



RAPPORT PROJET : ARCHITECTURE DES COMPOSANTS D'ENTREPRISE

Triple A Travel Story

Réalisé par

ANAS OUATIL
AYA KHATBOUCH
ABDELMONAIM ERRAJU

Table des matières

Introduction :	2
Architecture Microservices:	3
Conception des Microservices:	4
Conteneurisation avec Docker:	6
CI/CD avec Jenkins:	7
Intégration de SonarQube:	10
Conclusion:	14

Introduction :

Aujourd'hui, dans l'ère numérique, la manière dont nous partageons nos expériences de voyage a évolué de manière significative. C'est dans cette perspective que le projet "Triple A Travel Story" voit le jour. Notre ambition est de créer une plateforme novatrice, un blog de voyage interactif, permettant aux utilisateurs de partager leurs récits, photos et conseils de manière immersive.

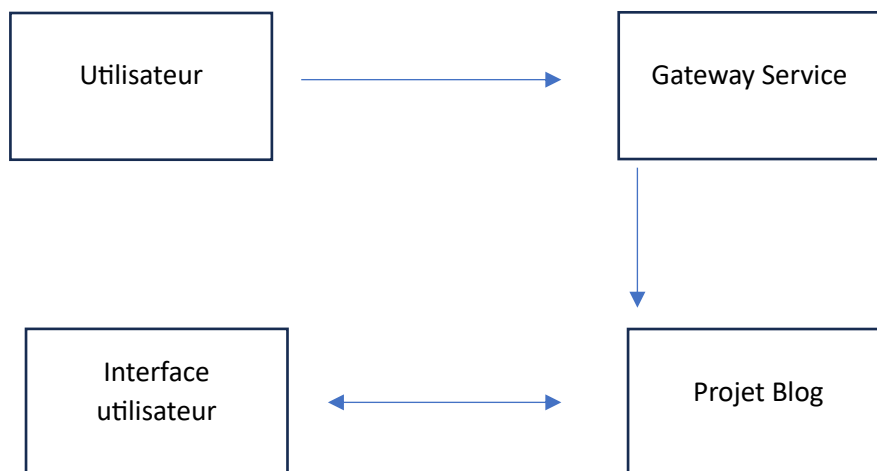
Pour réaliser ce projet ambitieux, nous avons opté pour une approche moderne et modulaire en utilisant une architecture basée sur des microservices. Ils offrent une flexibilité et une extensibilité accrues, permettant ainsi une évolution rapide de l'application tout en optimisant la gestion des ressources. Cette approche nous permettra de développer, déployer et maintenir chaque composant de manière indépendante, favorisant ainsi l'efficacité du développement.

L'outil principal de développement pour l'interface utilisateur de Triple A Travel Story sera React, une bibliothèque JavaScript largement utilisée pour la création d'interfaces utilisateur dynamiques et réactives. Grâce à React, nous pourrions concevoir une expérience utilisateur fluide et intuitive, offrant aux utilisateurs une immersion totale dans les récits de voyage partagés.

Dans ce rapport, nous explorerons en détail la conception et la mise en œuvre de chaque microservice, en mettant l'accent sur la manière dont ils interagissent pour former un écosystème complet. Nous examinerons également les choix technologiques spécifiques, les défis potentiels et les solutions envisagées pour garantir le succès de Triple A Travel Story.

Architecture Microservices:

L'architecture de Triple A Travel Story repose sur le modèle éprouvé des microservices, une approche permettant de décomposer l'application en petites unités autonomes et spécialisées. Cette fragmentation offre une flexibilité et une scalabilité accrues, facilitant la gestion indépendante de chaque service. Nous avons choisi cette architecture afin de favoriser la maintenance, le déploiement agile et l'évolutivité de notre application web.



1 .Gateway Service :

Gère la communication avec l'utilisateur.

Centralise la gestion des autorisations et de la sécurité.

Achemine les requêtes vers les services appropriés.

Agrège les données nécessaires pour la réponse.

2. Projet Blog :

Gère les publications, les commentaires et les données des récits de voyage.

Permet la création, la mise à jour et la suppression des articles.

Gère les interactions sociales au sein de la plateforme.

3.Mécanismes de communication

Les requêtes de l'utilisateur passent par le Gateway Service, qui expose une interface REST API.

Le Gateway Service utilise REST Template pour effectuer des appels à des services distants.

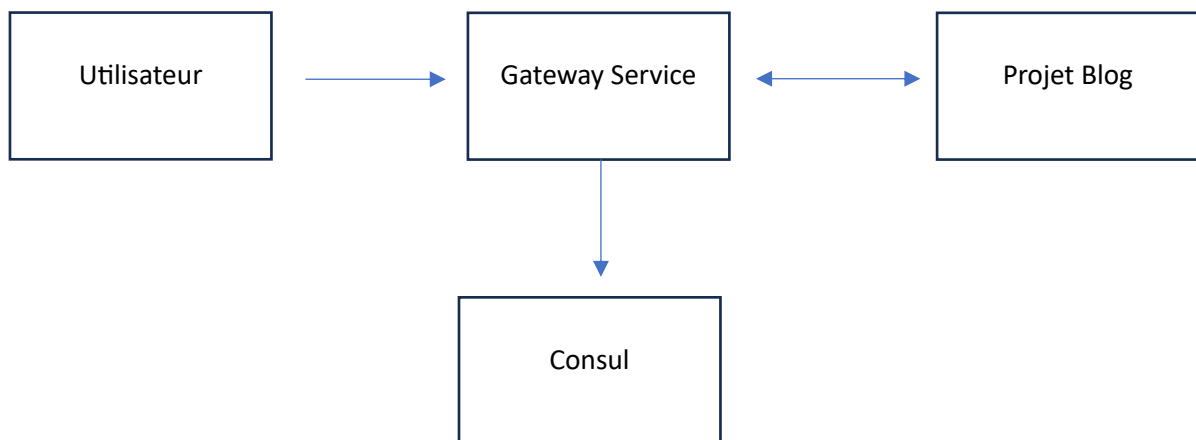
Les services, y compris le Projet Blog, exposent des interfaces REST API standardisées.

La communication entre les services se fait via des requêtes HTTP RESTful.

Le Gateway Service orchestre la communication entre les microservices en utilisant REST Template pour assurer la cohérence des interactions.

Cette architecture basée sur REST API et REST Template offre une flexibilité, une maintenabilité et une évolutivité tout en assurant une communication efficace entre les composants de Triple A Travel Story.

Conception des Microservices:



Microservice Gateway :

Découverte Dynamique :

Le Microservice Gateway s'appuie sur Consul pour la découverte dynamique des services.

Le Gateway peut identifier et se connecter aux instances actives des microservices disponibles.

Gestion des Configurations :

Consul stocke et récupère les configurations du Microservice Gateway.

Facilite la mise à jour des configurations sans redémarrage, assurant une gestion souple.

[Microservice Blog](#) :

Le Microservice Blog s'inscrit et se désinscrit dynamiquement auprès de Consul.

Consul maintient une liste à jour des instances actives du service.

Résilience et Tolérance aux Pannes :

Le Microservice Blog détecte les pannes et indisponibilités via Consul.

Mécanismes de résilience intégrés pour une adaptation transparente aux changements.

Consul :

[Registre de Services](#) :

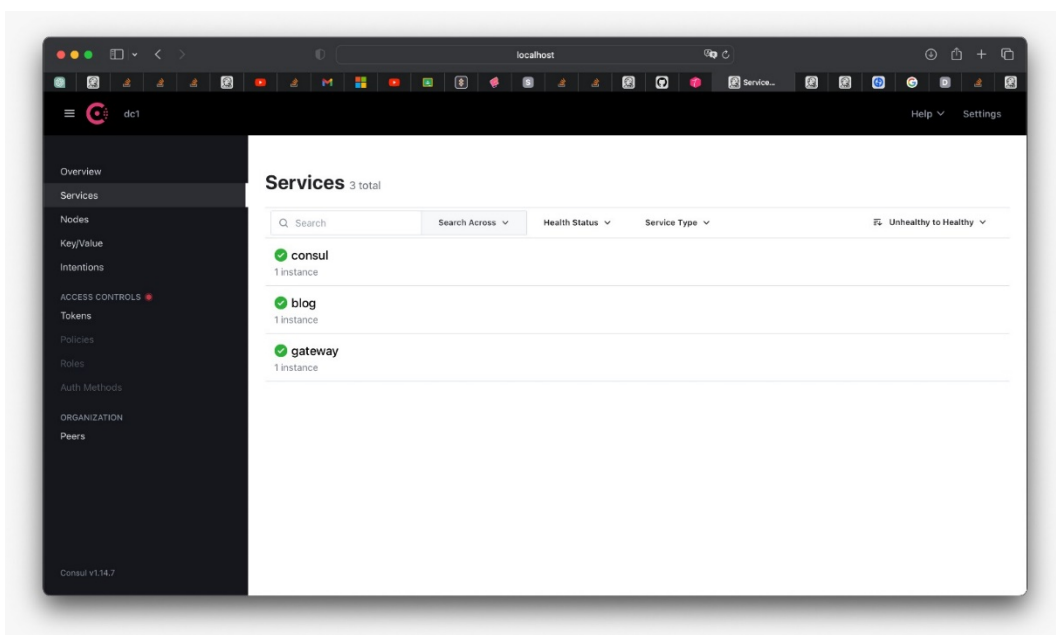
Consul est le point central pour le registre de services, contenant des informations en temps réel.

Les services s'inscrivent, se désinscrivent et consultent le registre pour découvrir les autres services.

Prise de Décision de Routage :

Consul facilite la prise de décision de routage pour le Microservice Gateway.

Fournit des informations actualisées sur les services disponibles.



Conteneurisation avec Docker:



































Avantages :

Portabilité : Les conteneurs Docker encapsulent tous les éléments nécessaires à l'exécution d'un microservice, garantissant ainsi une portabilité maximale. Un conteneur peut être déployé de manière cohérente sur n'importe quel environnement prenant en charge Docker, qu'il s'agisse d'un serveur local, d'un cloud public ou d'un service d'orchestration de conteneurs.

Isolation : Chaque microservice fonctionne dans son propre conteneur, ce qui assure une isolation complète des dépendances. Cela permet d'éviter les conflits de versions et facilite la mise à jour indépendante de chaque service, améliorant ainsi la maintenance et la flexibilité.

Facilité de Déploiement : Docker simplifie considérablement le processus de déploiement en encapsulant toutes les dépendances au sein du conteneur. Les déploiements peuvent être effectués de manière rapide et cohérente, garantissant une transition fluide entre les environnements de développement, de test et de production.

Évolutivité : La conteneurisation facilite l'évolutivité horizontale, permettant d'ajouter ou de retirer des instances de microservices de manière dynamique en fonction de la charge de travail. Les services peuvent être mis à l'échelle de manière indépendante pour répondre aux besoins spécifiques de chaque composant.

<input type="checkbox"/>		msblogtemplate		Running (5/5)		51 seconds ago			
<input type="checkbox"/>		phpmyadmin-1 9940effb64b 	phpmyadmin/phpmyadmin	Running	8082:80 	3 minutes ago			
<input type="checkbox"/>		mysql-container c18e8adb566 	mysql:latest	Running	3306:3306 	3 minutes ago			
<input type="checkbox"/>		consul-container1 4311ee683db6 	consul:1.15.4	Running	8500:8500 	3 minutes ago			
<input type="checkbox"/>		gateway-service-1 64d0b75d01be 	msblogtemplate-gateway-ser	Running	8888:8888 	3 minutes ago			
<input type="checkbox"/>		blog-service-1 7e84d7ab127a 	msblogtemplate-blog-service	Running	8081:8081 	51 seconds ago			

CI/CD avec Jenkins:

Processus CI/CD :

Intégration Continue (CI) :

Chaque fois qu'un développeur effectue un push sur le référentiel Git, le processus d'intégration continue est déclenché.

Jenkins récupère le code source depuis le référentiel, déclenche les tests automatisés et génère des rapports sur la qualité du code.

Les rapports de test et les métriques de qualité sont fournis aux développeurs, permettant une correction rapide des problèmes.

Déploiement Continue (CD) :

Lorsqu'une branche spécifique (par exemple, la branche principale) est validée après l'intégration continue, le processus de déploiement continu est activé.

Jenkins orchestre le déploiement des microservices dans des environnements de test, puis de pré-production, et enfin dans l'environnement de production, garantissant une progression en toute sécurité à travers les différentes étapes.

Configuration Jenkins :

Gestion des Dépendances :

Jenkins est configuré pour récupérer automatiquement les dépendances nécessaires à partir d'un gestionnaire de dépendances, assurant ainsi une reproductibilité de l'environnement.

Jenkinsfile :

Chaque microservice est associé à un Jenkinsfile, qui définit les étapes du pipeline CI/CD. Ce fichier contient les instructions nécessaires pour la construction, les tests, et le déploiement de l'application.

Tests Automatisés :

Jenkins est configuré pour exécuter des suites de tests automatisés à chaque intégration. Les résultats des tests sont rapportés, permettant d'identifier rapidement les éventuels problèmes de code.

Déploiement Graduel :

Le processus de déploiement continu est configuré pour une approche graduelle, permettant de minimiser les risques en déployant progressivement les nouvelles versions dans des environnements contrôlés avant d'atteindre la production.

Notifications :

Des notifications sont configurées à chaque étape du pipeline pour informer les membres de l'équipe des résultats, des succès et des échecs. Cela garantit une communication transparente tout au long du processus.

Script pipeline :

```
pipeline {
    agent any

    tools {
        maven 'maven'
    }

    stages {
        stage('Git Clone') {
            steps {
                script {
                    checkout([$class: 'GitSCM', branches: [[name: 'main']], userRemoteConfigs: [[url:
'https://github.com/OUATILANAS/microservice2.git']]])
                }
            }
        }

        stage('Build') {
            steps {
                sh 'cd blog && mvn clean install -DskipTests'
            }
        }
    }
}
```

```
stage('Build2') {  
    steps {  
        sh 'cd gateway && mvn clean install -DskipTests'  
    }  
}
```

```
stage('Check Docker Compose') {  
    steps {  
        script {  
            sh 'cd /usr/local/bin/ docker-compose --version'  
        }  
    }  
}
```

```
stage('Docker Build') {  
    steps {  
        script {  
            sh 'cd /usr/local/bin/ docker-compose build'  
        }  
    }  
}
```

```
stage('Docker Run') {  
    steps {  
        script {  
            sh 'cd /usr/local/bin/ docker-compose up'  
        }  
    }  
}
```

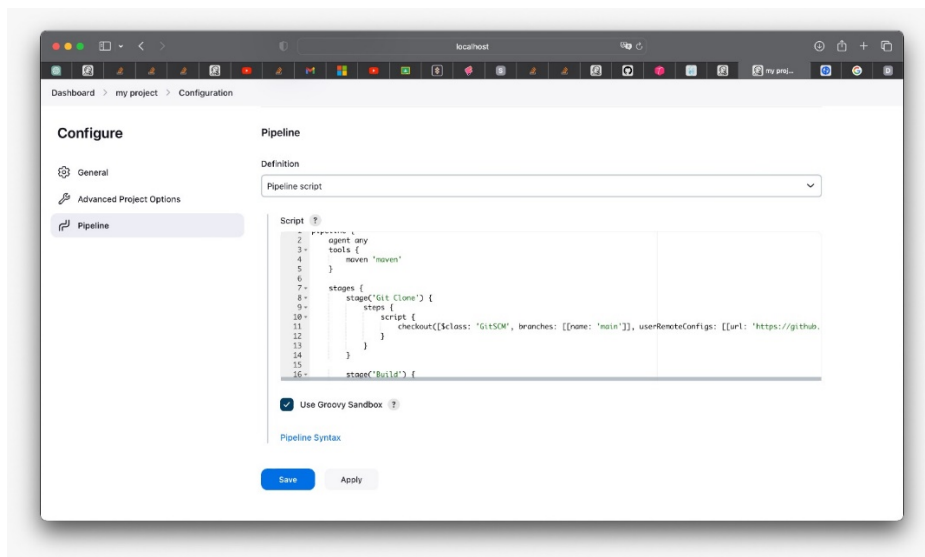


Tableau de bord > microservice2AO >

Status

Changes

Lancer un build

Configurer

Supprimer Pipeline

Full Stage View

GitHub

Renommer

Pipeline Syntax

GitHub Hook Log

microservice2AO

ttt

modifier la description

Désactiver le projet

Stage View

Average stage times:
(Average full run time: ~12min 0s)

	Declarative: Tool Install	Git Clone	Build	Build2	Check Docker Compose	Docker Build	Docker Run
#4 janv. 20 21:59 No Changes	597ms	5s	1min 16s	51s	5s	52s	8min 42s
#3 janv. 20 21:53 No Changes	960ms	3s	48s	24s	2s	38s	2min 18s

Historique des builds **tendance**

Pour les utilisateurs de Windows,

Supprimez **sh** et utilisez **bat**,

supprimez **'cd /usr/local/bin/'** et utilisez plutôt : **bat 'docker-compose up'**

Intégration de SonarQube:

Configuration de SonarQube :

Intégration avec Jenkins : SonarQube est intégré à notre pipeline CI/CD via Jenkins. À chaque exécution du pipeline, les analyses statiques du code sont automatiquement déclenchées, fournissant ainsi des rapports détaillés sur la qualité du code.

Configuration des Règles : Nous avons personnalisé les règles de SonarQube pour refléter nos normes internes de codage. Les règles par défaut ont été ajustées pour s'aligner avec les meilleures pratiques et les standards spécifiques à notre projet.

Utilisation des Plugins : Nous avons configuré et exploité des plugins complémentaires de SonarQube pour couvrir un large éventail de langages de programmation et de technologies utilisés dans notre application. Cela garantit une analyse complète de tout le code source, qu'il s'agisse du Microservice Gateway, du Microservice Blog, ou d'autres composants.

Bénéfices pour la Qualité du Code :

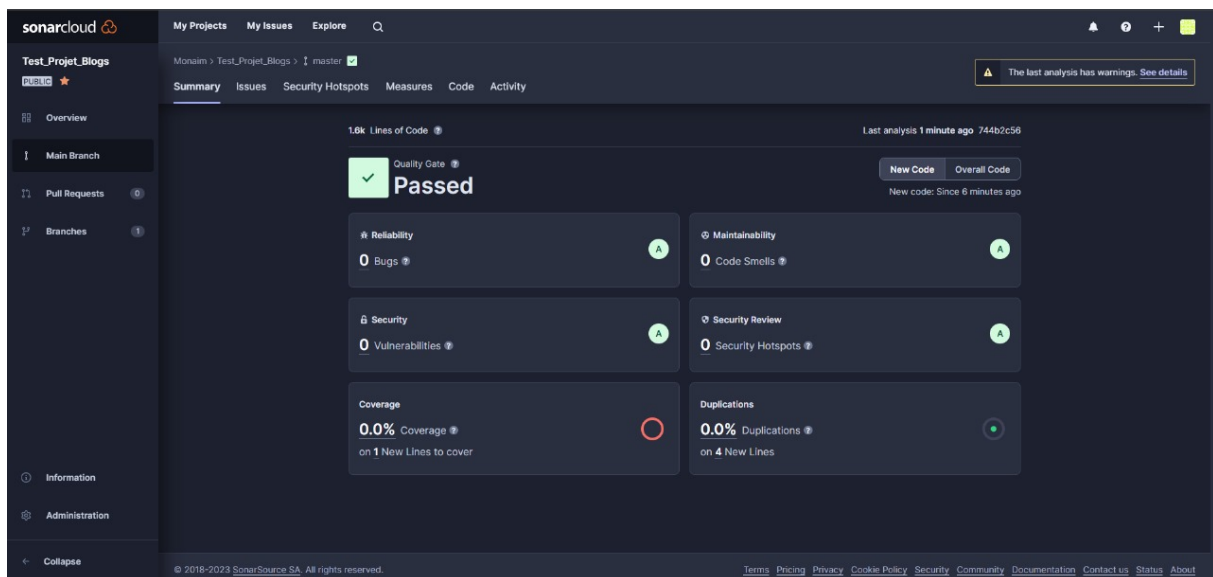
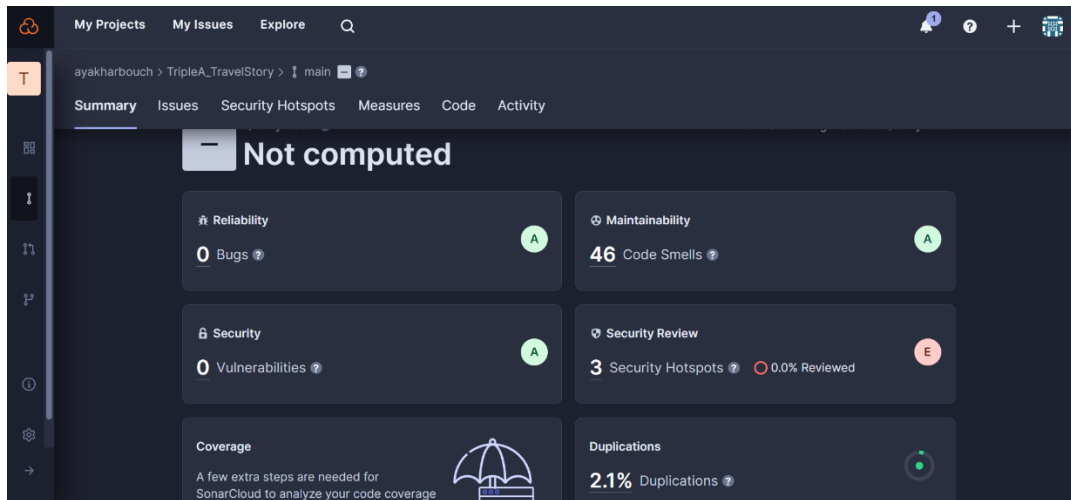
Détection Précoce des Problèmes : SonarQube identifie rapidement les problèmes potentiels de code, y compris les bugs, les vulnérabilités de sécurité, les odeurs de code et les pratiques de codage non conformes.

Analyse de la Dette Technique : SonarQube fournit des informations détaillées sur la dette technique, permettant aux développeurs de prioriser et d'adresser les zones du code nécessitant une amélioration.

Normes de Codage Cohérentes : Grâce à la configuration personnalisée des règles, SonarQube assure une conformité constante avec nos normes de codage internes, facilitant ainsi la collaboration et le partage de bonnes pratiques au sein de l'équipe.

Tableau de Bord Visuel : Les tableaux de bord de SonarQube offrent une vue visuelle de la qualité du code, permettant une surveillance continue et une évaluation transparente de la santé du projet.

Amélioration Continue : Les rapports de SonarQube servent de base pour des révisions de code régulières, favorisant une culture d'amélioration continue et d'excellence en matière de développement.



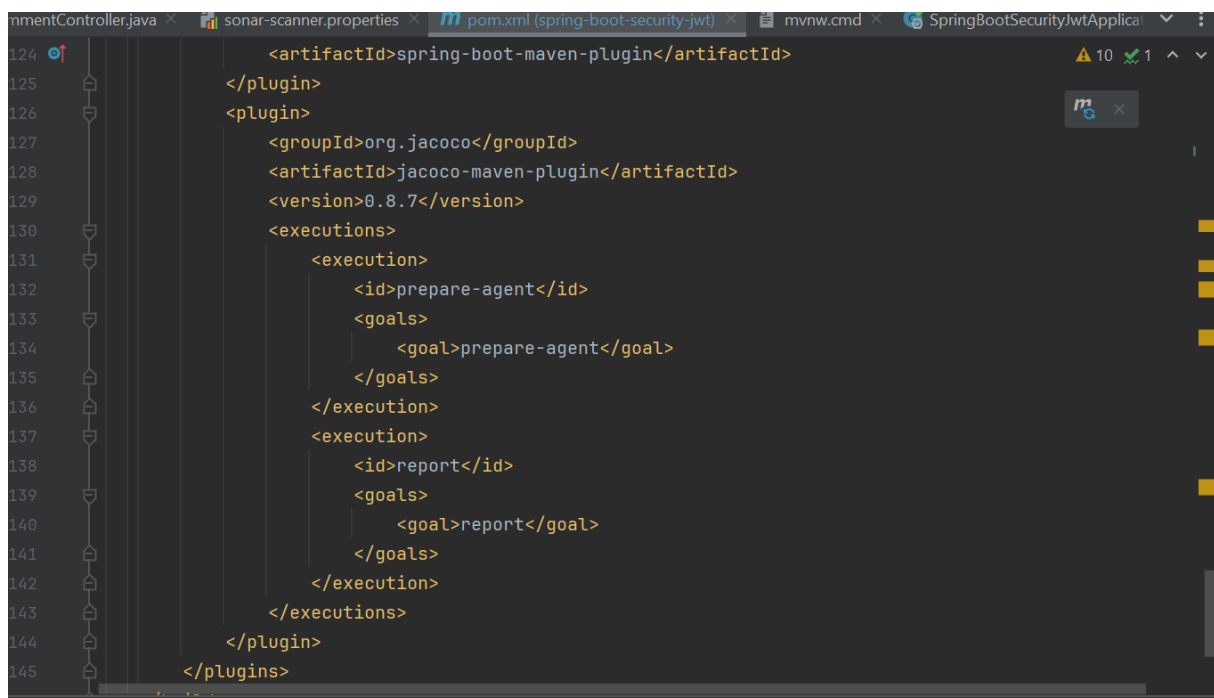
For the coverage we have to add in pom.xml

```
<plugin>
  <groupId>org.sonarsource.scanner.maven</groupId>
  <artifactId>sonar-maven-plugin</artifactId>
  <version>3.8.0.2131</version>
</plugin>
```

```
<dependency>
  <groupId>org.sonarsource.scanner.maven</groupId>
  <artifactId>sonar-maven-plugin</artifactId>
  <version>3.6.0.1398</version>
```

```
</dependency>
```

```
<plugin>  
  <groupId>org.jacoco</groupId>  
  <artifactId>jacoco-maven-plugin</artifactId>  
  <version>0.8.7</version>  
  <executions>  
    <execution>  
      <id>prepare-agent</id>  
      <goals>  
        <goal>prepare-agent</goal>  
      </goals>  
    </execution>  
    <execution>  
      <id>report</id>  
      <goals>  
        <goal>report</goal>  
      </goals>  
    </execution>  
  </executions>  
</plugin>
```



The screenshot shows an IDE window with the following tabs: `mentController.java`, `sonar-scanner.properties`, `pom.xml (spring-boot-security-jwt)`, `mvnw.cmd`, and `SpringBootSecurityJwtApplica...`. The `pom.xml` file is open, showing the following XML content:

```
124 <artifactId>spring-boot-maven-plugin</artifactId>  
125 </plugin>  
126 <plugin>  
127   <groupId>org.jacoco</groupId>  
128   <artifactId>jacoco-maven-plugin</artifactId>  
129   <version>0.8.7</version>  
130   <executions>  
131     <execution>  
132       <id>prepare-agent</id>  
133       <goals>  
134         <goal>prepare-agent</goal>  
135       </goals>  
136     </execution>  
137     <execution>  
138       <id>report</id>  
139       <goals>  
140         <goal>report</goal>  
141       </goals>  
142     </execution>  
143   </executions>  
144 </plugin>  
145 </plugins>
```

On the right side of the IDE, there is a Maven icon and a status bar showing 10 warnings and 1 error.

Conclusion:

Triple A Travel Story a atteint des jalons significatifs, adoptant avec succès l'architecture microservices, la conteneurisation avec Docker, la CI/CD avec Jenkins, l'intégration de SonarQube, et l'utilisation de Consul pour la découverte de services.

Les perspectives futures incluent le développement d'une application mobile dédiée, des améliorations de l'expérience utilisateur, l'optimisation des performances, l'élargissement des fonctionnalités, et la poursuite de l'évolutivité et de la résilience de l'architecture. Triple A Travel Story continue son engagement envers l'innovation et la satisfaction des utilisateurs.