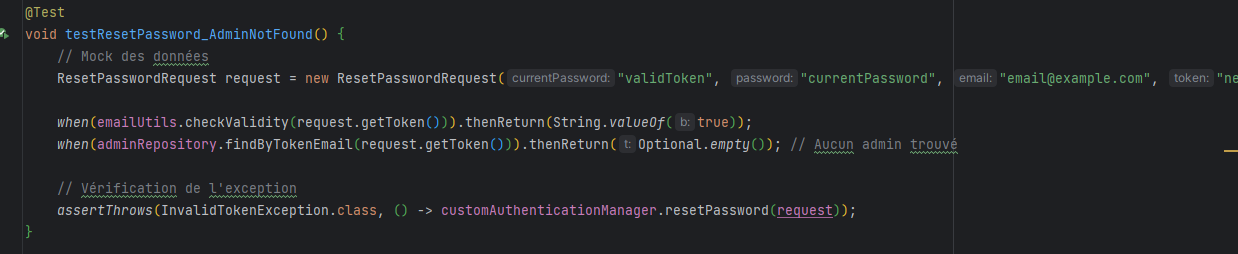
Pourquoi tester cette méthode ?

La méthode resetPassword effectue plusieurs opérations critiques, comme la validation du token, la récupération de l'utilisateur, le hachage du mot de passe, et la mise à jour dans la base de données. Voici comment vous pouvez écrire les tests unitaires pour cette méthode.





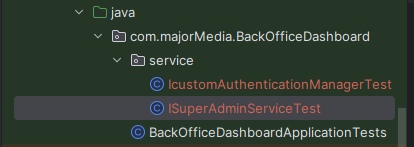




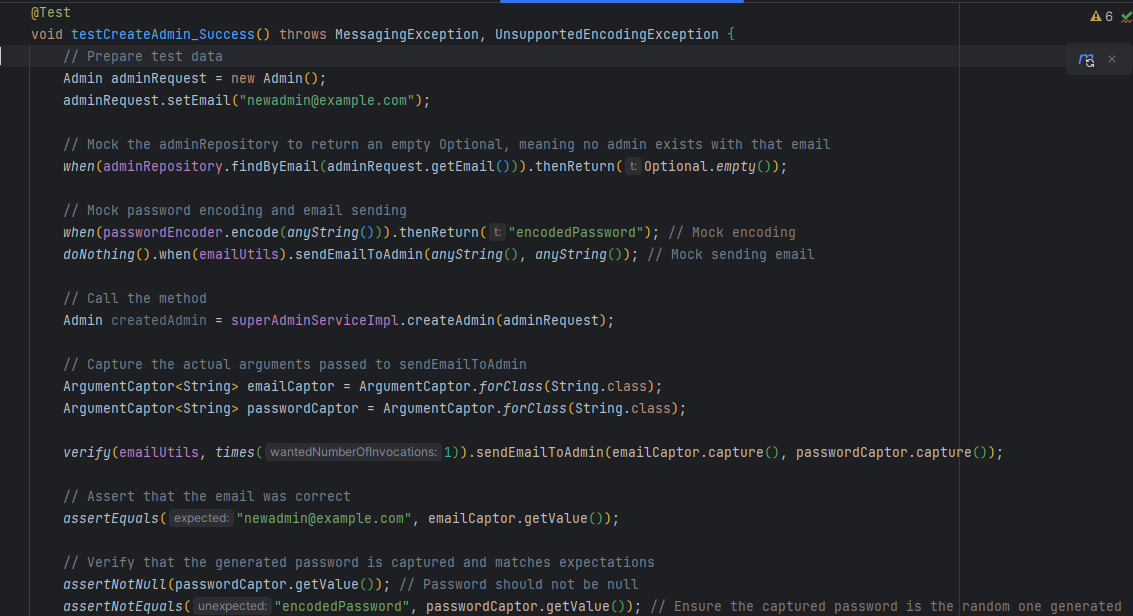
**Méthode critique : createAdmin (ISuperAdminService):**

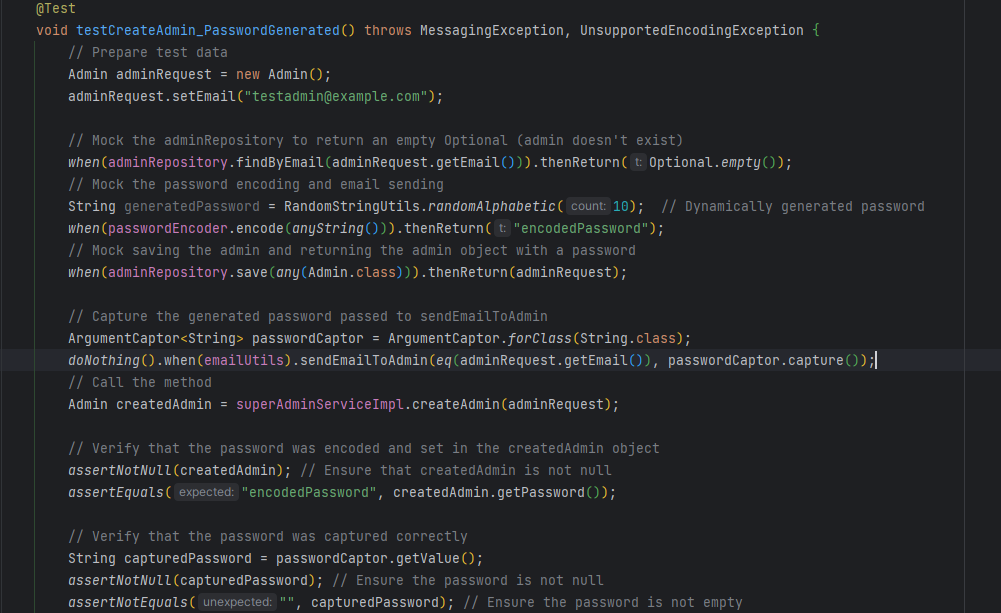
Pourquoi tester cette méthode ?

Elle gère la création des administrateurs et l'assignation de mots de passe sécurisés. Cela garantit que la logique de sécurité, comme le cryptage des mots de passe, fonctionne correctement et que les erreurs (comme un email déjà existant) sont correctement traitées. En outre, cela vérifie que l'email de notification est bien envoyé avec les bonnes informations.

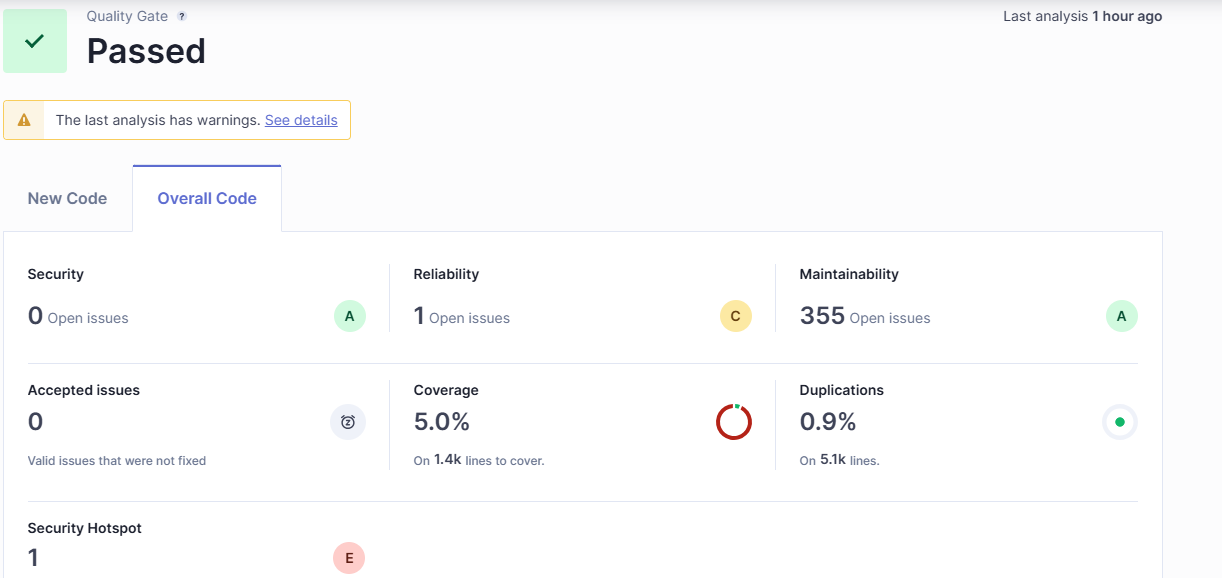
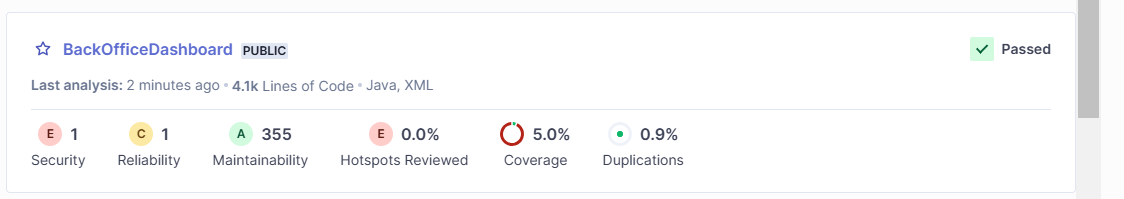








**Itérations des tests :**

****

1. **Documentation**