**Rapport TNB**

**Réalisé par : Anass MOAYA**

**Zaidane AbdelIlah**

**Soulaiman Ncir**

**1. Introduction**

**Aperçu du Projet**

Le projet de microservices entrepris représente une initiative stratégique visant à moderniser et à optimiser notre infrastructure logicielle. L'objectif principal de ce projet est de développer une suite de services indépendants mais interconnectés, chacun assurant une fonctionnalité spécifique au sein de notre écosystème d'applications.

Ce projet a été initié dans le contexte d'une nécessité croissante de flexibilité, de scalabilité et d'efficacité dans nos opérations informatiques. Confrontés à un environnement commercial en rapide évolution et à des demandes de plus en plus complexes de la part de nos clients, nous avons reconnu la nécessité de migrer vers une architecture qui pourrait non seulement répondre à ces exigences actuelles mais aussi s'adapter aisément aux besoins futurs. L'architecture microservices a été choisie en raison de sa capacité à offrir ces avantages, permettant ainsi une évolution et une maintenance plus aisées des applications, une meilleure répartition des ressources et une accélération du processus de développement.

Le projet englobe le développement de plusieurs microservices, chacun étant conçu pour fonctionner de manière autonome tout en participant à un système cohérent et unifié. Cette approche modulaire nous permet de déployer, de mettre à jour, et de maintenir chaque service de manière indépendante, réduisant ainsi les risques et les coûts associés à la gestion d'une application monolithique traditionnelle.

**Importance de l’Architecture Microservice**

L'adoption de l'architecture microservices pour notre projet s'inscrit dans une démarche stratégique visant à optimiser notre efficacité opérationnelle et à renforcer notre compétitivité sur le marché. Cette architecture présente plusieurs avantages clés qui la rendent particulièrement adaptée aux exigences actuelles du développement logiciel.

**Scalabilité**: L'une des caractéristiques les plus notables de l'architecture microservices est sa capacité à faciliter la scalabilité. Contrairement aux architectures monolithiques, où la scalabilité nécessite souvent de redimensionner l'ensemble de l'application, les microservices permettent une scalabilité granulaire. Cela signifie que nous pouvons augmenter ou diminuer les ressources allouées à des services spécifiques en fonction de la demande, sans affecter le reste du système. Cette approche ciblée optimise l'utilisation des ressources et améliore l'efficacité opérationnelle.

Flexibilité et Agilité: Les microservices offrent une flexibilité accrue dans le développement et la maintenance des applications. Chaque service étant développé et déployé indépendamment, les équipes peuvent adopter des cycles de développement plus courts et des mises à jour plus fréquentes. Cette agilité permet une réponse rapide aux besoins changeants du marché et aux demandes des clients, favorisant une innovation continue.

**Facilité de Maintenance et de Déploiement**: La maintenance et le déploiement des applications sont simplifiés avec une architecture microservices. Les services indépendants peuvent être mis à jour, testés et déployés sans risque de perturber les autres composants du système. Cette indépendance réduit les risques associés aux mises à jour et aux modifications, et facilite la gestion du cycle de vie des applications.

**Résilience et Fiabilité**: En cas de défaillance d'un microservice, l'impact sur l'ensemble du système est généralement limité. Cette isolation permet d'assurer une plus grande résilience et une meilleure continuité des opérations, car les autres services peuvent continuer à fonctionner normalement.

**Adaptabilité Technologique**: Les microservices permettent l'utilisation de différentes technologies et langages de programmation au sein du même projet. Cette hétérogénéité technologique favorise l'utilisation des outils les plus adaptés à chaque tâche, améliorant ainsi l'efficacité et la qualité du développement.

**2. Architecture Microservices**

**Architecture**

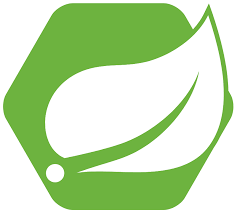


Figure 1 : logo de Eureka Server

Architecture

L'architecture de notre projet de microservices est conçue pour optimiser la modularité, l'indépendance et la communication efficace entre les différents services. Elle se compose de plusieurs services indépendants, chacun dédié à une fonctionnalité spécifique, qui interagissent entre eux pour former une application complète et cohérente.

La structure globale repose sur un modèle de services distribués, où chaque microservice est développé, déployé et maintenu de manière indépendante. Cette approche permet non seulement de répartir les charges de travail, mais également de faciliter la mise à l'échelle et la maintenance des services individuels.

Pour assurer une communication fluide et fiable entre les services, nous avons mis en place des API REST, qui offrent une interface standardisée pour l'échange de données. Ces API permettent aux services de communiquer entre eux de manière asynchrone, garantissant ainsi une performance et une disponibilité élevées.

La cohérence des données et la gestion des transactions entre les services sont gérées grâce à des mécanismes tels que les événements et les messages, permettant une synchronisation efficace tout en maintenant l'autonomie des différents composants.

**Description des Services**

Notre projet comporte cinq microservices principaux, chacun jouant un rôle crucial dans l'ensemble du système. Voici une description de la conception de chaque microservice :

**Microservice Tax Calculation**

Fonction: Ce service est chargé de calculer la taxe TNB (Taxe sur la Valeur Ajoutée). Il reçoit les données nécessaires et applique la logique de calcul pour déterminer le montant de la taxe.

Technologie/Approche: Le service utilise des algorithmes de calcul spécifiques pour assurer une évaluation précise et efficace de la taxe. Il peut interagir avec d'autres services pour obtenir les informations nécessaires.

**Microservice Terrain**

Fonction: Ce service gère les informations relatives aux terrains. Il permet de stocker, de récupérer et de mettre à jour les détails concernant divers terrains.

Technologie/Approche: Le service utilise une base de données pour stocker les informations sur les terrains. Il offre une API pour les opérations CRUD (Create, Read, Update, Delete) sur ces données.

**Microservice Tax Rate**

Fonction: Ce service est responsable de la gestion des taux de taxe. Il stocke et fournit des informations sur les différents taux de taxe applicables.

Technologie/Approche: Comme le service Terrain, il utilise une base de données pour conserver les taux de taxe et expose une API pour accéder à ces informations.

**Eureka Service Registry**

Fonction: Ce service agit comme un registre de service pour tous les microservices du projet. Il permet aux services de s'enregistrer et de découvrir les autres services disponibles dans l'architecture.

Technologie/Approche: Eureka est utilisé pour faciliter la découverte de service, ce qui est essentiel pour la communication efficace entre microservices dans une architecture distribuée.

**Gateway Service**

Fonction: Le Gateway Service agit comme un point d'entrée unique pour toutes les requêtes entrantes. Il route les requêtes vers les services appropriés.

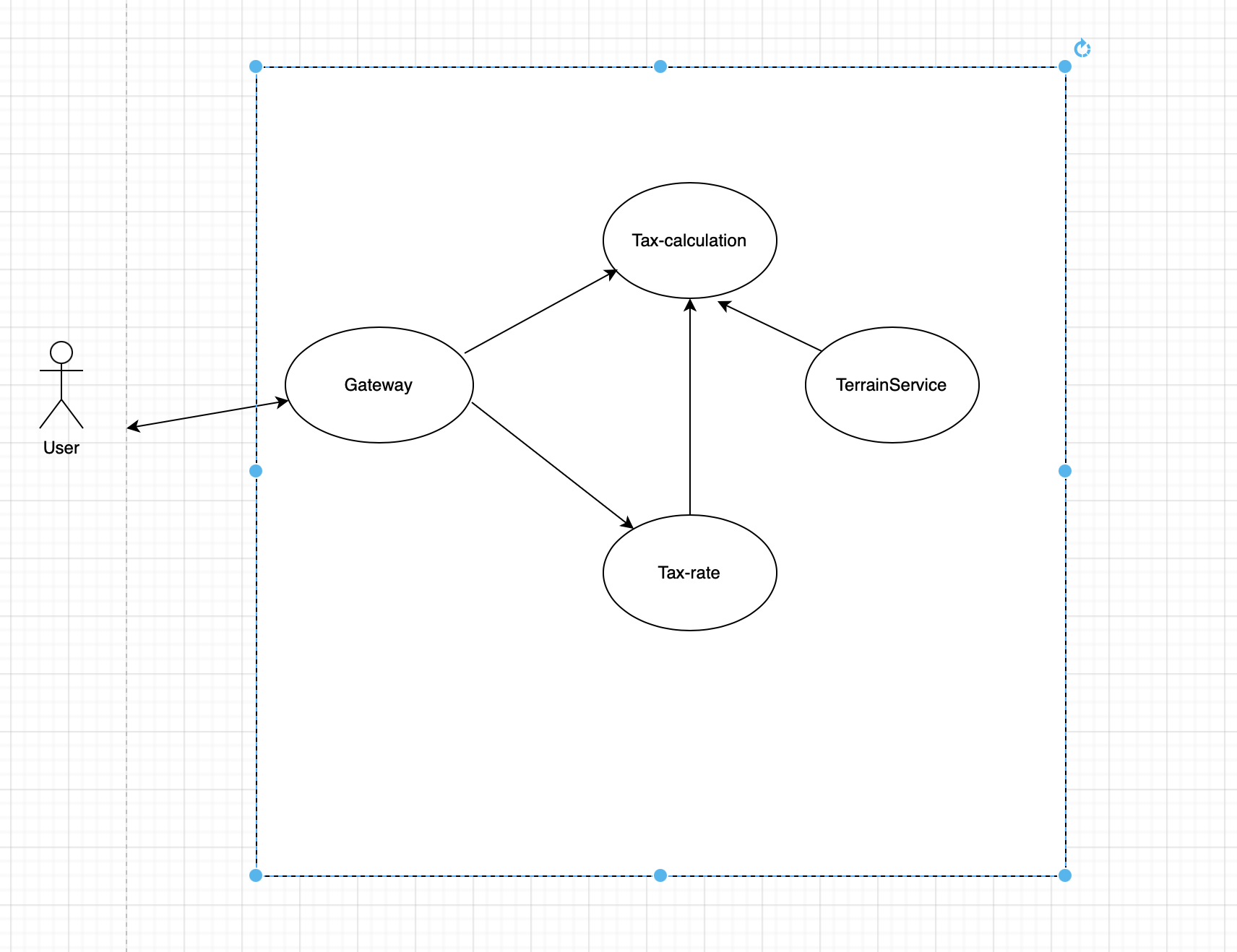
Technologie/Approche: Ce service utilise des règles de routage et peut implémenter des fonctionnalités telles que l'authentification, la journalisation des requêtes, et la gestion des erreurs.

**Communication entre les Microservices**

La communication entre les microservices Tax Calculation, Terrain, et Tax Rate est facilitée par OpenFeign, un client Feign déclaratif qui simplifie l'écriture de clients HTTP. OpenFeign est utilisé pour créer des interfaces facilement et communiquer entre services en utilisant des appels REST. Cela permet une intégration étroite et efficace entre les services tout en maintenant leur couplage lâche et leur indépendance.

Cette conception de microservices assure une séparation claire des responsabilités, une flexibilité dans le développement et la maintenance, et une communication fluide entre les services. Chaque service peut évoluer indépendamment, ce qui renforce la scalabilité et la résilience de l'ensemble du système.

**3. Conception des Microservices**

****

**4. Conteneurisation avec Docker**

Une image contenant Graphique, dessin humoristique, clipart, conception

Description générée automatiquement

Figure 2 : Docker logo

Conteneurisation avec Docker

Dans notre projet de microservices, nous avons adopté Docker pour la conteneurisation des différents services. Cette approche offre plusieurs avantages significatifs qui améliorent le déploiement, la scalabilité et la gestion de l'environnement de notre application.

Implémentation

**Création de Dockerfiles**: Pour chaque microservice (Tax Calculation, Terrain, Tax Rate, Eureka Service Registry, et Gateway Service), un Dockerfile a été créé. Ce fichier détaille les instructions nécessaires pour construire l'image Docker du service, incluant la base de l'image, les dépendances, les variables d'environnement, et les commandes pour exécuter le service.

**Construction des Images Docker**: Les images Docker pour chaque service sont construites à partir de leurs Dockerfiles respectifs. Cela crée une image standardisée et portable pour chaque service, qui peut être déployée de manière cohérente dans différents environnements.

**Docker Compose pour la Gestion des Services**: Pour faciliter le déploiement et la gestion des services, Docker Compose est utilisé. Docker Compose permet de définir et de gérer plusieurs conteneurs comme un ensemble de services interconnectés. Un fichier docker-compose.yml est utilisé pour configurer ces services, leur réseau, et leurs dépendances.

**Avantages**

**Consistance et Isolation**: Docker assure que chaque service fonctionne dans un environnement isolé et cohérent, éliminant les problèmes de « ça marche sur ma machine ». Cette isolation aide également à prévenir les conflits entre les services.

**Déploiement Rapide et Facile**: Avec Docker, le déploiement des services devient rapide et facile. Les conteneurs Docker peuvent être déployés sur n'importe quel système prenant en charge Docker, ce qui simplifie le processus de déploiement dans divers environnements, qu'ils soient de développement, de test ou de production.

**Scalabilité:** Docker facilite la scalabilité des services. Les conteneurs peuvent être rapidement démarrés ou arrêtés, permettant une montée en charge ou une réduction des ressources en fonction de la demande.

**Utilisation Efficace des Ressources**: Les conteneurs Docker utilisent les ressources de manière plus efficace que les machines virtuelles traditionnelles, car ils partagent le même noyau OS et n'ont pas besoin d'un OS invité complet.

**Intégration avec les Pipelines CI/CD**: Docker s'intègre bien avec les systèmes CI/CD comme Jenkins, facilitant l'automatisation du processus de construction, de test et de déploiement des services.

En conclusion, l'utilisation de Docker dans notre projet de microservices a considérablement amélioré la gestion, le déploiement et la scalabilité de nos services, tout en offrant une plateforme cohérente pour leur développement et leur exécution.

**5. CI/CD avec Jenkins**

Une image contenant dessin humoristique, Dessin animé, illustration, clipart

Description générée automatiquement

Figure 3 : Jenkins logo

Notre projet utilise Jenkins pour mettre en œuvre un pipeline de CI/CD (Continuous Integration/Continuous Deployment), un élément clé pour assurer une intégration et un déploiement rapides et fiables de nos microservices. Voici comment nous avons configuré et utilisé Jenkins dans notre projet :

**Processus**

**Intégration Continue (CI)** : À chaque fois qu'un changement est poussé dans notre dépôt de code (par exemple, sur GitHub), Jenkins déclenche automatiquement un processus d'intégration continue. Ce processus comprend la récupération du code source le plus récent, l'exécution des tests unitaires et d'intégration, et la construction des images Docker pour chaque microservice.

**Déploiement Continu (CD)** : Une fois que le processus d'intégration continue réussit sans erreurs, le pipeline de déploiement continu est déclenché. Cela inclut le déploiement des images Docker construites dans l'environnement de test, suivi par le déploiement dans l'environnement de production en cas de succès dans l'environnement de test.

**Configuration du Pipeline Jenkins**

**Pipeline as Code** : Nous utilisons la fonctionnalité "Pipeline as Code" de Jenkins, où le pipeline est défini dans un fichier Jenkinsfile stocké dans le dépôt de code. Ce fichier définit les différentes étapes du pipeline, y compris la construction, les tests, et le déploiement.

**Étapes du Pipeline :**

pipeline {

agent any

environment {

PATH = "/usr/local/bin:$PATH"

SONARQUBE\_HOST = "http://localhost:9000"

SONARQUBE\_TOKEN = "sqp\_1aa46522226fa8e5f5378b87d28685c1068aea7b"

}

stages {

stage('Clone Repository') {

steps {

git branch: 'main', url: 'https://github.com/anasmoaya/TNB-calculation---Backend---microservices-'

}

}

stage('SonarQube Analysis') {

steps {

script {

sh 'chmod +x TerrainService/gradlew'

sh "cd TerrainService && ./gradlew -Dsonar.projectKey=tnb-terrain-service -Dsonar.projectName='tnb-terrain-service' -Dsonar.host.url=${SONARQUBE\_HOST} -Dsonar.login=${SONARQUBE\_TOKEN}"

}

}

}

stage('Build Gateway Service') {

steps {

script {

sh '/usr/local/bin/docker build -t gateway-service-tnb ./GatewaySerivce'

}

}

}

stage('Build Registry Service') {

steps {

script {

sh '/usr/local/bin/docker build -t registry-service-tnb ./RegistryService'

}

}

}

stage('Build Microservices') {

steps {

script {

sh '/usr/local/bin/docker build -t tax-calculation-service ./TaxCalculationService'

sh '/usr/local/bin/docker build -t tax-rate-service ./TaxRateService'

sh '/usr/local/bin/docker build -t terrain-service-tnb ./TerrainService'

}

}

}

stage('Run Docker Compose') {

steps {

sh '/usr/local/bin/docker-compose up -d'

}

}

}

}

Clone Repository

Cette étape clone le dernier code du dépôt spécifié sur la branche principale. Le dépôt utilisé est https://github.com/anasmoaya/TNB-calculation---Backend---microservices-, et la branche ciblée est main.

**SonarQube Analysis**

Durant cette étape, une analyse SonarQube est effectuée pour le service TerrainService. Les commandes exécutées sont les suivantes :

Attribution des permissions d'exécution au script gradlew dans le répertoire TerrainService.

Exécution de ./gradlew avec les paramètres nécessaires pour réaliser l'analyse SonarQube, en utilisant l'URL et le token de SonarQube spécifiés dans les variables d'environnement.

**Build Gateway Service**

Cette étape construit l'image Docker pour le Gateway Service. Le script utilisé construit une image nommée gateway-service-tnb à partir du répertoire ./GatewaySerivce.

**Build Registry Service**

Cette étape s'occupe de construire l'image Docker pour le Registry Service. Le script construit une image nommée registry-service-tnb à partir du répertoire ./RegistryService.

**Build Microservices**

Cette étape construit les images Docker pour les microservices restants :

Construction de l'image tax-calculation-service à partir du répertoire ./TaxCalculationService.

Construction de l'image tax-rate-service à partir du répertoire ./TaxRateService.

Construction de l'image terrain-service-tnb à partir du répertoire ./TerrainService.

**Run Docker Compose**

La dernière étape exécute docker-compose up -d pour démarrer l'ensemble des services définis dans le fichier docker-compose.yml. Cette commande est exécutée en mode détaché (-d), ce qui permet aux conteneurs de s'exécuter en arrière-plan.

Chaque étape du pipeline joue un rôle essentiel dans le processus d'intégration et de déploiement continu, garantissant que les dernières modifications du code sont automatiquement intégrées, testées et déployées.

**6. Déploiement Automatique et Gestion des Services**

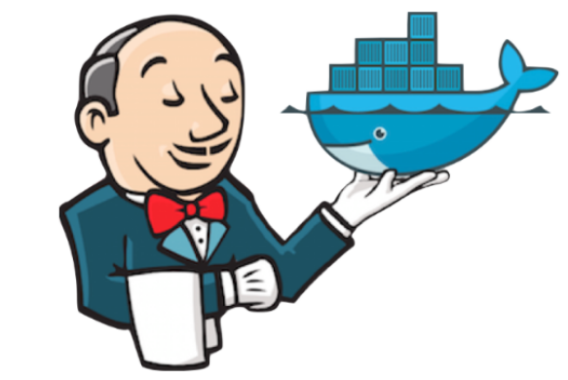


Figure 4 : Jenkins + Docker

Le déploiement automatique dans notre projet est orchestré grâce à Jenkins et Docker. Jenkins gère le CI/CD, facilitant des déploiements rapides et efficaces. Docker assure la conteneurisation, garantissant la cohérence des environnements de déploiement. Eureka, notre service de découverte, gère dynamiquement les instances de service, permettant leur localisation et interaction fluide. Cette synergie entre Jenkins, Docker et Eureka crée un écosystème robuste pour nos microservices, assurant déploiement rapide, fiabilité et adaptabilité face aux exigences évolutives de l'application.

**7. Intégration de SonarQube**



SonarQube, intégré dans notre environnement Docker, joue un rôle essentiel dans notre pipeline CI/CD. Il analyse en continu le code source pour détecter les vulnérabilités, les bugs et les mauvaises pratiques de codage. Cette intégration assure non seulement la qualité et la sécurité de notre code, mais contribue également à améliorer constamment les standards de codage et la maintenabilité du projet. SonarQube fournit ainsi une visibilité claire sur la santé du code et guide l'équipe vers des améliorations ciblées.

**8. Conclusion**

Notre projet a brillamment établi une architecture microservices à la fois robuste, flexible et évolutive. L'intégration harmonieuse d'outils de développement, de déploiement et d'assurance qualité, notamment Jenkins, Docker et SonarQube, a joué un rôle crucial dans ce succès. Cette combinaison a permis de créer un environnement où la continuité, l'efficacité et la qualité du développement sont maintenues à un niveau élevé. L'adaptabilité de l'architecture assure sa pertinence face aux évolutions technologiques futures, posant ainsi les bases d'une croissance soutenue et d'une amélioration continue. Ce projet est un exemple remarquable de la manière dont la technologie moderne peut être utilisée pour construire des systèmes complexes, mais gérables, qui répondent aux exigences dynamiques du monde des affaires actuel.