



ECOLE MAROCAINE DES
SCIENCES DE L'INGENIEUR
Membre de
HONORIS UNITED UNIVERSITIES

Rapport

ARCHITECTURE DES COMPOSANTS D'ENTREPRISE

**Conception d'une Application Mobile de Suivi de
l'Énergie avec l'Implémentation des
Microservices**

Réalisé par:

Zineb OUNAIR

Anas JEGOUAL

Badr AIT HAMMOU

Table des matières

1. Introduction	1
1.1. Aperçu du projet	1
1.2. Importance de l'architecture microservices	1
2. Architecture Microservices	1
2.1. Architecture	1
2.2. Description des services	3
2.3. Mécanismes de communication	3
3. Conception des Microservices	3
3.1. Approche de conception pour chaque service	3
4. Conteneurisation avec Docker	5
4.1. Implémentation	5
4.2. Avantages.....	9
5. CI/CD avec Jenkins	10
5.1. Processus.....	10
5.2. Configuration	11
6. Intégration de SonarQube	13
6.1. Configuration	13
6.2. Bénéfices pour la qualité du code.....	15
7. Conclusion	16
7.1. Résumé des accomplissements	16
7.2. Perspectives futures.....	16

1. Introduction

1.1. Aperçu du projet

Le projet vise à développer une application mobile dédiée au suivi de la consommation d'énergie, offrant une interface intuitive et en temps réel pour analyser les données associées. Cette initiative s'adresse à une gamme d'utilisateurs, tels que les responsables énergétiques, les gestionnaires de l'infrastructure, et d'autres parties prenantes, avec pour objectif principal de fournir une vue consolidée des tendances de consommation, des fluctuations de coûts, et des performances énergétiques.

L'application comprend deux principaux microservices, "**fuel_spring_server**" et "**fuel_flask_server**," chacun jouant un rôle spécifique dans la gestion des fonctionnalités liées au suivi de la consommation d'énergie et à l'authentification des utilisateurs respectivement. Ces microservices sont conçus pour être autonomes, modulaires, et interagir de manière cohérente pour assurer une expérience utilisateur optimale.

1.2. Importance de l'architecture microservices

L'architecture microservices est cruciale pour notre projet car elle offre des avantages clés:

1. **Modularité et évolutivité:** En divisant l'application en petits modules autonomes, cela facilite le développement, la maintenance et l'ajout de nouvelles fonctionnalités de manière flexible et évolutive.
2. **Indépendance des services:** Chaque microservice fonctionne de manière autonome, avec sa propre base de données et ses fonctionnalités spécifiques. Cela simplifie la gestion des services, réduit les dépendances, et permet des déploiements plus fréquents.
3. **Communication légère:** Les microservices interagissent via des mécanismes légers comme les API REST, assurant une communication efficace. Cela améliore la réactivité du système et facilite l'intégration de nouveaux services.
4. **Isolation des problèmes:** En cas de problème dans un microservice, les autres continuent de fonctionner normalement, assurant la résilience globale du système. Cela permet une détection et une correction rapides des problèmes.
5. **Agilité du développement:** L'architecture microservices favorise une approche de développement agile avec des cycles rapides, des mises à jour fréquentes, et une distribution continue.

2. Architecture Microservices

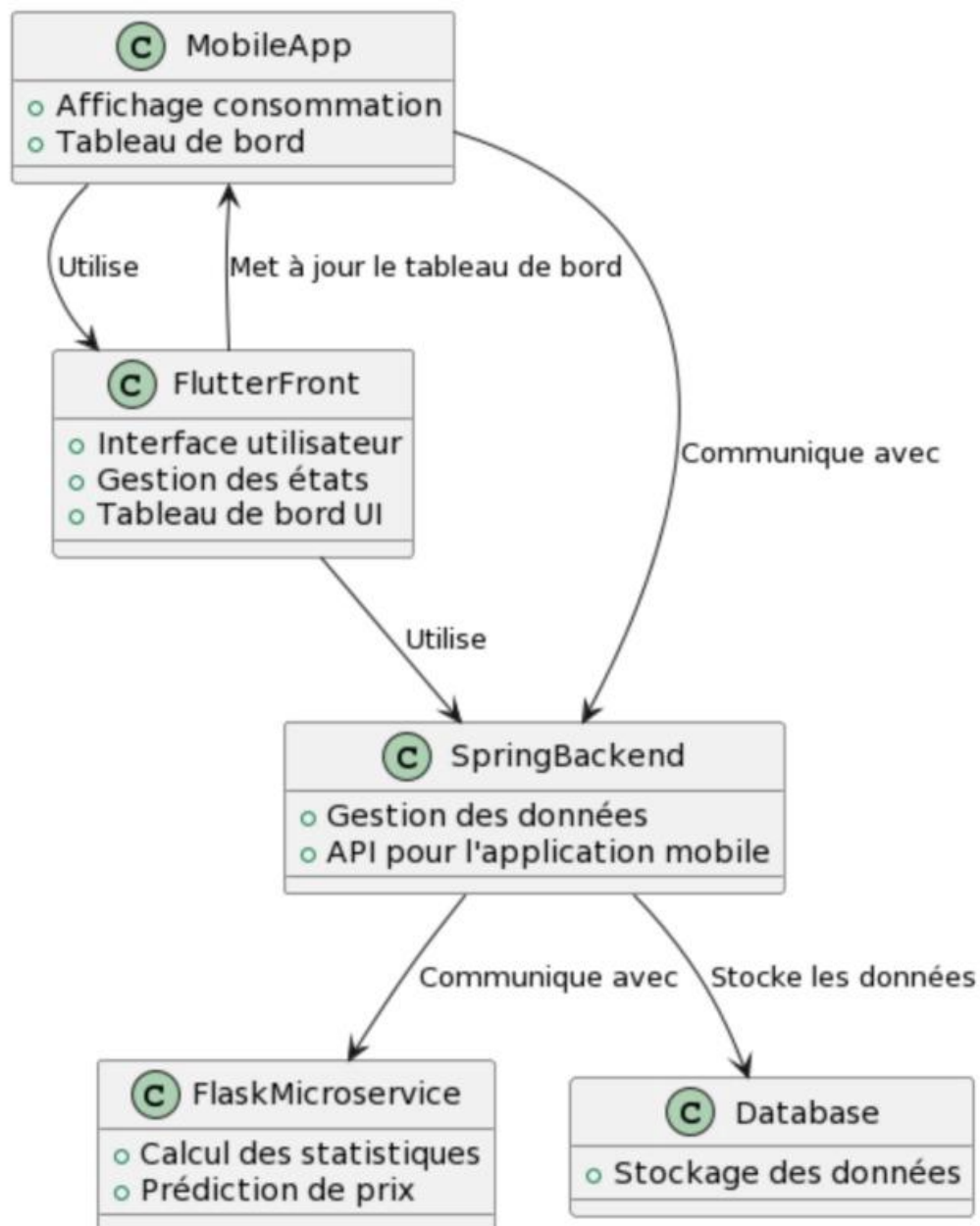
2.1. Architecture

L'architecture microservices du projet repose sur la décomposition de l'application en services autonomes, indépendants et spécialisés. Chaque microservice est conçu pour accomplir une

fonctionnalité spécifique, favorisant la modularité, la maintenance et l'évolutivité du système.

Caractéristiques Architecturales:

- ➔ Indépendance : Chaque microservice est indépendant et peut être déployé, mis à l'échelle, et mis à jour de manière isolée.
- ➔ Autonomie : Chaque microservice est autonome, possédant sa propre base de données et ses propres fonctionnalités.
- ➔ Communication légère : Les microservices **communiquent** entre eux de manière légère, généralement via des **API REST** ou des **messages asynchrones**.



2.2. Description des Services

➔ Microservice "fuel_spring_server" :

- Responsabilité : Gestion des fonctionnalités liées au suivi de la consommation d'énergie avec Spring Boot.
- Technologies : Utilisation de Spring Boot, Java.
- Communication : Communique avec d'autres microservices et composants via des requêtes REST.

➔ Microservice "fuel_flask_server" :

- Responsabilité : Gestion de l'authentification des utilisateurs avec Flask (Python).
- Technologies : Utilisation de Flask, Python.
- Communication: Interagit avec d'autres microservices via des API REST.

2.3. Mécanismes de Communication

➔ Communication Inter-Microservices :

- Utilisation de protocoles légers tels que HTTP/REST pour la communication entre les microservices.
- Mise en œuvre de passerelles API (API Gateways) pour faciliter l'accès aux microservices.

➔ Communication avec la Base de Données :

Chaque microservice possède sa propre base de données, assurant l'indépendance et la gestion autonome des données.

3. Conception des Microservices

3.1. Approche de Conception pour Chaque Service

3.1.1. Microservice "fuel_spring_server" :

- **Approche de Conception** : Adoption du modèle MVC (Modèle-Vue-Contrôleur) pour la séparation des préoccupations.
- **Base de Données** : Utilisation de bases de données relationnelles (MySQL) pour stocker les données liées à la consommation d'énergie.
- **Sécurité** : Mise en œuvre de mécanismes de sécurité tels que l'authentification et l'autorisation pour protéger les ressources.

3.1.2. Microservice "fuel_flask_server" :

- **Approche de Conception** : Adoption d'une architecture orientée événements pour la gestion de l'authentification des utilisateurs.
- **Sécurité** : Implémentation de stratégies de sécurité adaptées à l'environnement Python Flask.

3.1.3. "Eureka" :

- Approche de Conception : Modèle de registre de service pour la découverte automatique des microservices.
- Rôle : Registre centralisé facilitant l'enregistrement et la découverte des microservices.

Application	AMIs	Availability Zones	Status
FUEL-FLASK	n/a (1)	(1)	UP (1) - 127.0.0.1:fuel-flask:8070
FUEL-GATEWAY	n/a (1)	(1)	UP (1) - 8f42741804a2:fuel-gateway:8888
FUEL-SPRING	n/a (1)	(1)	UP (1) - b88152f10ad8:fuel-spring:8090

Name	Value
total-avail-memory	83mb
num-of-cpus	2
current-memory-usage	40mb (48%)

3.1.4. Microservice "Gateway" :

- Approche de Conception : Modèle de passerelle pour orchestrer les requêtes entre clients et microservices.
- Fonctionnalités : Gestion des autorisations, acheminement des requêtes, et interface unifiée pour les clients.

Cette approche de conception pour chaque microservice garantit une autonomie fonctionnelle tout en facilitant l'évolutivité et la maintenance. La communication légère entre les microservices contribue à une architecture flexible et adaptable aux exigences changeantes du projet.

4. Conteneurisation avec Docker

4.1. Implémentation

dockerfile du microService d'authentification avec Flask

```
FROM maven:3.8.4-openjdk-17 AS builder
WORKDIR /app
COPY ./src ./src
COPY ./pom.xml .
RUN mvn clean package

FROM openjdk:17-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} server.jar
ENTRYPOINT ["java","-jar","/server.jar"]
```

dockerfile de MicroService Fuel avec Spring

```
FROM maven:3.8.4-openjdk-17 AS builder
WORKDIR /app
COPY ./src ./src
COPY ./pom.xml .
RUN mvn clean package

FROM openjdk:17-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} server.jar
ENTRYPOINT ["java","-jar","/server.jar"]
```

Fichier “docker-compose.yml”:

MySQL:

```
version: '3'

services:
  mysql:
    image: mysql:latest
    container_name: mysql-container1
    environment:
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
    networks:
      - microservices-network
```

PhpMyAdmin:

```
phpmyadmin:
  image: phpmyadmin/phpmyadmin
  environment:
    PMA_HOST: mysql
    PMA_PORT: 3306
    MYSQL_ROOT_PASSWORD: root
  ports:
    - "8081:80"
  networks:
    - microservices-network

networks:
  microservices-network:
    driver: bridge
```


Eureka:

```
version: '3'

services:
  mysql:
    image: mysql:latest
    container_name: mysql-container1
    environment:
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
    networks:
      - microservices-network
```

Gateway:

```
gateway-service:
  build:
    context: ./fuel_gateway
  ports:
    - "8888:8888"
  depends_on:
    - mysql
    - eureka
  networks:
    - microservices-network
  environment:
    SPRING_CLOUD_EUREKA_HOST: eureka
    SPRING_CLOUD_EUREKA_PORT: 8010
    SPRING_CLOUD_EUREKA_ENABLED: 'true'
```

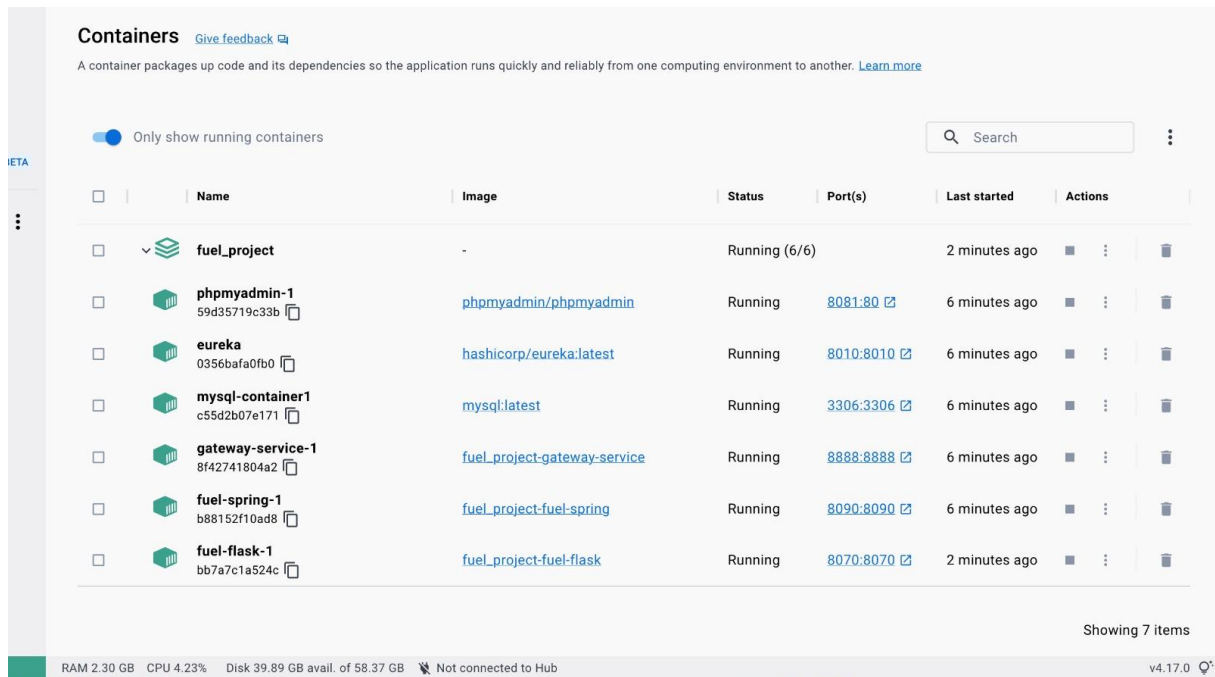
Flask:





























```
fuel-flask:
  build:
    context: ./fuel_flask_server
  ports:
    - "8070:8070"
  depends_on:
    - mysql
    - eureka
    - gateway-service
  networks:
    - microservices-network
  environment:
    SPRING_CLOUD_EUREKA_HOST: eureka
    SPRING_CLOUD_EUREKA_PORT: 8010
    SPRING_CLOUD_EUREKA_ENABLED: 'true'
    SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/fuel?createDatabaseIfNotExist=true
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: root
  healthcheck:
    test: "/usr/bin/mysql --user=root --password=root --execute \"SHOW DATABASES;\""
    interval: 5s
    timeout: 2s
    retries: 100
```

Fuel-Spring:

```
fuel-spring:
  build:
    context: ./fuel_spring_server
  ports:
    - "8090:8090"
  depends_on:
    - mysql
    - eureka
    - gateway-service
  networks:
    - microservices-network
  environment:
    SPRING_CLOUD_EUREKA_HOST: eureka
    SPRING_CLOUD_EUREKA_PORT: 8010
    SPRING_CLOUD_EUREKA_ENABLED: 'true'
    SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/fuel?createDatabaseIfNotExist=true
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: root
  healthcheck:
    test: "/usr/bin/mysql --user=root --password=root --execute \"SHOW DATABASES;\""
    interval: 5s
    timeout: 2s
    retries: 100
```

Images déployées dans docker:



	Name	Image	Status	Port(s)	Last started	Actions
<input type="checkbox"/>	 fuel_project	-	Running (6/6)		2 minutes ago	  
<input type="checkbox"/>	 phpmyadmin-1 59d35719c33b	phpmyadmin/phpmyadmin	Running	8081:80	6 minutes ago	  
<input type="checkbox"/>	 eureka 0356bafa0fb0	hashicorp/eureka:latest	Running	8010:8010	6 minutes ago	  
<input type="checkbox"/>	 mysql-container1 c55d2b07e171	mysql:latest	Running	3306:3306	6 minutes ago	  
<input type="checkbox"/>	 gateway-service-1 8f42741804a2	fuel_project-gateway-service	Running	8888:8888	6 minutes ago	  
<input type="checkbox"/>	 fuel-spring-1 b88152f10ad8	fuel_project-fuel-spring	Running	8090:8090	6 minutes ago	  
<input type="checkbox"/>	 fuel-flask-1 bb7a7c1a524c	fuel_project-fuel-flask	Running	8070:8070	2 minutes ago	  

Showing 7 items

RAM 2.30 GB CPU 4.23% Disk 39.89 GB avail. of 58.37 GB Not connected to Hub v4.17.0

4.2. Avantages

L'implémentation avec Docker présente plusieurs avantages significatifs pour le déploiement et la gestion des applications. Parmi des avantages clés:

- **Rapidité de Déploiement** : Les conteneurs peuvent être déployés rapidement, réduisant ainsi le temps nécessaire à la mise en production de nouvelles fonctionnalités ou de correctifs.
- **Gestion des Dépendances** : Docker simplifie la gestion des dépendances en encapsulant toutes les bibliothèques et composants nécessaires à l'intérieur du conteneur, éliminant les conflits de version.
- **Évolutivité Facilitée** : Docker facilite l'évolutivité horizontale en permettant le déploiement de plusieurs instances de conteneurs, équilibrant la charge entre elles.
- **Gestion des Versions** : Les images Docker permettent de versionner les applications, facilitant la gestion des différentes versions dans des environnements différents.
- **Facilité de Gestion** : Docker simplifie la gestion des cycles de vie des applications, de la création à la suppression, en passant par la mise à l'échelle et la mise à jour.
- **Sécurité** : Les conteneurs Docker offrent des mécanismes de sécurité, tels que les espaces de noms et les contrôles d'accès, pour isoler les applications et prévenir les vulnérabilités.

5. CI/CD avec Jenkins

5.1. Processus

Le processus CI/CD (Intégration Continue / Déploiement Continu) avec Jenkins est un moyen de garantir une approche automatisée et continue tout au long du cycle de vie du développement logiciel, de l'intégration des modifications au déploiement en production. Le processus CI/CD avec Jenkins suit ces étapes pour assurer un développement et un déploiement continus:

Intégration Continue (CI):

- ➔ Versionnement : les développeurs travaillent sur des branches de fonctionnalités, et tout le code est géré avec Git.
- ➔ Déclenchement automatique : à chaque modification du code (push), Jenkins détecte automatiquement les changements grâce à des hooks Git ou des webhooks.
- ➔ Compilation (Build) : Jenkins récupère le code, le compile et exécute des tests unitaires pour garantir sa fonctionnalité.

Déploiement Continu (CD):

- ➔ Tests Automatisés : Après la compilation, Jenkins lance des tests automatisés (intégration, système) pour assurer la stabilité et la qualité.
- ➔ Package : Si les tests réussissent, Jenkins prépare l'application pour le déploiement, incluant la création d'images Docker ou la génération d'artefacts.
- ➔ Déploiement sur Environnement de Staging : L'application est déployée sur un environnement de staging pour des tests approfondis avant la production.
- ➔ Tests sur Environnement de Staging : Des tests avancés, comme les tests de charge et d'acceptation, sont effectués sur l'environnement de staging.
- ➔ Approbation : En cas de succès des tests, une approbation manuelle ou automatisée est requise pour déclencher le déploiement en production.
- ➔ Déploiement en Production : Jenkins déploie l'application sur l'environnement de production après approbation.
- ➔ Surveillance : Des outils de surveillance sont configurés pour suivre les performances et la stabilité de l'application en production.

5.2. Configuration

```
pipeline {
  agent any
  tools {
    maven 'maven'
  }
  stages {
    stage('Git Clone') {
      steps {
        script {
          checkout([$class: 'GitSCM', branches: [[name: 'main']],
            userRemoteConfigs: [[url: 'https://github.com/AnasJeg/fuel-project.git']]])
        }
      }
    }

    stage('Build Authentification') {
      steps {
        script {
          dir('flask_server') {
            bat './mvnw clean install'
          }
        }
      }
    }

    stage('Build Spring') {
      steps {
        script {
          dir('fuel_spring_server') {
            bat './mvnw clean install -DskipTests'
          }
        }
      }
    }
  }
}
```

try sample Pipeline... ▾

```
    stage('Build Docker Images') {
      steps {
        script {
          bat 'docker-compose build'
        }
      }
    }

    stage('Run') {
      steps {
        script {
          bat 'docker-compose up -d'
        }
      }
    }
  }
}
```

Explication des Stages dans le Pipeline Jenkins :

1. Git Clone :

Objectif : Récupérer le code source du projet depuis le référentiel Git.

Étapes : Utilisation de la commande checkout pour cloner le dépôt Git. La branche principale (main) est spécifiée comme la branche cible.

2. Build Authentification :

Objectif : Compiler le code source du microservice d'authentification basé sur Flask.

Étapes : Utilisation de Maven (./mvnw) pour exécuter la phase de build avec la commande clean install. Cela assure la compilation du code et la création d'artefacts, mais les tests sont ignorés (-DskipTests).

3. Build Spring :

Objectif : Compiler le code source du microservice Spring.

Étapes : Utilisation de Maven (./mvnw) pour exécuter la phase de build avec la commande clean install. Les tests sont ignorés (-DskipTests) pour accélérer le processus de build.

4. Build Docker Images:

Objectif : Construire les images Docker à partir des artefacts générés pendant le build.

Étapes : Utilisation de docker-compose build pour créer les images Docker en fonction des configurations spécifiées dans le fichier docker-compose.yml. Cette étape prépare les services pour le déploiement dans des conteneurs Docker.

5. Run :

Objectif : Démarrer les conteneurs Docker pour exécuter l'application.

Étapes : Utilisation de docker-compose up -d pour lancer les conteneurs en mode détaché (background). Cette étape rend l'application disponible pour l'utilisation, et les services peuvent interagir entre eux selon les spécifications du fichier docker-compose.yml.

Stage View



Liens permanents

6. Intégration de SonarQube

6.1. Configuration

➔ Configuration de SonarQube dans Jenkins :

Nous avons ajouté les informations de configuration de SonarQube dans les paramètres globaux de Jenkins, notamment l'URL de notre serveur SonarQube et les informations d'authentification nécessaires.

➔ Configuration du Projet dans SonarQube :

Assurer que nos projets Spring et Flask sont configurés dans SonarQube. Nous pouvons le faire manuellement ou à l'aide de scripts dans notre pipeline Jenkins.

➔ Ajout de l'Analyse SonarQube dans le Pipeline :

Modifier notre script Jenkins pour inclure l'étape d'analyse de code SonarQube après la construction du projet.

```
stage('SonarQube') {
    steps {
        script {
            dir('fuel_spring_server') {
                bat 'mvn sonar:sonar'
            }
        }
    }
}
stage('Build Docker Images') {
    steps {
        script {
            bat 'docker-compose build'
        }
    }
}
stage('Run') {
    steps {
        script {
            bat 'docker-compose up -d'
        }
    }
}
}}
```

Stage « SonarQube » :

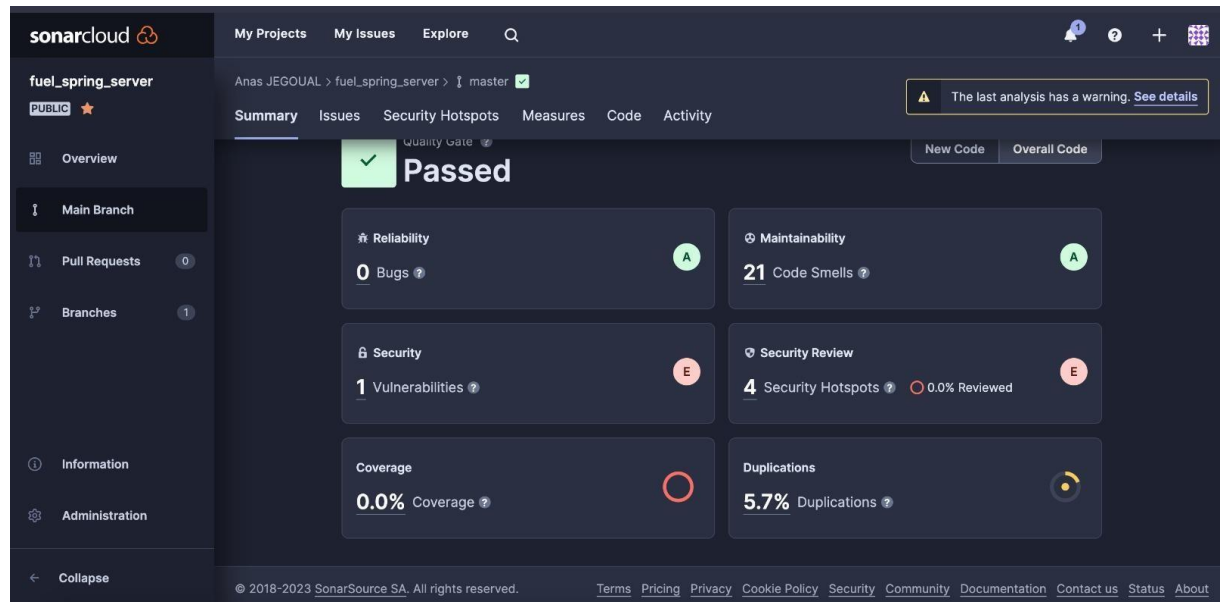
Responsable de l'analyse du code source du microservice "fuel_spring_server" à l'aide de SonarQube. Voici une explication des éléments clés :

`dir('fuel_spring_server')` : Change le répertoire de travail vers le dossier du microservice "fuel_spring_server."

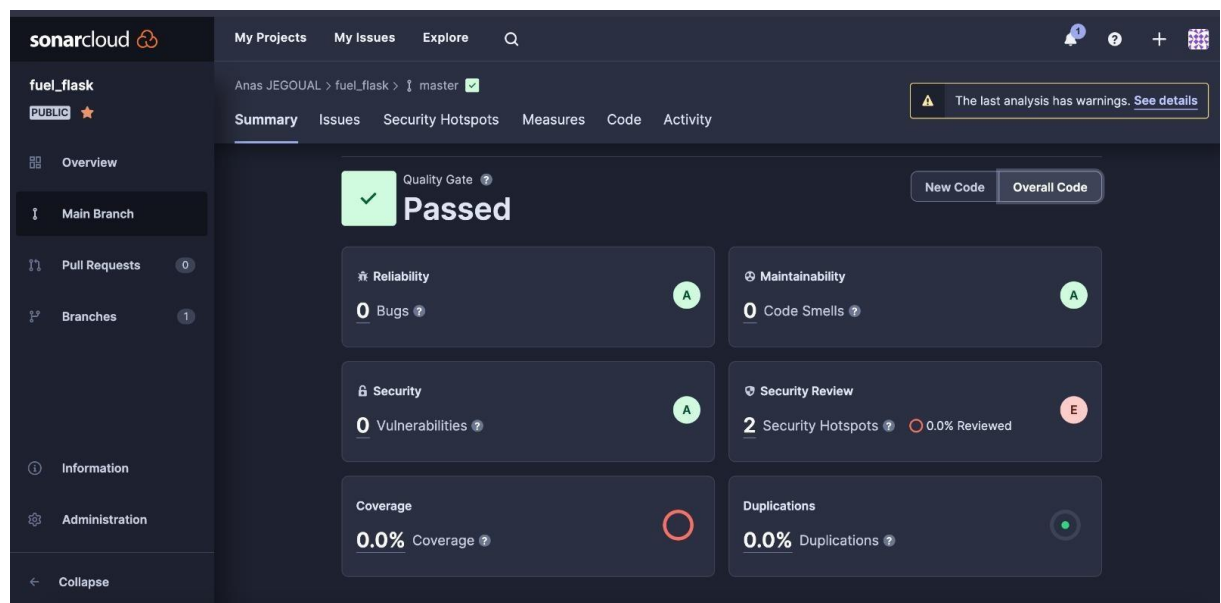
`bat 'mvn sonar:sonar'` : Exécute la commande Maven permettant de déclencher l'analyse de code SonarQube. Cela génère des métriques et des rapports détaillés sur la qualité du code, tels que la couverture de code, la complexité, les violations de règles, etc.

Ce stage automatise le processus d'analyse de code du microservice "fuel_spring_server" avec SonarQube, intégrant ainsi l'évaluation continue de la qualité du code dans le pipeline Jenkins.

- Pour "fuel_spring_server"



- Pour "fuel_flask"



6.2. Bénéfices pour la qualité du code :

Détection Précoce des Problèmes: SonarQube identifie les problèmes de qualité du code tôt dans le processus de développement.

Mesures de la Qualité du Code: Obtenir des métriques telles que le taux de couverture de code, la complexité cyclomatique, la duplication de code, etc.

Priorisation des Corrections: SonarQube classe les problèmes en fonction de leur gravité, permettant une priorisation efficace des corrections.

Amélioration Continue: En intégrant SonarQube dans le pipeline, chaque nouvelle itération du code est évaluée, favorisant l'amélioration continue de la qualité du code.

Standardisation du Code: Définir des règles de qualité du code et garantir que le code suit les normes établies.

Bénéfice de JaCoCo: L'intégration de JaCoCo avec SonarQube permet une évaluation exhaustive de la qualité du code, en fournissant des rapports détaillés sur la couverture de code, les bugs potentiels, les vulnérabilités de sécurité:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>${jacoco.version}</version>
      <executions>
        <execution>
          <id>jacoco-initialize</id>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
        </execution>
        <execution>
          <id>jacoco-site</id>
          <phase>package</phase>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

En utilisant SonarQube de manière proactive, nous pouvons maintenir un code plus propre, plus fiable et plus maintenable tout au long du cycle de vie de développement de notre application.

7. Conclusion

7.1. Résumé des accomplissements

Le projet a réalisé des accomplissements significatifs en mettant en œuvre une application mobile de suivi de la consommation d'énergie basée sur une architecture microservices. Les points saillants comprennent l'utilisation de technologies modernes telles que Spring Boot, Flask, Flutter, Docker, et l'adoption de pratiques de développement robustes.

Architecture Microservices: Mise en place de microservices avec Spring Boot et Flask, offrant une architecture modulaire, évolutive et facile à maintenir.

Frontend avec Flutter: Développement d'une interface utilisateur réactive et multiplateforme avec Flutter, améliorant l'expérience utilisateur.

Conteneurisation avec Docker: Utilisation de Docker pour la conteneurisation des microservices, assurant une gestion simplifiée du déploiement et de l'isolation des services.

Intégration Continue avec Jenkins: Configuration d'un pipeline Jenkins automatisé pour l'intégration continue, favorisant le développement agile et la détection précoce des erreurs.

Qualité du Code avec SonarQube et JaCoCo: Intégration de SonarQube et JaCoCo pour évaluer et améliorer la qualité du code, avec des rapports détaillés sur la couverture de code et les problèmes potentiels.

7.2. Perspectives futures

Notre projet démontre quelques perspectives:

Expansion des Fonctionnalités: Ajout de fonctionnalités avancées pour enrichir l'expérience utilisateur et répondre aux besoins émergents.

Déploiement automatique avec Ngrock.

Déploiement automatique avec Azure Cloud.

Intégration de Nouvelles Technologies: Explorer et intégrer de nouvelles technologies pour rester à jour avec les avancées technologiques et répondre aux exigences changeantes.

Tests Approfondis: Élargir les suites de tests automatisés et manuels pour garantir une stabilité continue et une robustesse accrue.

En résumé, ce projet, tout en réalisant des accomplissements notables, offre une base solide pour évoluer et s'adapter aux défis futurs, démontrant ainsi un engagement envers l'innovation continue et la satisfaction des utilisateurs.