

# PROJET (2024)

« 5<sup>ème</sup> année informatique et réseaux »

## Conception de Système de taxes TNB au Maroc

### Ingénierie Informatique et Réseaux

**Réalisé par :**

CHBALY Adil

KHARDALI Ikram

OHLALE Badr

**Encadré par :**

Dr LACHGAR Mohamed

**Année Universitaire :**

2023 - 2024

## Dédicace

Nous dédions ce travail à Dieu le plus puissant, source infinie de grâce et de bienveillance, dont la lumière a éclairé chaque étape de notre parcours. C'est avec humilité que nous exprimons notre gratitude pour Sa guidance et Sa présence qui ont été nos forces motrices dans cette quête académique.

Ensuite, nous tournons nos pensées et nos paroles vers nos chers parents, dont l'amour inconditionnel et les sacrifices ont été les fondations solides sur lesquelles nous avons bâti notre éducation. Leur soutien indéfectible et leur dévouement sans limites sont des témoignages vivants de leur amour. Que Dieu les enveloppe de santé et de longévité, car ils méritent toute notre reconnaissance éternelle.

À notre grand frère, nous dédions une part spéciale de cette réussite. Son soutien infaillible a été une inspiration constante. Que chaque jour de sa vie soit empli de bonheur et d'épanouissement. À notre jeune frère, nous espérons que notre succès lui insuffle le courage nécessaire pour ses propres aspirations académiques.

Notre sœur chère mérite également notre reconnaissance pour son soutien indéfectible, son encouragement sans faille, et sa contribution significative à notre réussite. À tous nos amis et proches, dont la liste est bien plus longue que ces mots, votre amitié et votre soutien ont été des trésors inestimables. Nous vous souhaitons le bonheur, la joie et la réussite dans toutes vos entreprises.

Enfin, notre gratitude va à toutes les personnes qui nous ont encouragés, guidés et accompagnés dans cette entreprise. Puissent nos paroles refléter la profondeur de notre amour et l'étendue de notre gratitude envers chacun d'entre vous.

## Remerciement

Au terme de ce projet, nous ressentons le besoin sincère d'exprimer notre profonde gratitude envers ceux qui ont été des piliers inestimables de notre réussite académique. En tête de cette reconnaissance, nos remerciements vont à Dieu, le Tout-Puissant, pour nous avoir accordé le courage, la patience et la santé indispensables pour cheminer à travers ce parcours éducatif semé de défis et d'apprentissages. Sa guidance nous a été précieuse à chaque étape de notre formation, nous permettant de surmonter les obstacles avec détermination.

Nous souhaitons également exprimer notre sincère reconnaissance à notre encadrant estimé, Monsieur **LACHGAR Mohamed**. Son engagement, ses précieux conseils et son soutien indéfectible ont été des pierres angulaires dans la concrétisation de ce projet. Sa disponibilité, ses orientations éclairées et son expertise ont été d'une valeur inestimable, nous guidant avec assurance vers la réussite.

De même, nous sommes reconnaissants envers le jury, dont la présence, l'engagement et les contributions ont été inestimables pour l'évaluation et l'enrichissement de ce travail. Leurs retours constructifs et leur investissement dans notre projet ont été d'une aide précieuse pour l'amélioration de notre travail.

Enfin, l'EMSI (École Marocaine des Sciences de l'Ingénieur) a joué un rôle fondamental dans notre parcours académique. Son environnement propice à l'apprentissage dans le domaine des sciences de l'ingénieur et des Sciences de données a été le cadre essentiel qui a nourri notre progression et notre épanouissement.

## Résumé

Le projet sur la Taxe sur les Terrains Non Bâties (TNB) au Maroc constitue une initiative complète visant à moderniser et simplifier la gestion fiscale associée aux terrains non construits. La TNB, une taxe annuelle, est tributaire de la classification des terrains en différentes catégories, telles que villas, maisons, appartements, etc. Chaque catégorie est soumise à un taux spécifique, initialement fixé à 100 dirhams par mètre carré, mais susceptible de varier au fil du temps en fonction de divers critères.

Les principales exigences du projet incluent la mise en place d'un système de classification des terrains par catégorie, la gestion des variations des taux associés à chaque catégorie au fil des années, et la possibilité de recherche des terrains d'un redevable spécifique par le biais de sa Carte d'Identité Nationale (CIN). Le système doit également fournir un historique détaillé des taxes annuelles associées à ces terrains pour permettre une gestion transparente et informée.

Une autre composante clé du projet est la mise en place d'une fonction de recherche permettant de déterminer le taux applicable à une catégorie de terrain donnée. Cette fonctionnalité s'inscrit dans une logique de transparence et d'accessibilité des informations fiscales.

Le processus de calcul de la taxe TNB est automatisé et repose sur l'entrée de données spécifiques, à savoir le numéro de CIN du redevable, l'année fiscale concernée, et les détails du terrain en question. Le résultat produit par le système est le montant total de la taxe TNB à payer, calculé en fonction des taux actuels et de la superficie du terrain, à condition que la taxe ne soit pas déjà acquittée pour l'année en question.

En somme, l'objectif ultime de ce projet est de fournir une plateforme intégrée, intuitive et efficace pour la gestion de la TNB au Maroc. Il vise à faciliter les processus administratifs liés à cette taxe, à garantir une collecte juste et transparente, tout en offrant aux contribuables et aux autorités fiscales un accès facile aux informations pertinentes pour une prise de décision éclairée.

## Abstract

The project on the Tax on Unbuilt Land (TNB) in Morocco is a comprehensive initiative aimed at modernizing and simplifying the fiscal management associated with undeveloped lands. The TNB, an annual tax, is contingent upon the classification of lands into different categories, such as villas, houses, apartments, etc. Each category is subject to a specific rate, initially set at 100 dirhams per square meter but subject to variation over time based on various criteria.

The main requirements of the project include the implementation of a system for classifying lands by category, managing variations in rates associated with each category over the years, and the ability to search for lands owned by a specific taxpayer through their National Identity Card (CIN). The system must also provide a detailed history of the annual taxes associated with these lands to facilitate transparent and informed management.

Another key component of the project is the implementation of a search function to determine the applicable rate for a given land category. This feature aligns with a logic of transparency and accessibility of tax information.

The process of calculating the TNB tax is automated and relies on specific data inputs, namely the taxpayer's CIN, the relevant fiscal year, and details of the land in question. The result produced by the system is the total amount of TNB tax to be paid, calculated based on current rates and the land's area, provided that the tax has not already been paid for the given year.

In summary, the ultimate goal of this project is to provide an integrated, intuitive, and efficient platform for TNB management in Morocco. It aims to streamline administrative processes related to this tax, ensure fair and transparent collection, while offering taxpayers and tax authorities easy access to relevant information for informed decision-making.

## Table de matière

Dédicace .....	2
Remerciement .....	3
Résumé.....	4
Abstract .....	5
Table de matière .....	6
Liste des figures.....	8
Introduction générale.....	10
Chapitre 1 : Développement de l'architecture des microservices.....	11
Introduction :.....	11
Architecture du projet :.....	11
1. Eureka-Service :.....	12
2. API Gateway :.....	12
3. Keycloak :.....	13
3.1. Etapes à suivre :.....	14
4. Description globale des microservices utilisés :.....	18
Conclusion :.....	19
Chapitre 2 : conteneurisation de l'application.....	21
Introduction :.....	21
1. Docker compose keycloak :.....	21
2. Docker compose du projet global :.....	23
Conclusion :.....	26
Chapitre 3 : Automatisation et Intégration Continue avec Jenkins .....	27
Introduction :.....	27
Définition du Jenkins : .....	27
Pourquoi utiliser Jenkins ? :.....	27
Conclusion :.....	29
Chapitre 4 : SonarQube - Outil de Qualité du Code en Continu .....	30
Introduction :.....	30
1. C'est quoi SonarQube ?.....	30
2. Architecture de SonarQube :.....	30
3. Intégration harmonieuse : Optimiser les Déclenchements Automatiques avec les Webhooks GitHub et Jenkins :.....	33

Conclusion :	40
Chapitre 5 : Technologies utilisées et réalisation	41
Introduction :	41
1. Technologies utilisées :	41
1.1. Keycloak :	41
1.2 Spring Boot :	41
1.3 Angular :	42
1.4. Apache Maven :	42
1.5. Resilience4j :	43
1.6. PostgreSQL :	43
2. Réalisation du projet :	44
Conclusion :	45
Conclusion Générale	46

## Liste des figures

Figure 1 : Architecture globale du projet .....	11
Figure 2 : Interface de creation de realm.....	14
Figure 3 : Interface de création du client .....	14
Figure 4 : Access settings and web cors .....	15
Figure 5 : Creation du role USER .....	15
Figure 6 : Creation du role ADMIN .....	15
Figure 7 : Creation du compte ADMIN .....	16
Figure 8 : Assigner le role ADMIN au compte ADMIN .....	16
Figure 9 : Création du compte USER .....	17
Figure 10 : Assigner le role USER au compte USER .....	17
Figure 11 : Configuration du login et registry .....	18
Figure 12 : Contenu du fichier keycloak-docker.yml.....	22
Figure 13 : Contenu du fichier global docker-compose.yml (partie1) .....	23
Figure 14 : Contenu du fichier global docker-compose.yml (partie2) .....	23
Figure 15 : Contenu du fichier global docker-compose.yml (partie3) .....	24
Figure 16 : Contenu du fichier global docker-compose.yml (partie4) .....	24
Figure 17 : Contenu du fichier global docker-compose.yml (partie5) .....	25
Figure 18 : Contenu du fichier global docker-compose.yml (partie6) .....	25
Figure 19 : Affichage après le lancement de notre pipeline .....	28
Figure 20 : Notre docker-compose final.....	29
Figure 21 Création de notre CI/CD Pipeline avec Jenkins .....	31
Figure 22 Screen de notre interface SonarQube et SonarQube Analysis (Partie 1) .....	32
Figure 23 Screen de notre interface SonarQube et SonarQube Analysis (Partie 2) .....	32
Figure 24 : Et Voilà Notre Pipeline est bien fonctionnel .....	33
Figure 25 : Création de notre Webhook avec GitHub .....	33
Figure 26 : Le Webhook marche bien avec notre serveur Jenkins .....	34
Figure 27 : Ajout du lien GitHub dans le Pull Request System.....	34
Figure 28 : Pusher un changement .....	34
Figure 29 : Le CI/CD marche bien .....	35
Figure 30 : Contenu du JenkinsFile (partie 1).....	35
Figure 31 : Contenu du JenkinsFile (partie 2).....	36
Figure 32 : Contenu du JenkinsFile (partie 3).....	37
Figure 33 : Contenu du JenkinsFile (partie 4).....	38
Figure 34 : Contenu du JenkinsFile (partie 5).....	39
Figure 35 : Contenu du JenkinsFile (partie 6).....	40
Figure 36 : Logo de Keycloak .....	41
Figure 37 : Logo Spring boot .....	42
Figure 38 : Logo Angular.....	42
Figure 39 : Logo de Apache Maven .....	43
Figure 40 : Logo Resilience4j .....	43
Figure 41 : Logo PostgreSQL.....	43



Figure 42 : Interface de catégorie .....	44
Figure 43 : Interface de taxes.....	44
Figure 44 : Interface de terrains.....	44

## Introduction générale

Le projet sur la Taxe sur les Terrains Non Bâties (TNB) au Maroc constitue une initiative visant à optimiser la gestion fiscale liée aux propriétés non construites. La TNB, un montant annuel, est calculée en fonction de la classification des terrains en différentes catégories telles que villas, maisons, appartements, etc. Chaque catégorie est assujettie à un taux spécifique, initialement fixé à 100 dirhams par mètre carré, mais sujet à des variations au fil du temps.

Les études d'exigences pour ce projet incluent la classification des terrains par catégorie et l'évolution des taux associés à chaque catégorie au fil des années. Le projet vise également à permettre la recherche des terrains appartenant à un redevable spécifique en utilisant sa Carte d'Identité Nationale (CIN) et à fournir un historique des taxes annuelles associées à ces terrains.

Une autre fonctionnalité clé consiste à permettre la recherche du taux applicable à une catégorie de terrain donnée. Une fois ces données recueillies, le système doit être en mesure de calculer la taxe TNB pour un terrain particulier en multipliant le taux de terrain par la surface du terrain, à condition que la taxe n'ait pas déjà été réglée pour l'année en question.

L'objectif global du projet est de fournir une plateforme efficiente et transparente, facilitant la gestion et le calcul de la taxe TNB au Maroc. L'entrée du système serait constituée du numéro de CIN du redevable, de l'année fiscale concernée et des détails spécifiques du terrain. Le résultat attendu serait le montant total de la taxe TNB à payer, calculé en fonction des taux en vigueur et de la superficie du terrain, si cette taxe n'a pas déjà été acquittée pour l'année en question.

# Chapitre 1 : Développement de l'architecture des microservices

## Introduction :

Le projet sur la Taxe sur les Terrains Non Bâties (TNB) au Maroc vise à optimiser la gestion fiscale des terrains non construits. La TNB, une taxe annuelle, dépend de la classification des terrains en catégories telles que villas, maisons, etc., avec des taux spécifiques, initialement fixés à 100 dirhams par mètre carré mais sujets à des variations. Les études d'exigences comprennent la classification des terrains et l'évolution des taux par catégorie au fil des ans. Le projet permet la recherche des terrains d'un redevable par sa Carte d'Identité Nationale (CIN) et fournit un historique des taxes annuelles. Une fonctionnalité clé est la recherche du taux par catégorie. Le système calcule la taxe TNB en fonction du taux et de la surface, à condition qu'elle ne soit pas déjà payée. L'objectif global est de fournir une plateforme efficace, transparente, simplifiant la gestion et le calcul de la taxe TNB. Les données d'entrée incluent le CIN du redevable, l'année fiscale et les détails du terrain, et le résultat attendu est le montant total de la taxe TNB à payer.

## Architecture du projet :

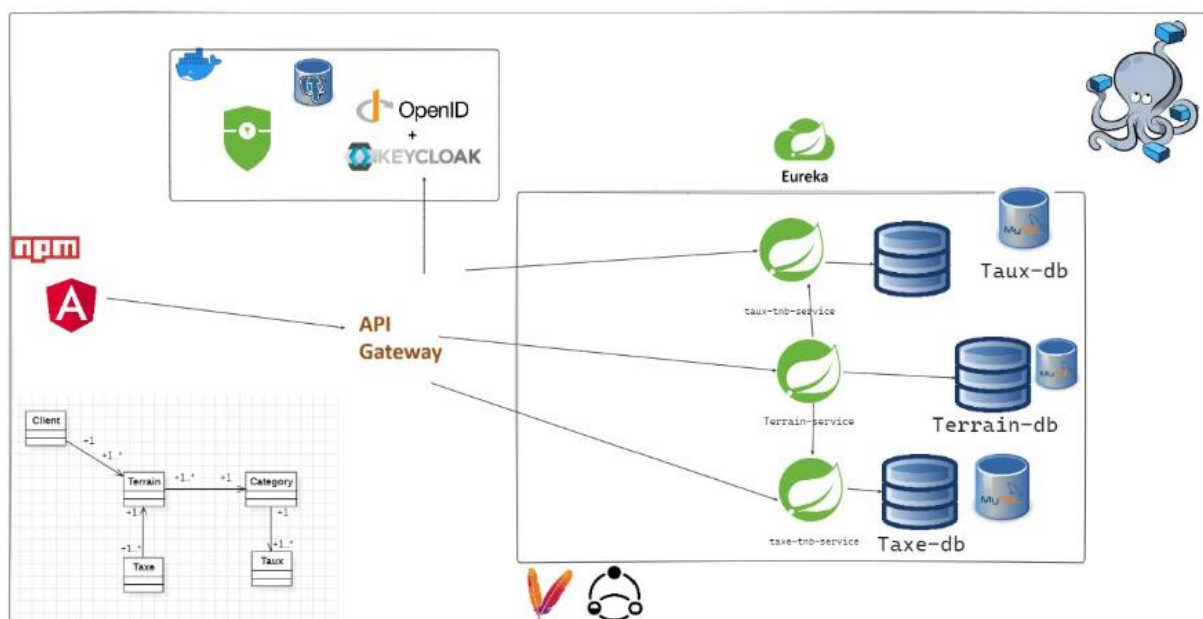


Figure 1 : Architecture globale du projet

Ce schéma d'architecture de notre système informatique ou du workflow de notre application microservices. Voici les détails clés :

- En haut à gauche, on trouve des icônes représentant des technologies de sécurité et d'authentification, y compris un cadenas, le logo de **Docker**, le logo d'**OpenID** et le logo de **Keycloak**, ce qui suggère que cette partie de l'architecture est responsable de la gestion de l'authentification et de la sécurité des utilisateurs.
- Juste en dessous, le logo d'**Angular** (un "A" rouge avec une bordure noire) et le logo de **npm** (Node Package Manager), qui est un gestionnaire de paquets pour le langage de programmation JavaScript. Cela indique l'utilisation d'**Angular** pour le développement frontal et **npm** pour la gestion des dépendances.

- Au centre de l'image se trouve l'indication "**API Gateway**", qui est un point d'entrée unique pour gérer les requêtes API dans une architecture microservices.

- Sur le côté droit de la figure, nous avons une série d'icônes vertes ressemblant à des spirales, représentant des services individuels dans l'architecture microservices. Chacun de ces services est associé à une base de données (db) : "**Taux-db**", "**Terrain-db**", et "**Taxe-db**". Ces bases de données sont représentées par des icônes de cylindres bleus avec le symbole des bases de données.

- En haut à droite, il y a le logo de "Eureka", qui est le service de découverte pour les applications **Spring Boot**, utilisé dans les architectures microservices pour localiser les services.

- En bas à gauche, on trouve un diagramme de classes UML avec des entités "**Client**", "**Terrain**", "**Category**", "**Taxe**", "**Taux**" et leurs relations, ce qui suggère la modélisation des données ou des objets au sein de l'application.

Notre figure représentant une vue d'ensemble de l'infrastructure technique de l'application avec des services d'authentification, une passerelle API, des services individuels avec des bases de données correspondantes, et un service de découverte pour orchestrer la communication entre les services.

## 1. Eureka-Service :

Eureka est une application conçue pour la localisation d'instances de services. Elle se compose d'une partie serveur et d'une partie cliente. La communication entre ces deux parties s'effectue via des API Web exposées par le composant serveur. Les services doivent être créés en tant que clients Eureka, se connectant et s'enregistrant sur un serveur Eureka. Ces clients peuvent ensuite s'enregistrer périodiquement auprès du serveur et fournir des signes de vie. Le service Eureka, en tant que composant serveur, conserve les informations de localisation des clients, les mettant ainsi à la disposition d'autres services en tant que registre de services. Vous pouvez trouver des informations plus détaillées dans la documentation disponible sur le dépôt Git d'Eureka.

## 2. API Gateway :

Un **API Gateway** (passerelle API) est un composant logiciel qui joue un rôle central dans l'architecture des services web et des microservices. Son rôle principal est de faciliter et de gérer l'accès aux services en exposant une API unifiée pour des services multiples. Voici quelques-unes de ses fonctionnalités principales :

1. **Routing des Requêtes** : L'API Gateway peut diriger les requêtes des clients vers les services appropriés en fonction de divers critères tels que le chemin de l'URL, les en-têtes, ou d'autres paramètres.
2. **Gestion de la Sécurité** : Il assure la sécurité en gérant l'authentification et l'autorisation des utilisateurs, en appliquant des politiques de sécurité telles que l'OAuth, et en cryptant les communications.
3. **Transformation de Données** : L'API Gateway peut transformer les données entre le format demandé par le client et le format utilisé par le service, facilitant ainsi l'intégration entre des services avec des formats de données différents.
4. **Mise en Cache** : Il peut mettre en cache les réponses des services, améliorant ainsi les performances en évitant des requêtes redondantes pour des données identiques.

5. **Gestion de Trafic** : Il gère la charge en effectuant la répartition de charge entre plusieurs instances de services pour garantir la stabilité et la disponibilité.
6. **Analytique** : Il collecte des données sur les requêtes, les temps de réponse, les erreurs, etc., pour fournir des informations analytiques et des statistiques sur l'utilisation des services.
7. **Monitoring et Logging** : L'API Gateway peut fournir des fonctionnalités de surveillance (monitoring) et de journalisation (logging) pour aider à diagnostiquer les problèmes et à suivre les performances.

En résumé, l'API Gateway agit comme un point d'entrée centralisé pour les clients qui souhaitent accéder à divers services dans une architecture distribuée. Il simplifie la gestion des communications entre les clients et les services, améliore la sécurité, la performance, et fournit des fonctionnalités d'analyse et de suivi.

### 3. Keycloak :

**Keycloak** est une solution open-source de gestion d'identité et d'accès (IAM - Identity and Access Management) développée par Red Hat. Cette plateforme propose des fonctionnalités complètes pour gérer les identités utilisateur, l'authentification, l'autorisation, ainsi que d'autres aspects liés à la sécurité des applications.

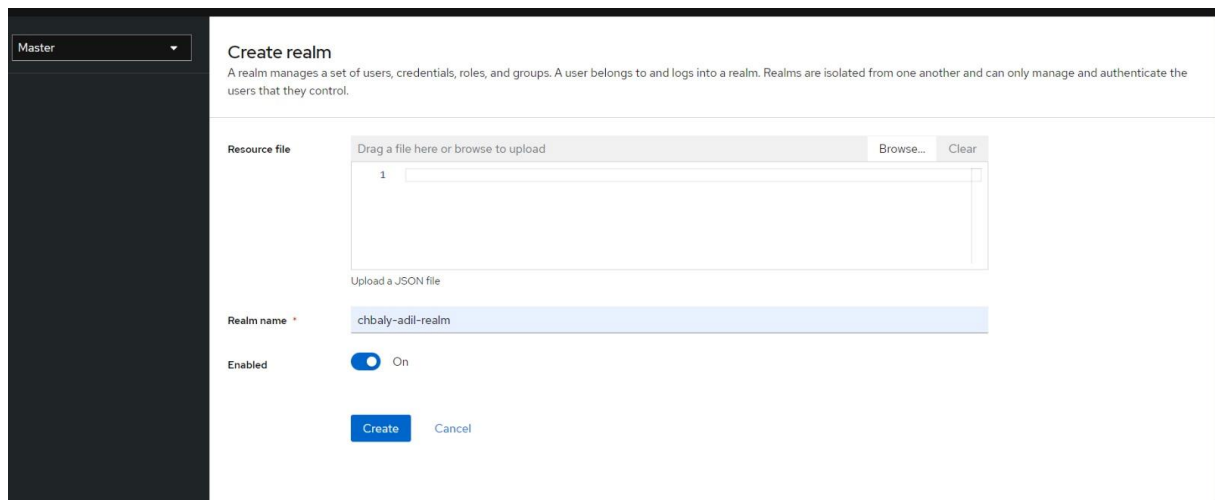
Voici quelques-unes des caractéristiques et des fonctionnalités clés de Keycloak :

1. **Gestion d'Identité** : Keycloak permet de gérer les identités des utilisateurs, y compris l'enregistrement, la modification, et la suppression des comptes utilisateur.
2. **Authentification et Autorisation** : Il prend en charge divers mécanismes d'authentification, y compris les utilisateurs et les mots de passe, la connexion sociale, et l'authentification à deux facteurs. Keycloak offre également des mécanismes d'autorisation robustes, permettant de définir des politiques d'accès granulaires.
3. **Single Sign-On (SSO)** : Keycloak facilite la mise en place du SSO, permettant aux utilisateurs de se connecter une fois et d'accéder à plusieurs applications sans avoir à se reconnecter.
4. **Fédération d'Identité** : Il prend en charge la fédération d'identité, permettant l'intégration avec d'autres systèmes d'authentification tels que SAML, OAuth, et OpenID Connect.
5. **Intégration avec les Protocoles Standards** : Keycloak utilise des protocoles standards tels que OAuth 2.0 et OpenID Connect, facilitant l'intégration avec une variété d'applications et de services.
6. **Administration et Tableau de Bord** : Il propose une interface d'administration graphique pour la gestion des utilisateurs, des clients, des rôles, et d'autres paramètres liés à la sécurité.

7. **Adaptabilité et Personnalisation** : Keycloak est conçu pour être extensible et personnalisable. Il offre des API permettant d'intégrer des fonctionnalités supplémentaires selon les besoins spécifiques.
8. **Audit et Logging** : La plateforme propose des fonctionnalités de journalisation (logging) et d'audit pour suivre les activités des utilisateurs et garantir la conformité.

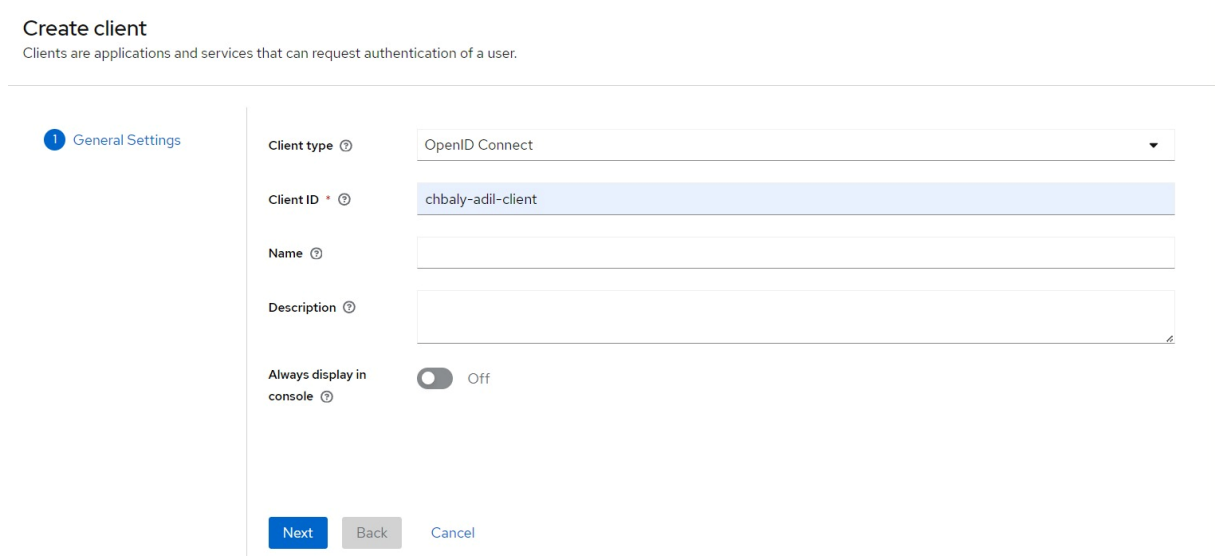
Keycloak est souvent utilisé comme composant central dans les architectures modernes de gestion d'identité et d'accès, offrant une solution complète et flexible pour répondre aux exigences de sécurité des applications et des services.

### 3.1. Etapes à suivre :



The screenshot shows the 'Create realm' page in Keycloak. On the left is a dark sidebar with a 'Master' dropdown. The main content area has the title 'Create realm' and a subtitle: 'A realm manages a set of users, credentials, roles, and groups. A user belongs to and logs into a realm. Realms are isolated from one another and can only manage and authenticate the users that they control.' Below this, there's a 'Resource file' section with a text area for a file path, a 'Browse...' button, and a 'Clear' button. An 'Upload a JSON file' link is also present. The 'Realm name' field is filled with 'chbaly-adil-realm'. The 'Enabled' toggle switch is turned 'On'. At the bottom are 'Create' and 'Cancel' buttons.

Figure 2 : Interface de creation de realm



The screenshot shows the 'Create client' page in Keycloak. The title is 'Create client' with a subtitle: 'Clients are applications and services that can request authentication of a user.' On the left is a sidebar with a 'General Settings' tab. The main content area has several fields: 'Client type' is set to 'OpenID Connect'; 'Client ID' is filled with 'chbaly-adil-client'; 'Name' and 'Description' are empty text areas. The 'Always display in console' toggle switch is turned 'Off'. At the bottom are 'Next', 'Back', and 'Cancel' buttons.

Figure 3 : Interface de création du client

## Access settings

Root URL <sup>?</sup>

Home URL <sup>?</sup>

Valid redirect URIs <sup>?</sup>  ⊖  
[+ Add valid redirect URIs](#)

Valid post logout redirect URIs <sup>?</sup>  ⊖  
[+ Add valid post logout redirect URIs](#)

Web origins <sup>?</sup>  ⊖  
[+ Add web origins](#)

Admin URL <sup>?</sup>

Figure 4 : Access settings and web cors

USER Action ▾

Details

Attributes

Users in role

Permissions

Role name \*

Description

[Save](#) [Revert](#)

Figure 5 : Creation du role USER

ADMIN Action ▾

Details

Attributes

Users in role

Permissions

Role name \*

Description

[Save](#) [Revert](#)

Figure 6 : Creation du role ADMIN

Username \* admin

Email new1@gmail.com

Email verified ⓘ ☒ On

First name admin

Last name

Enabled ⓘ ☒ On

Required user actions ⓘ Select action ▼

Groups ⓘ [Join Groups](#)

[Create](#) [Cancel](#)

Figure 7 : Creation du compte ADMIN

Assign roles to admin account

Filter by roles ▼ Search by role name → 1-4 ▼ < >

<input type="checkbox"/>	Name	Description
<input checked="" type="checkbox"/>	ADMIN	
<input type="checkbox"/>	offline_access	`\${role_offline-access}`
<input type="checkbox"/>	uma_authorization	`\${role_uma_authorization}`
<input type="checkbox"/>	USER	

1-4 ▼ < >

[Assign](#) [Cancel](#)

Figure 8 : Assigner le role ADMIN au compte ADMIN



Username \*

Email

Email verified ⓘ ☒ On

First name

Last name

Enabled ⓘ ☒ On

Required user actions ⓘ

Groups ⓘ

Figure 9 : Création du compte USER

Assign roles to adil account ✕

1 - 4

<input type="checkbox"/>	Name	Description
<input type="checkbox"/>	ADMIN	
<input type="checkbox"/>	offline_access	`\${role_offline-access}`
<input type="checkbox"/>	uma_authorization	`\${role_uma_authorization}`
<input checked="" type="checkbox"/>	USER	

1 - 4

Figure 10 : Assigner le role USER au compte USER

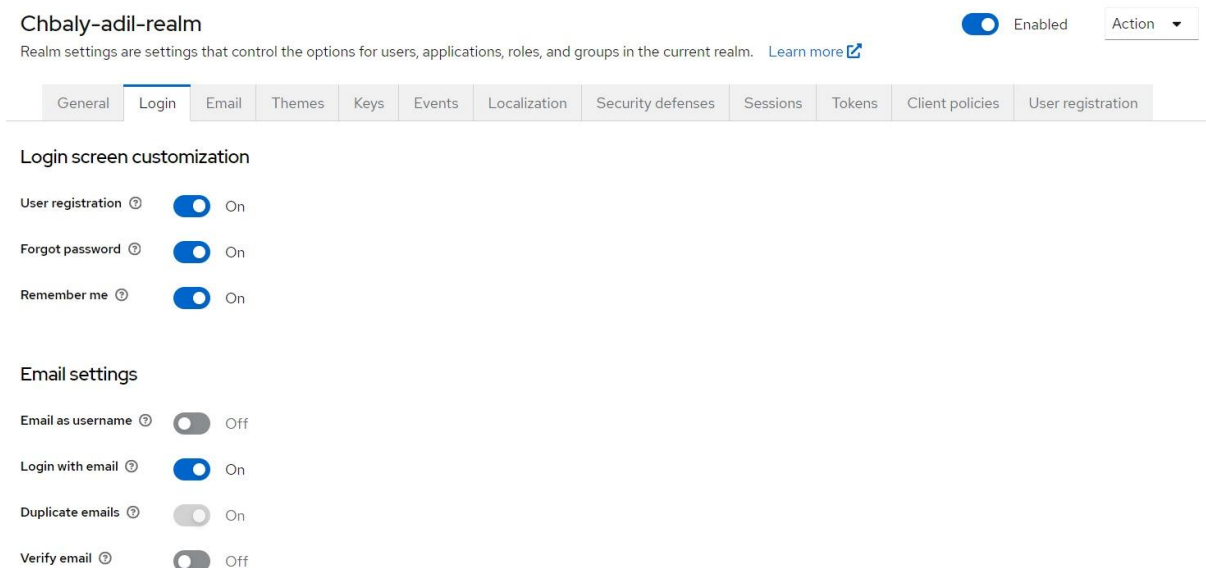


Figure 11 : Configuration du login et registry

#### 4. Description globale des microservices utilisés :

Ce schéma architectural de notre écosystème d'application web impliquant plusieurs technologies et composants.

- En haut à gauche, on trouve une représentation de la **sécurité** impliquant **Keycloak**, une solution open source de gestion d'identité et d'accès. Il offre la fédération d'utilisateurs, une authentification forte, la gestion des utilisateurs, une autorisation fine et est intégré à **OpenID** pour la vérification de l'identité.

- En dessous des composants de sécurité et de base de données, il y a un logo **Angular** marqué avec "**npm**", indiquant que le côté client de l'application est construit en utilisant le framework Angular et que les dépendances sont gérées avec **npm** (Node Package Manager).

- Au centre de la figure, il y a une **API Gateway**, qui est le point d'entrée pour les demandes des clients vers les services backend. Elle agit comme un proxy inverse, routant les demandes des clients vers les services backend appropriés.

- Trois services backend sont indiqués par des flèches circulaires vertes, chacun ayant une base de données correspondante représentée par des icônes de cylindres bleus :

- **taux-service** avec une base de données nommée **Taux-db**

- **Terrain-service** avec une base de données nommée **Terrain-db**

- **taxe-tnb-service** avec une base de données nommée **Taxe-db**

- Au-dessus de ces services, il y a **Eureka**, suggérant que le système utilise Netflix Eureka pour la découverte de services. Eureka permet aux services de s'enregistrer à l'exécution au fur et à mesure de leur apparition dans le paysage système et de découvrir d'autres services pour la communication.

- En bas à gauche, il y a une section **Client** avec un schéma entité-relation (ERD) simpliste montrant les relations entre différentes entités : Terrain, Catégorie, Taxe et Taux. Cela suggère que le **client** interagit avec ces **entités**, via l'**API Gateway**.

Dans l'ensemble, le diagramme illustre une architecture de microservices avec une interface utilisateur **Angular**, une couche de sécurité **Keycloak**, un mécanisme de découverte de services via **Eureka**, une **API Gateway** routant vers divers microservices, chacun avec sa propre base de données **PostgreSQL**, le tout fonctionnant dans un environnement conteneurisé, probablement sur une plateforme Linux.

---

### En général :

**Authentification et Sécurité** : Le système utilise Keycloak avec OpenID pour gérer l'authentification et la sécurité.

**Discovery Service** : Eureka est utilisé comme service de découverte pour maintenir la liste des services disponibles.

**Base de données** : Il y a trois bases de données séparées pour les différents microservices, probablement pour séparer les domaines de données (Taux-db, Terrain-db, Taxe-db).

**Microservices** : Il y a des microservices distincts pour gérer différentes entités ou domaines fonctionnels (taux-service, terrain-service, taxe-service).

**API Gateway** : Un point d'entrée unique pour les clients qui peut acheminer les requêtes vers les microservices appropriés.

**Clients** : Les utilisateurs interagissent avec le système via un client qui communique avec l'API Gateway.

**Technologies Front-end** : Angular et NPM sont indiqués, suggérant qu'ils sont utilisés pour le développement front-end.

**Logiciel de Containerisation** : Docker est représenté, ce qui suggère que les services peuvent être déployés dans des conteneurs pour faciliter le déploiement et la scalabilité.

**N.B** : Vous trouvez ci-joint le lien GitHub du projet global :

<https://github.com/AdilCHBALY/msa-docker-keycloak-r4j-openFeign>

### Conclusion :

Ce Chapitre axé sur le développement de l'architecture des microservices pour le projet de la Taxe sur les Terrains Non Bâties (TNB) au Maroc, offre une vision globale et détaillée de la structure technique de l'application. En décrivant les différentes composantes, du service d'authentification Keycloak à la passerelle API, des microservices individuels aux bases de données associées, le chapitre établit une fondation solide pour la gestion efficace de la taxe

TNB. La représentation architecturale montre une orchestration harmonieuse des éléments, avec des services distincts pour gérer les taux, les terrains et les taxes, tous interconnectés et fonctionnant dans un environnement conteneurisé. En utilisant des technologies telles qu'Angular et npm pour le développement front-end, Eureka pour la découverte de services, et Docker pour la containerisation, l'architecture est conçue pour répondre aux exigences modernes de sécurité, de scalabilité et d'efficacité. Dans l'ensemble, ce chapitre jette les bases solides d'un système de gestion de taxe TNB robuste, transparent et efficient basé sur une architecture microservices bien pensée.

## Chapitre 2 : conteneurisation de l'application

### Introduction :

Ce chapitre explore l'intégration de Docker Compose avec Keycloak, une solution open-source de gestion des identités et des accès. Docker Compose simplifie le déploiement d'applications Docker multi-conteneurs grâce à un fichier de configuration YAML. En combinant ces deux technologies, nous créons un environnement efficace pour déployer Keycloak et ses dépendances. La première section se concentre sur l'utilisation de Docker Compose pour Keycloak, tandis que la deuxième examine le "docker-compose" global du projet, orchestrant l'ensemble des services nécessaires. Cette approche facilite le déploiement et la gestion cohérente de l'application, en mettant en avant la synergie entre Docker Compose et Keycloak.

### 1. Docker compose keycloak :

"**Docker Compose**" est un outil qui permet de définir et gérer des applications Docker multi-conteneurs. Il utilise un fichier YAML pour configurer les services, les réseaux et les volumes, et permet de démarrer l'ensemble de l'application avec une seule commande.

"**Keycloak**" est un logiciel open-source de gestion des identités et des accès. Il offre des fonctionnalités telles que l'authentification unique (SSO), la gestion des utilisateurs, la fédération d'identité, etc. Il est souvent utilisé pour sécuriser et gérer l'accès aux applications et services.

"**Docker Compose Keycloak**" fait référence à l'utilisation de Docker Compose pour déployer un environnement Keycloak avec plusieurs conteneurs interconnectés. Cela inclut généralement le conteneur principal de Keycloak, éventuellement des bases de données, des serveurs de messagerie ou d'autres services nécessaires au bon fonctionnement de Keycloak.

En résumé, Docker Compose facilite le déploiement et la gestion d'un ensemble de conteneurs Docker, tandis que Keycloak fournit une solution d'authentification et de gestion des accès. Utiliser Docker Compose avec Keycloak permet de simplifier le processus de déploiement de Keycloak et de ses dépendances.

```

1 version: '3'
2
3 volumes:
4   postgres_data:
5     driver: local
6
7 services:
8   postgres:
9     image: postgres
10    volumes:
11      - postgres_data:/var/lib/postgresql/data
12    environment:
13      POSTGRES_DB: keycloak
14      POSTGRES_USER: keycloak
15      POSTGRES_PASSWORD: password
16   keycloak:
17     image: quay.io/keycloak/keycloak:19.0.3
18     environment:
19       DB_VENDOR: POSTGRES
20       DB_ADDR: postgres
21       DB_DATABASE: keycloak
22       DB_USER: keycloak
23       DB_SCHEMA: public
24       DB_PASSWORD: password
25       KEYCLOAK_ADMIN: admin
26       KEYCLOAK_ADMIN_PASSWORD: Pa55w0rd
27     command:
28       - start-dev
29     ports:
30       - 8080:8080
31     depends_on:
32       - postgres
3

```

Figure 12 : Contenu du fichier keycloak-docker.yml

## 2. Docker compose du projet global :

```
1  version: '3'
2  services:
3    eureka-server:
4      container_name: eureka-server
5      build: ./eureka-server
6      ports:
7        - "8761:8761"
8      healthcheck:
9        test: [
10          "CMD",
11          "curl",
12          "-f",
13          "http://localhost:8761/actuator/health"
14        ]
15      interval: 10s
16      retries: 3
```

Figure 13 : Contenu du fichier global docker-compose.yml (partie1)

```
17  api-gateway:
18    container_name: api-gateway
19    build: ./gateway
20    ports:
21      - "8888:8888"
22    environment:
23      DISCOVERY_SERVICE_URL: http://eureka-server:8761/eureka/
24    healthcheck:
25      test: [
26        "CMD",
27        "curl",
28        "-f",
29        "http://api-gateway:8888/actuator/health"
30      ]
31      interval: 10s
32      retries: 3
33    depends_on:
34      eureka-server:
35        condition: service_healthy
```

Figure 14 : Contenu du fichier global docker-compose.yml (partie2)

```

36     taux-service:
37         container_name: taux-service
38         build: ./taux-tnb-service
39         ports:
40             - "8089:8089"
41         environment:
42             DISCOVERY_SERVICE_URL: http://eureka-server:8761/eureka/
43         depends_on:
44             eureka-server:
45                 condition: service_healthy
46         healthcheck:
47             test: [
48                 "CMD",
49                 "curl",
50                 "-f",
51                 "http://taux-service:8089/actuator/health"
52             ]
53         interval: 10s
54         retries: 3

```

Figure 15 : Contenu du fichier global docker-compose.yml (partie3)

```

55     taxe-service:
56         container_name: taxe-service
57         build: ./taxe-tnb-service
58         ports:
59             - "8084:8084"
60         environment:
61             DISCOVERY_SERVICE_URL: http://eureka-server:8761/eureka/
62         depends_on:
63             eureka-server:
64                 condition: service_healthy
65         healthcheck:
66             test: [
67                 "CMD",
68                 "curl",
69                 "-f",
70                 "http://taxe-service:8084/actuator/health"
71             ]
72         interval: 10s
73         retries: 3

```

Figure 16 : Contenu du fichier global docker-compose.yml (partie4)



```

74     terrain-category-service:
75         container_name: terrain-category-service
76         build: ./terrain-service
77         ports:
78             - "8085:8085"
79         environment:
80             DISCOVERY_SERVICE_URL: http://eureka-server:8761/eureka/
81         depends_on:
82             eureka-server:
83                 condition: service_healthy
84         healthcheck:
85             test:
86                 [
87                     "CMD",
88                     "curl",
89                     "-f",
90                     "http://terrain-category-service:8085/actuator/health"
91                 ]
92             interval: 10s
93             retries: 3

```

Figure 17 : Contenu du fichier global docker-compose.yml (partie5)

```

94     angular-front:
95         container_name: angular-front
96         build: ./tnb-front
97         ports:
98             - "4200:80"
99         depends_on:
100             api-gateway:
101                 condition: service_healthy
102             taux-service:
103                 condition: service_healthy
104             taxe-service:
105                 condition: service_healthy
106             terrain-category-service:
107                 condition: service_healthy

```

Figure 18 : Contenu du fichier global docker-compose.yml (partie6)

## Conclusion :

Ce Chapitre consacré à la conteneurisation de l'application avec Docker Compose et Keycloak, offre un aperçu détaillé de la manière dont ces deux technologies s'intègrent pour créer un environnement de déploiement efficace. En mettant en avant l'utilisation stratégique de Docker Compose pour déployer Keycloak avec ses dépendances, le chapitre souligne la simplicité et la cohérence du processus de déploiement. La première partie se concentre sur l'orchestration de conteneurs spécifiques à Keycloak, tandis que la deuxième partie examine le fichier global de Docker Compose qui coordonne l'ensemble des services nécessaires au projet. Cette approche renforce la gestion harmonieuse de l'application, soulignant la synergie entre Docker Compose et Keycloak dans le contexte de notre projet axé sur les microservices.

## Chapitre 3 : Automatisation et Intégration Continue avec Jenkins

### Introduction :

Ce chapitre consacré à Jenkins constitue une plongée approfondie dans l'intégration continue et le déploiement continu (CI/CD) au sein de notre projet. Jenkins, en tant que serveur d'automatisation, joue un rôle essentiel dans l'accélération du cycle de vie du développement logiciel en orchestrant des processus automatisés, du build à la livraison.

Ce chapitre mettra en lumière les principes fondamentaux de Jenkins, détaillant son architecture, ses fonctionnalités clés et son rôle central dans l'assurance qualité et le déploiement efficace des microservices. Nous explorerons également la configuration de pipelines CI/CD, la gestion des versions, et l'intégration avec d'autres outils de développement.

L'objectif ultime de ce chapitre est de démontrer comment Jenkins devient un pilier central de notre infrastructure DevOps, contribuant à l'amélioration de l'efficacité du développement, à la réduction des erreurs, et à la garantie d'une livraison logicielle fiable et régulière.

### Définition du Jenkins :

Jenkins est un outil logiciel open source d'intégration continue écrit en Java. Il permet de tester et de signaler en temps réel des modifications isolées dans un code de grande ampleur. Ce logiciel permet aux développeurs de rechercher et de résoudre rapidement les anomalies, ainsi que d'automatiser les tests de leurs builds.

L'intégration continue a évolué depuis sa conception. Au départ, un build quotidien était la norme. Aujourd'hui, en règle générale, chaque membre d'une équipe soumet son travail tous les jours (voire plus fréquemment) et un build est effectué à chaque modification importante.

Bien utilisée, l'intégration continue offre divers avantages, tels que les retours constants sur l'état du logiciel. Comme l'intégration continue permet de détecter les anomalies en amont au cours du développement, les erreurs restantes sont généralement plus petites, moins complexes et plus faciles à résoudre.

Jenkins est dérivé d'un projet appelé Hudson, qui appartient à Oracle et continue à être développé en parallèle.

### Pourquoi utiliser Jenkins ? :

L'utilisation de Jenkins dans notre projet peut apporter plusieurs avantages importants, notamment dans le contexte d'un projet de microservices.

1. **Intégration Continue (CI) :** Jenkins facilite la mise en œuvre de l'intégration continue, permettant une intégration régulière et automatisée des modifications de code dans le référentiel. Cela contribue à identifier rapidement les erreurs d'intégration et à maintenir un code de haute qualité.

2. **Déploiement Continu (CD)** : Jenkins permet de mettre en place des pipelines de déploiement continu, automatisant le processus de déploiement des microservices. Cela garantit des déploiements fréquents, fiables et reproductibles.
3. **Orchestration des Microservices** : Avec Jenkins, vous pouvez orchestrer et coordonner les déploiements des microservices, en gérant les dépendances et en assurant une mise en production harmonieuse.
4. **Gestion des Versions** : Jenkins peut être intégré à des outils de gestion de versions tels que Git, facilitant ainsi le suivi des modifications de code, la gestion des branches, et la synchronisation cohérente entre les différentes parties du projet.
5. **Extensibilité** : Jenkins offre une large gamme de plugins permettant d'intégrer facilement d'autres outils et technologies. Cela vous donne la flexibilité nécessaire pour adapter Jenkins à vos besoins spécifiques.
6. **Surveillance et Rapports** : Jenkins fournit des tableaux de bord et des rapports détaillés sur l'état des builds, des tests et des déploiements. Cela permet une surveillance efficace de l'état du projet et facilite l'identification rapide des problèmes.
7. **Communauté Active** : Jenkins bénéficie d'une communauté active et de nombreux tutoriels, ce qui facilite l'apprentissage et la résolution des problèmes éventuels.
8. **Automatisation** : Jenkins permet d'automatiser diverses tâches, réduisant ainsi la charge manuelle sur les équipes de développement et garantissant des processus cohérents.

En résumé, Jenkins offre une solution robuste d'intégration continue et de déploiement continu, adaptée aux projets de microservices en permettant une automatisation efficace, une gestion des versions précise, et une orchestration fiable des microservices.

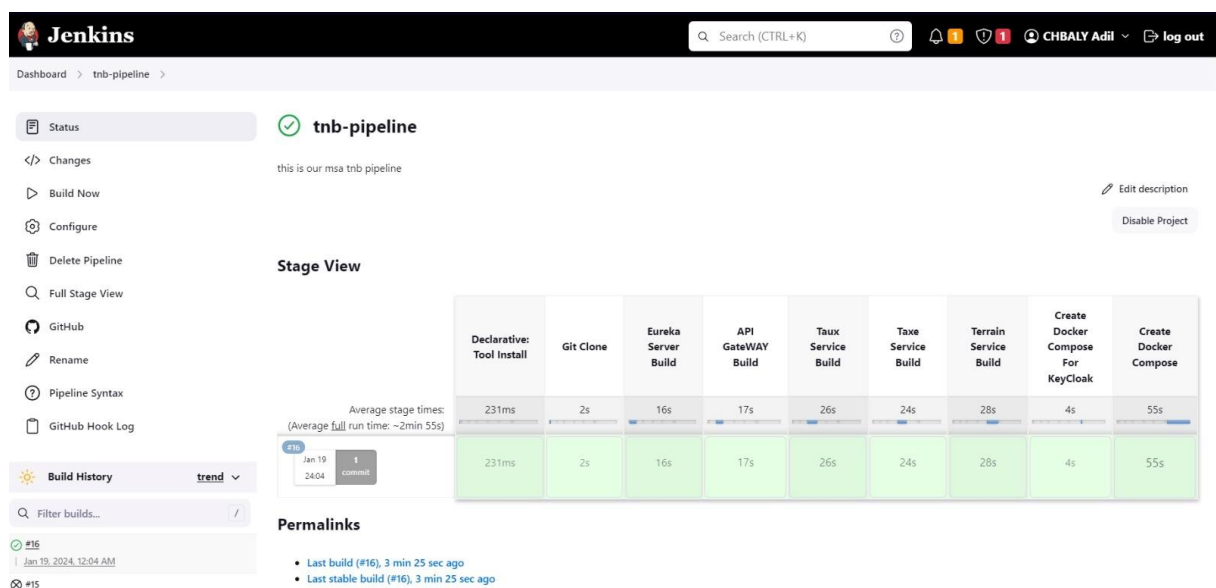


Figure 19 : Affichage après le lancement de notre pipeline

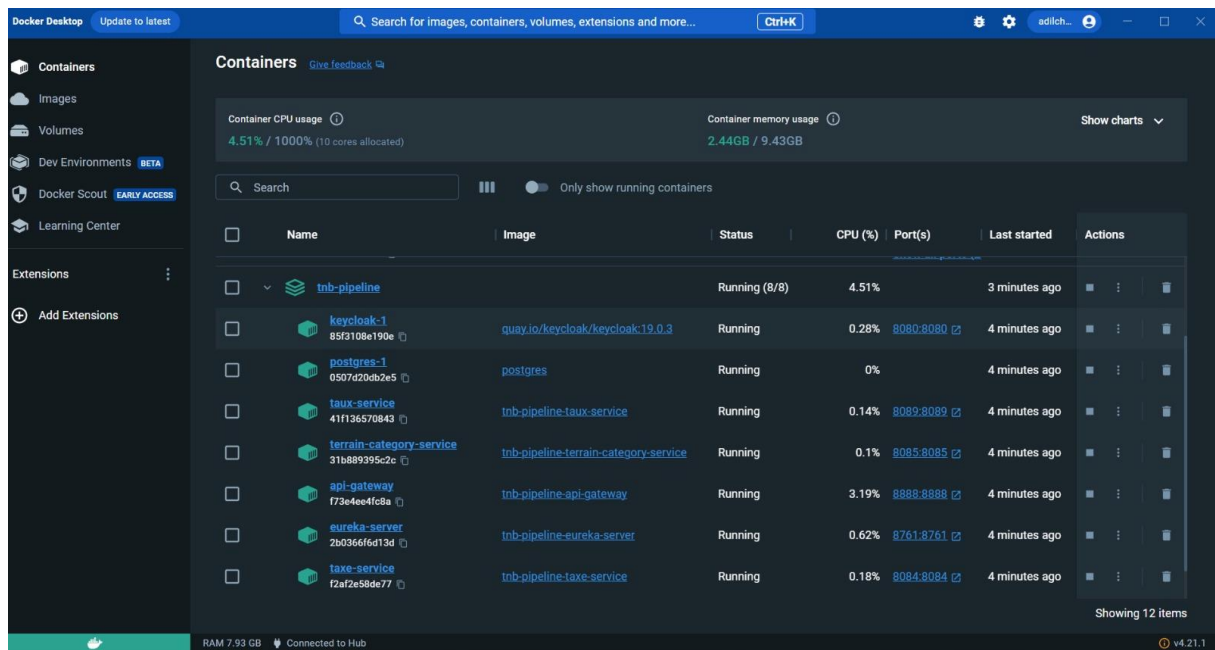


Figure 20 : Notre docker-compose final

## Conclusion :

Ce Chapitre dédié à Jenkins, axe central de l'intégration continue et du déploiement continu (CI/CD), offre une plongée approfondie dans les principes et les avantages fondamentaux de cette plateforme. En mettant en évidence son rôle essentiel dans l'accélération du cycle de vie du développement logiciel, de la construction à la livraison, le chapitre explore l'architecture de Jenkins, ses fonctionnalités clés, et son intégration harmonieuse dans notre infrastructure DevOps. L'objectif central est de démontrer comment Jenkins devient un pilier incontournable pour améliorer l'efficacité du développement, réduire les erreurs et garantir des livraisons logicielles fiables et régulières. À travers la définition approfondie de Jenkins, les avantages de son utilisation, et des exemples concrets de son intégration dans notre pipeline CI/CD, le chapitre établit clairement le rôle pivot de Jenkins dans notre approche de développement axée sur les microservices.

## Chapitre 4 : SonarQube - Outil de Qualité du Code en Continu

### Introduction :

Dans le contexte de notre projet axé sur les microservices, la qualité du code joue un rôle crucial dans la pérennité et la robustesse de l'application. Pour garantir une qualité logicielle constante et détecter rapidement les défauts potentiels, nous intégrons SonarQube dans notre pipeline d'intégration continue et de déploiement continu (CI/CD).

Ce chapitre se penche sur l'implémentation de SonarQube au sein de notre processus de développement, offrant une analyse approfondie de la qualité du code à chaque étape du cycle de vie. Nous explorerons les avantages de cette intégration, notamment la détection proactive de défauts, la mesure de la duplication de code, l'évaluation de la documentation, et la surveillance de la couverture des tests.

Au fil des sections, nous aborderons la configuration de SonarQube, son intégration harmonieuse avec Jenkins, et la manière dont ces deux outils travaillent de concert pour améliorer la qualité du code. En adoptant cette approche, notre objectif est d'élever la qualité de notre code à un niveau supérieur, contribuant ainsi à la stabilité et à la fiabilité de nos microservices.

### 1. C'est quoi SonarQube ?

SonarQube, anciennement connu sous le nom de Sonar2, est un logiciel open source dédié à la qualité du code. Il offre une approche continue de la Qualimétrie, facilitant la détection, la classification et la résolution des défauts dans le code source. Parmi ses fonctionnalités, on retrouve la capacité à identifier les duplications de code, à mesurer le niveau de documentation, et à évaluer la couverture de test déployée.

L'interface web intuitive de SonarQube permet une surveillance continue de la qualité du code. Elle offre une vue complète des défauts présents dans l'ensemble du code, y compris ceux ajoutés par les nouvelles versions. SonarQube peut être intégré à des systèmes d'automatisation tels que Jenkins, permettant ainsi d'inclure l'analyse de qualité comme une extension naturelle du processus de développement.

SonarQube devient ainsi un allié précieux dans l'amélioration continue du code, favorisant des pratiques de développement saines et contribuant à la création de logiciels robustes et fiables.

### 2. Architecture de SonarQube :

Ce processus de développement logiciel intégrant des outils de versionnage, de conteneurisation et d'analyse de la qualité du code. Voici une décomposition détaillée :

1. **"git" avec "git push" :**

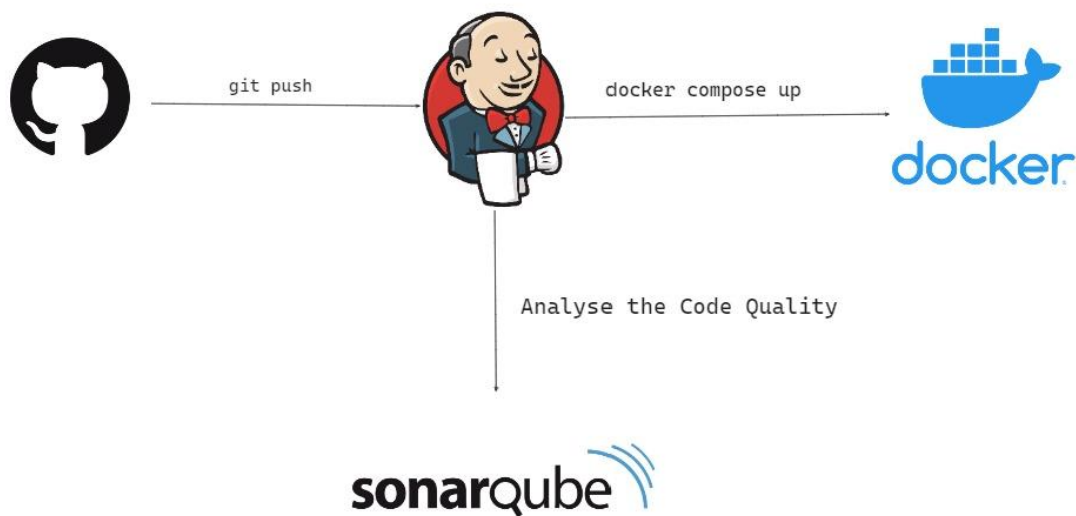
- "git" représente un système de contrôle de version.
- "git push" suggère l'action de pousser le code vers un dépôt distant.
- L'endurance du développeur dans le processus de développement.

2. **"Docker" avec "docker compose up" :**

- "Docker" est présent, représentant la conteneurisation d'applications.
- "docker compose up" suggère la commande pour lancer des applications dans des conteneurs Docker.

3. **"Analyse the Code Quality" connecté à l'icône de "SonarQube" :**

- En dessous du personnage central, le texte "Analyse the Code Quality" indique l'étape d'analyse de la qualité du code.
- L'icône de "SonarQube" est associée à l'outil utilisé pour cette analyse.



*Figure 21 Création de notre CI/CD Pipeline avec Jenkins*

L'ensemble dépeint un cycle de développement où le code est poussé, l'application est exécutée dans des conteneurs Docker, et la qualité du code est évaluée via SonarQube. Le personnage central représente la cohésion du développeur dans ce processus séquentiel.

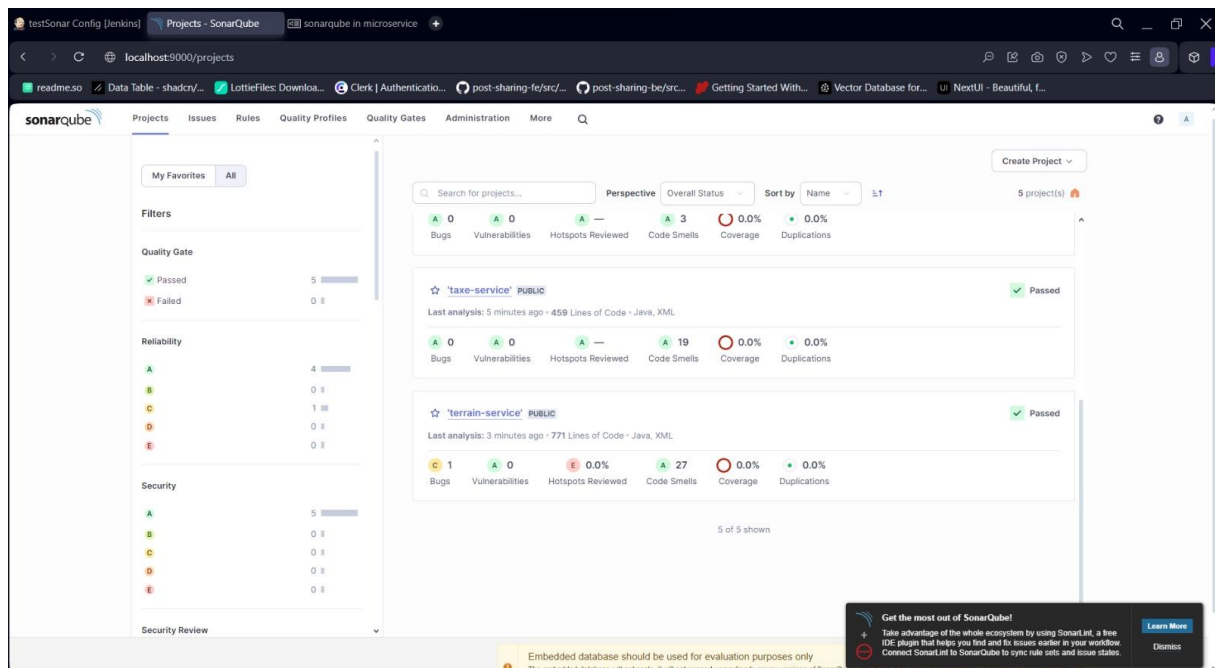


Figure 22 Screen de notre interface SonarQube et SonarQube Analysis (Partie 1)

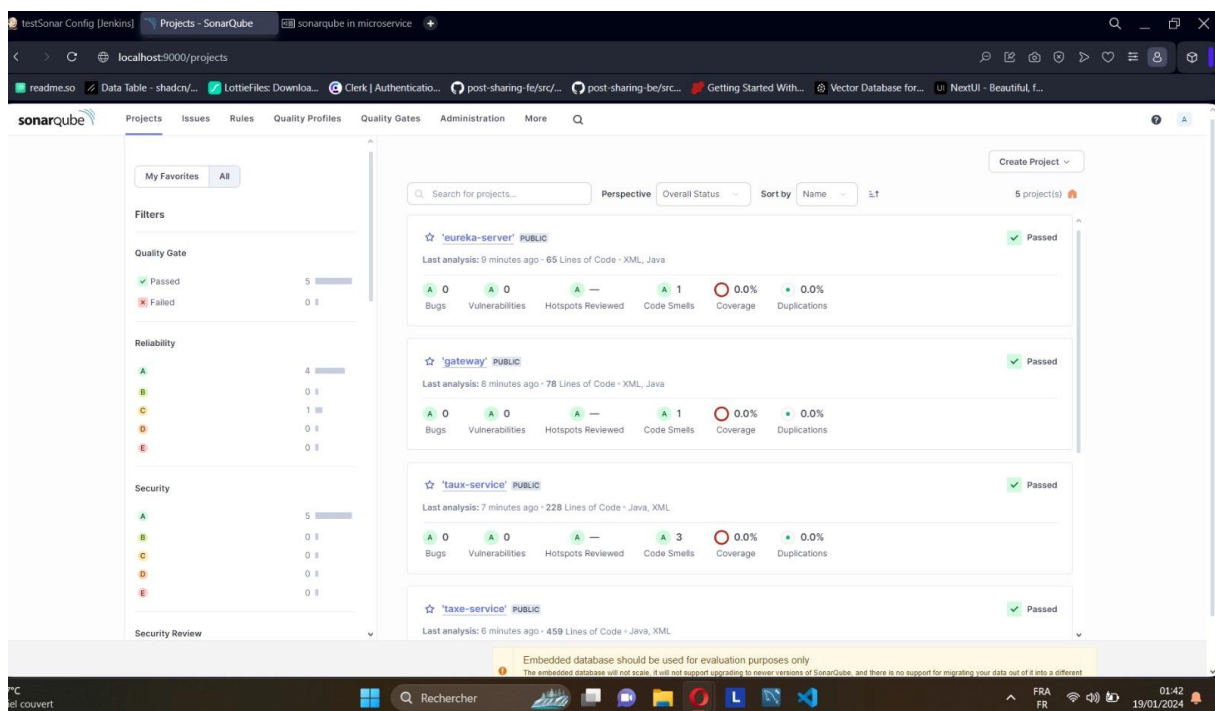


Figure 23 Screen de notre interface SonarQube et SonarQube Analysis (Partie 2)



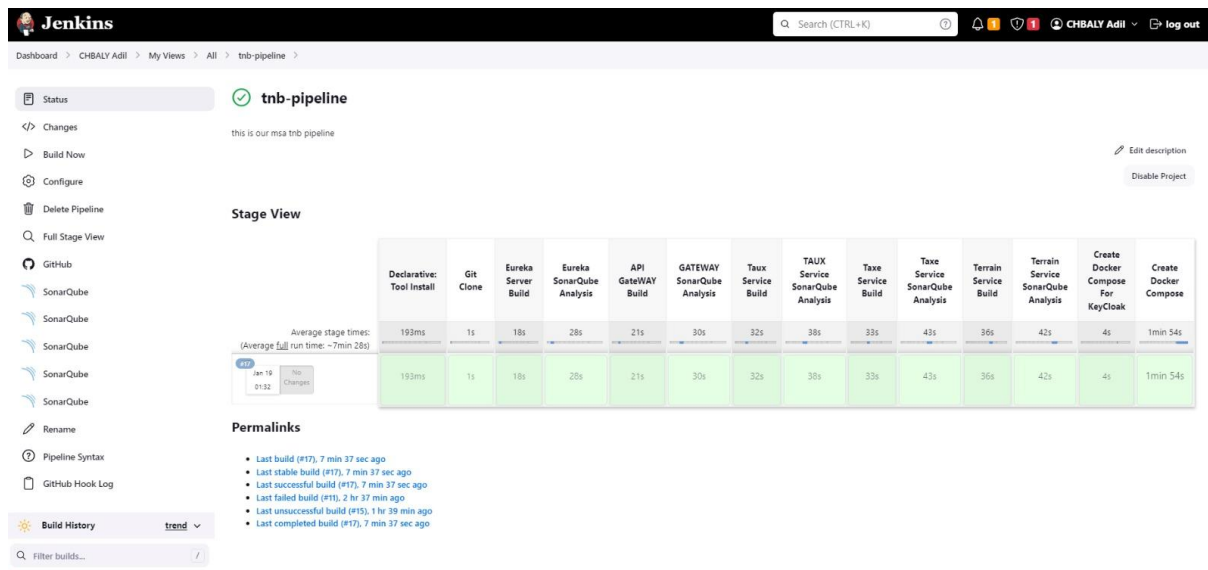


Figure 24 : Et Voilà Notre Pipeline est bien fonctionnel

### 3. Intégration harmonieuse : Optimiser les Déclenchements Automatiques avec les Webhooks GitHub et Jenkins :

Cette partie évoque l'intégration fluide entre GitHub et Jenkins, deux outils essentiels dans le processus de développement logiciel moderne. L'utilisation de webhooks pour déclencher automatiquement des processus dans Jenkins à partir des événements sur GitHub promet une coordination harmonieuse, améliorant ainsi l'efficacité du flux de travail de développement. L'accent sur l'optimisation des déclenchements automatiques suggère une approche réfléchie pour maximiser les avantages de cette intégration, renforçant ainsi la qualité et la rapidité des déploiements logiciels.

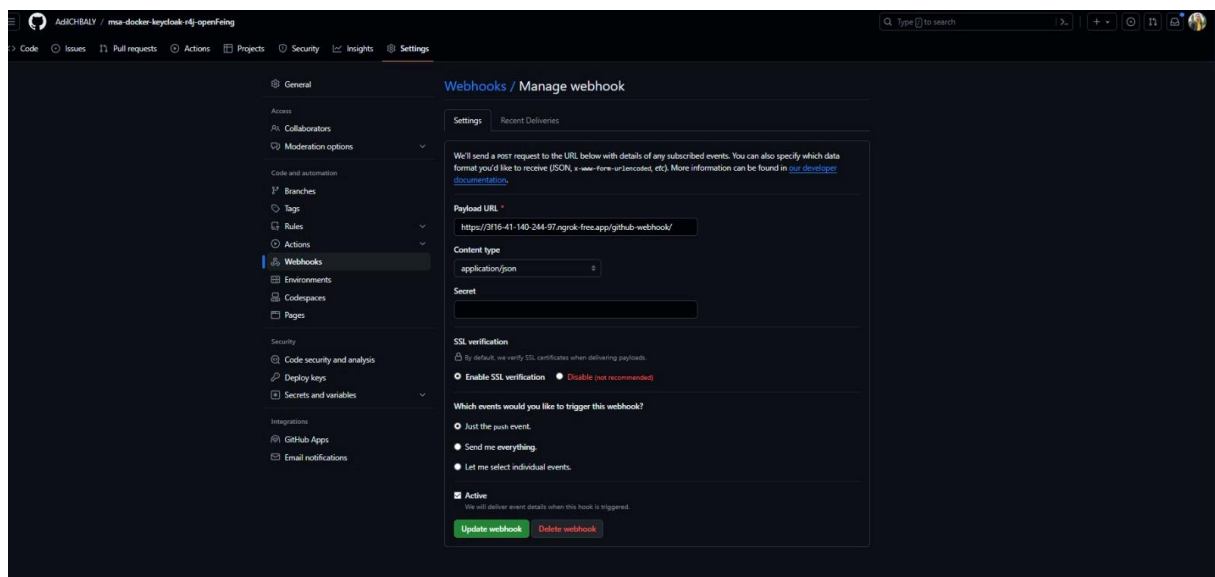


Figure 25 : Création de notre Webhook avec GitHub

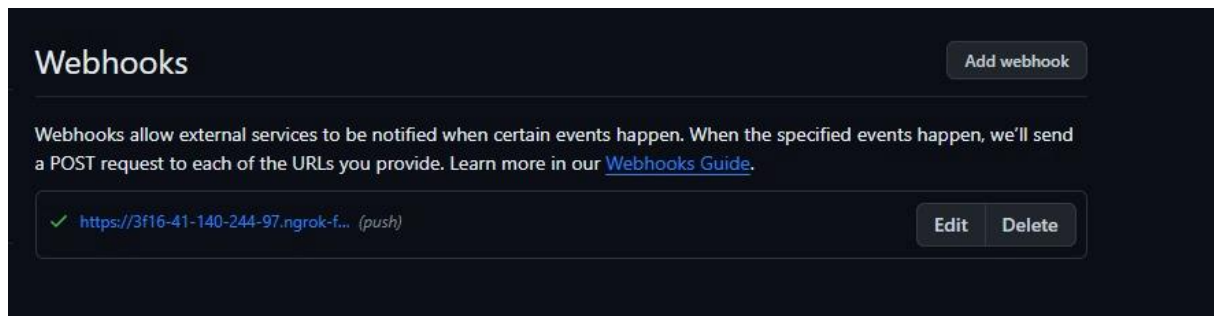


Figure 26 : Le Webhook marche bien avec notre serveur Jenkins

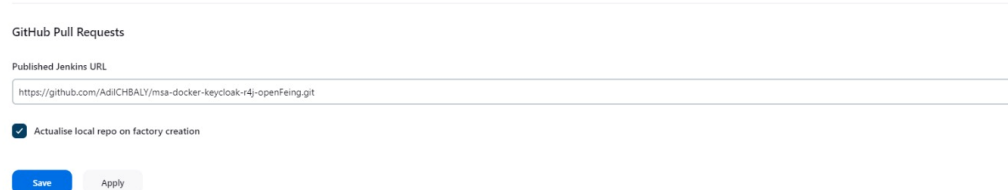


Figure 27 : Ajout du lien GitHub dans le Pull Request System

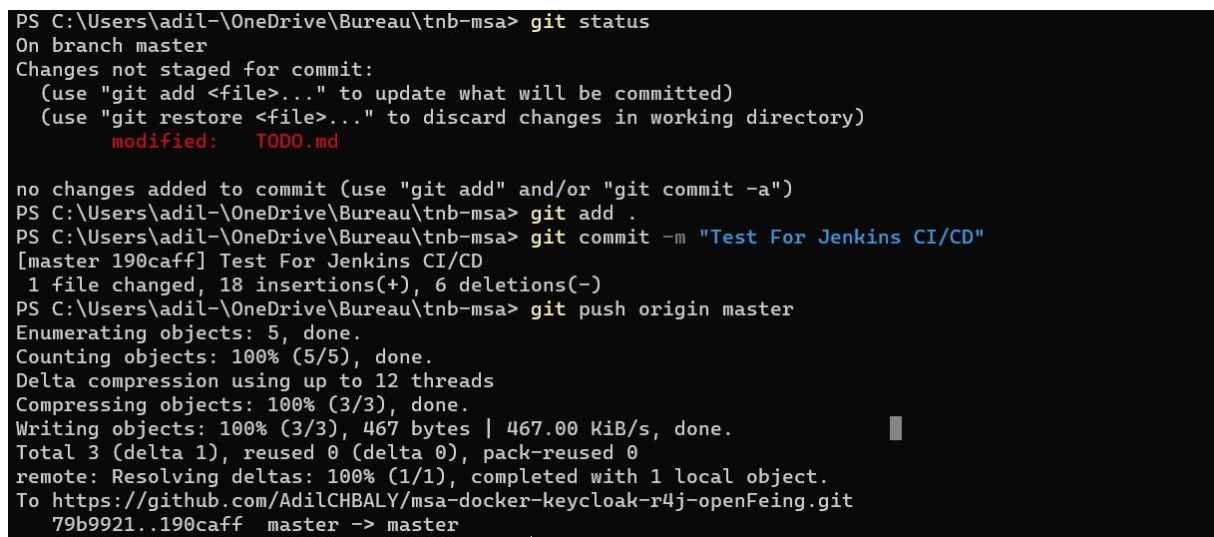


Figure 28 : Pusher un changement

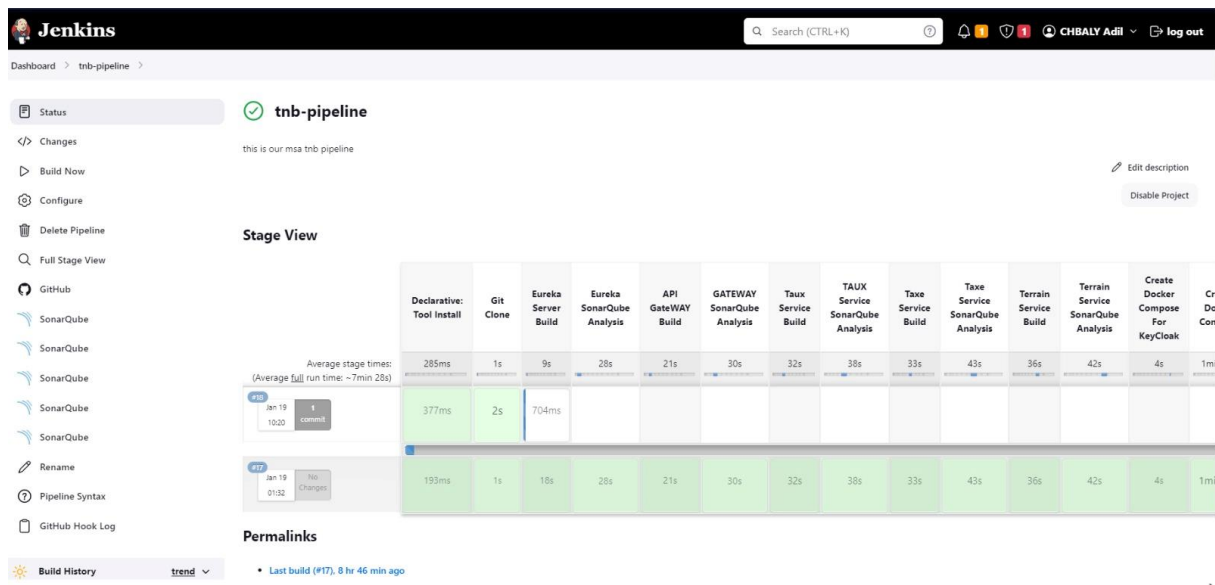


Figure 29 : Le CI/CD marche bien

```

pipeline {
  agent any
  tools {
    maven 'maven'
    nodejs 'node'
  }

  stages {
    stage('Git Clone') {
      steps {
        script {
          checkout([$class: 'GitSCM', branches: [[name: 'master']], userRemoteConfigs: [[url: 'https://github.com/AdilCHBALY/msa-docker-keycloak-r4j-openFeing.git']]])
        }
      }
    }
    stage('Eureka Server Build') {
      steps {
        dir('eureka-server') {
          bat 'mvn clean install'
        }
      }
    }
  }
}

```

Figure 30 : Contenu du JenkinsFile (partie 1)

```

stage('Eureka Server Build') {
    steps {
        dir('eureka-server') {
            bat 'mvn clean install'
        }
    }
}

stage('Eureka SonarQube Analysis'){
    steps{
        dir('eureka-server'){
            withSonarQubeEnv('sonarqube-server'){
                bat "mvn clean verify sonar:sonar \
-Dsonar.projectKey=eureka-server \
-Dsonar.projectName='eureka-server' \
-Dsonar.host.url=http://localhost:9000 \
-Dsonar.token=sqp_1362c34c86c64c0c9b8437516fcc86f6c699420e"
            }
        }
    }
}

stage('API GateWAY Build') {
    steps {
        dir('gateway') {
            bat 'mvn clean install'
        }
    }
}

```

Figure 31 : Contenu du JenkinsFile (partie 2)

```

stage('GATEWAY SonarQube Analysis'){
    steps{
        dir('gateway'){
            withSonarQubeEnv('sonarqube-server'){
                bat "mvn clean verify sonar:sonar \
-Dsonar.projectKey=gateway \
-Dsonar.projectName='gateway' \
-Dsonar.host.url=http://localhost:9000 \
-Dsonar.token=sqp_7367234622190d81c8a3491b39ce80cce3f53461"
            }
        }
    }
}

stage('Taux Service Build') {
    steps {
        dir('taux-tnb-service') {
            bat 'mvn clean install'
        }
    }
}

stage('TAUX Service SonarQube Analysis'){
    steps{
        dir('taux-tnb-service'){
            withSonarQubeEnv('sonarqube-server'){
                bat "mvn clean verify sonar:sonar \

```

Figure 32 : Contenu du JenkinsFile (partie 3)

```

-Dsonar.projectKey=taux-service \
-Dsonar.projectName='taux-service' \
-Dsonar.host.url=http://localhost:9000 \
-Dsonar.token=sqp_ae29bee089fbc73abfa5aacce4c706b6cb489039"
    }
    }
}
stage('Taxe Service Build') {
    steps {
        dir('taxe-tnb-service') {
            bat 'mvn clean install'
        }
    }
}
stage('Taxe Service SonarQube Analysis'){
    steps{
        dir('taxe-tnb-service'){
            withSonarQubeEnv('sonarqube-server'){
                bat "mvn clean verify sonar:sonar \
-Dsonar.projectKey=taxe-service \
-Dsonar.projectName='taxe-service' \
-Dsonar.host.url=http://localhost:9000 \
-Dsonar.token=sqp_648ed72f5490a9a307f454c583b6769af6de0ad3"
            }
        }
    }
}
}

```

Figure 33 : Contenu du JenkinsFile (partie 4)

```

stage('Terrain Service Build') {
    steps {
        dir('terrain-service') {
            bat 'mvn clean install'
        }
    }
}

stage('Terrain Service SonarQube Analysis'){
    steps{
        dir('terrain-service'){
            withSonarQubeEnv('sonarqube-server'){
                bat "mvn clean verify sonar:sonar \
-Dsonar.projectKey=terrain-service \
-Dsonar.projectName='terrain-service' \
-Dsonar.host.url=http://localhost:9000 \
-Dsonar.token=sqp_5e4c868685001b028f80d8c3301edbbf4fd2b725"
            }
        }
    }
}

stage('Create Docker Compose For KeyCloak') {
    steps {
        script {
            bat "docker compose -f .\\keycloak-compose.yml up -d"
        }
    }
}

```

Figure 34 : Contenu du JenkinsFile (partie 5)

```

stage('Create Docker Compose For KeyCloak') {
    steps {
        script {
            bat "docker compose -f .\\keycloak-compose.yml up -d"
        }
    }
}
stage('Create Docker Compose') {
    steps {
        script {
            bat "docker compose up -d"
        }
    }
}
}
}
}

```

Figure 35 : Contenu du JenkinsFile (partie 6)

## Conclusion :

Ce chapitre centré sur SonarQube en tant qu'outil de qualité du code, souligne l'importance cruciale de maintenir une qualité logicielle constante dans notre projet axé sur les microservices. En intégrant SonarQube dans notre pipeline CI/CD, nous visons la détection proactive de défauts, la mesure de la duplication de code, l'évaluation de la documentation et la surveillance de la couverture des tests. Cette section explore en profondeur les fonctionnalités de SonarQube, mettant en avant son rôle en tant qu'allié essentiel pour garantir des pratiques de développement saines. L'architecture du processus de développement, illustrée avec Git, Docker et Jenkins, démontre une cohésion intégrale. L'optimisation des déclenchements automatiques avec des Webhooks souligne notre approche réfléchie pour renforcer l'efficacité du flux de travail, renforçant ainsi notre engagement envers une qualité de code supérieure et une stabilité accrue pour nos microservices.



## Chapitre 5 : Technologies utilisées et réalisation

### Introduction :

Dans ce deuxième chapitre, nous explorerons en détail les technologies essentielles qui ont été sélectionnées et mises en œuvre dans le cadre de notre projet. Ces choix technologiques ont été délibérément faits pour répondre aux besoins spécifiques de notre solution et garantir une mise en œuvre efficace et performante. Nous examinerons les langages de programmation, les **frameworks**, les outils de gestion, et d'autres éléments technologiques clés qui composent l'infrastructure de notre système. Cette exploration approfondie des technologies utilisées jettera les bases nécessaires pour une compréhension complète du développement, de l'architecture, et du fonctionnement de notre solution.

### 1. Technologies utilisées :

#### 1.1. Keycloak :

**Keycloak** est un produit logiciel open source permettant la connexion unique (single sign-on) avec une gestion des identités et des accès conçus pour les applications et services modernes. Depuis mars 2018, ce projet de la communauté WildFly est sous la direction de Red Hat, qui l'utilise comme projet amont pour leur produit Red Hat Single Sign-On.



Figure 36 : Logo de Keycloak

#### 1.2 Spring Boot :

L'extension **Spring Boot** est une solution de convention plutôt que de configuration de Spring conçue pour faciliter la création d'applications Spring de qualité professionnelle avec une configuration minimale.



*Figure 37 : Logo Spring boot*

### 1.3 Angular :

**Angular** est une plateforme de développement, construite sur TypeScript. En tant que plateforme, Angular comprend :

Un cadre basé sur des composants pour la création d'applications web évolutives.

Une collection de bibliothèques bien intégrées qui couvrent une grande variété de fonctionnalités, notamment le routage, la gestion des formulaires, la communication client-serveur, etc.

Une suite d'outils de développement pour vous aider à développer, construire, tester et mettre à jour votre code.



*Figure 38 : Logo Angular*

### 1.4. Apache Maven :

**Apache Maven** (couramment appelé Maven) est un outil de gestion et d'automatisation de production des projets logiciels Java en général et Java EE en particulier. Il est utilisé pour automatiser l'intégration continue lors d'un développement de logiciel. Maven est géré par l'organisation Apache Software Foundation. L'outil était précédemment une branche de l'organisation Jakarta Project.



Figure 39 : Logo de Apache Maven

#### 1.5. Resilience4j :

**Resilience4j** est une bibliothèque Java légère conçue pour aider les développeurs à construire des systèmes résilients. Elle propose des implémentations de patrons de conception de résilience tels que la tolérance aux pannes, le circuit breaker, le retry, etc. Cette bibliothèque facilite la création d'applications robustes qui peuvent mieux gérer les erreurs, les pannes et les conditions défavorables.



Figure 40 : Logo Resilience4j

#### 1.6. PostgreSQL :

**PostgreSQL** est un système de gestion de base de données relationnelle orienté objet puissant et open source qui est capable de prendre en charge en toute sécurité les charges de travail de données les plus complexes.



Figure 41 : Logo PostgreSQL

## 2. Réalisation du projet :

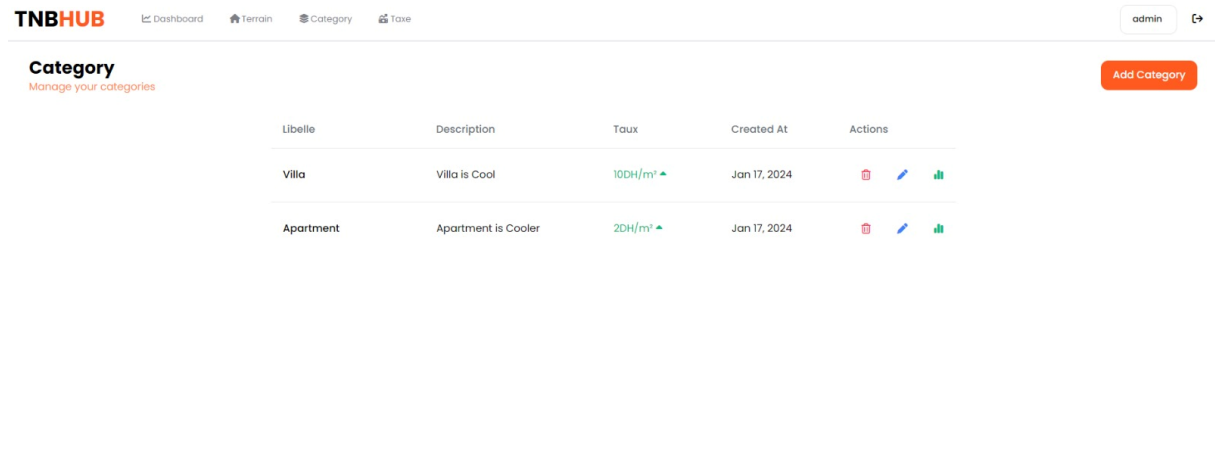


Figure 42 : Interface de catégorie

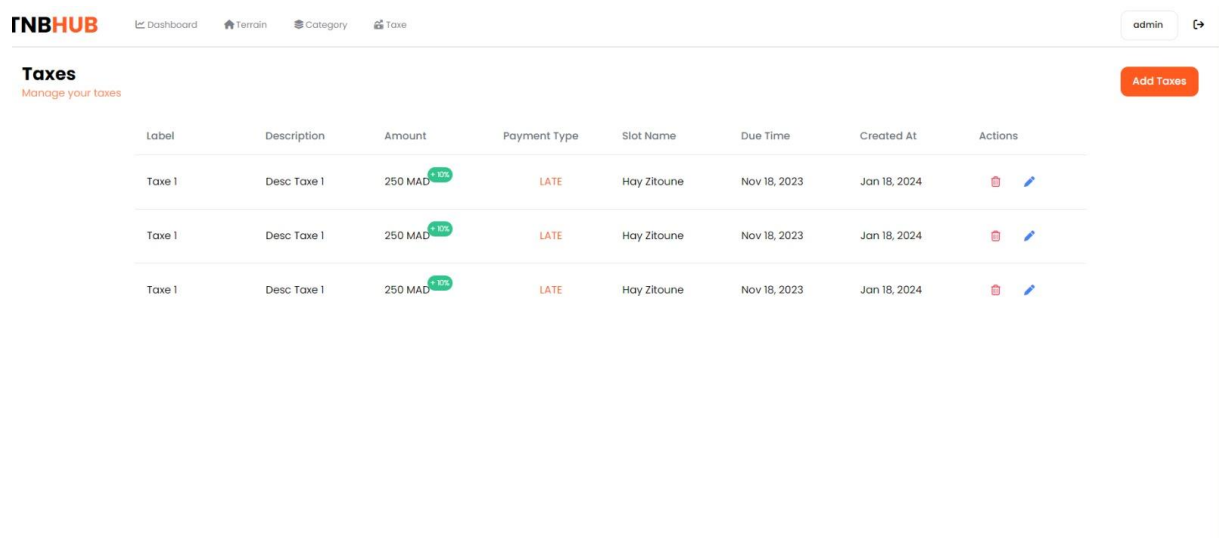


Figure 43 : Interface de taxes

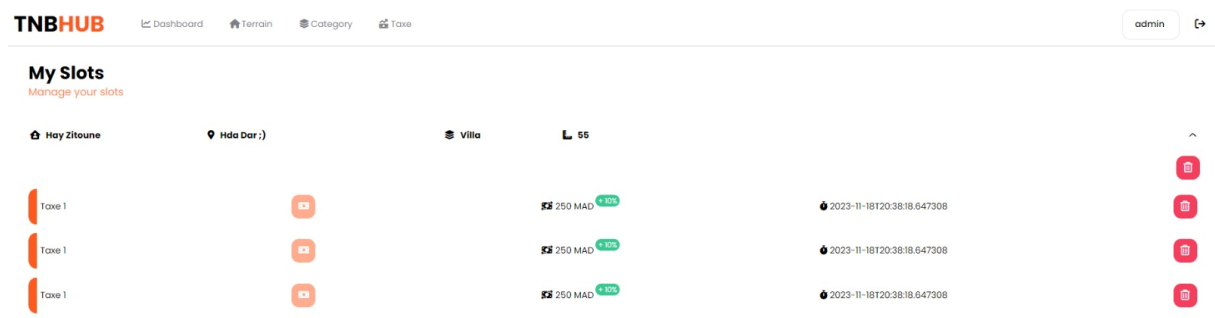


Figure 44 : Interface de terrains

## Conclusion :

En résumé, ce chapitre sur les technologies utilisées a examiné en détail les choix technologiques cruciaux de notre projet. Des langages de programmation aux frameworks en passant par les outils de gestion, chaque décision a été prise pour construire une infrastructure solide. Cette exploration approfondie établit les bases nécessaires pour une mise en œuvre réussie, démontrant notre engagement envers des solutions modernes et performantes. Les prochains chapitres exploiteront cette base pour détailler le développement et l'intégration des fonctionnalités spécifiques de notre solution. En utilisant ces technologies comme fondement, nous sommes prêts à relever les défis pour offrir une solution innovante et robuste.

## Conclusion Générale

Ce projet représente une aventure passionnante à travers la conception, le développement et la mise en œuvre d'une solution technologique répondant à des besoins spécifiques. Tout au long du parcours, nous avons abordé divers aspects, de la définition des exigences à la sélection des technologies, en passant par la mise en œuvre de fonctionnalités clés.

L'architecture microservices, basée sur des technologies modernes telles que Spring Boot, Keycloak, et Docker, a été délibérément choisie pour sa capacité à offrir une extensibilité, une robustesse et une évolutivité maximales. La mise en place de patrons de résilience avec Resilience4j garantit une performance fiable même dans des conditions difficiles.

La gestion d'identité et d'accès à l'aide de Keycloak offre une couche de sécurité solide, tandis que l'utilisation d'Eureka facilite la découverte de services au sein de notre écosystème. L'intégration d'une API Gateway assure un routage efficace des demandes des clients vers les services backend appropriés.

Le chapitre dédié aux technologies utilisées a fourni un éclairage sur les décisions stratégiques prises pour construire une infrastructure technique robuste, mettant en avant notre engagement envers des solutions modernes et performantes.

En synthèse, ce projet a combiné innovation technologique, bonnes pratiques de développement et réponse proactive aux besoins spécifiques. Alors que nous concluons cette phase, notre solution se tient prête à répondre aux défis à venir et à offrir une contribution significative dans le domaine auquel elle est destinée. Ce rapport témoigne de notre parcours, de nos réussites et de notre engagement continu envers l'excellence technologique.