



NOMS DES PARTICIPANTS :

1. ALETANOU Loïc	20P092
2. BEKALE BIKO Yann	20P329
3. BELINGA EKO Yan	20P310
4. DJIFACK Line Audrey	20P309
5. FOTSO NANA	22P136
6. TAZO Belle-Nickelle	20P349

Table des matières

Table des tableaux.....	v
Liste des figures	vi
Abbreviations.....	vii
Résumé.....	viii
Introduction.....	1
Chapitre 1 : Présentation générale du projet	3
1. Contexte et problématique	3
1.1 Contexte	3
1.2 Problématique	3
1.3 Pourquoi une API Gateway est une solution pertinente [2] ?	4
2. Objectifs du projet.....	4
3. Planification du projet.....	5
3.1. Etapes de réalisation	5
3.1.1. Initialisation du Projet.....	5
3.1.2. Implémentation d'Eureka Server	5
3.1.3. Implémentation de l'API Gateway	6
3.1.4. Ajout du Rate Limiter	6
3.1.5. Ajout du Circuit Breaker.....	6
3.1.6. Tests	6
3.2. Diagramme de Gantt	7
3.3. Répartition des tâches	7
4. Revue de la littérature	8
4.1. Généralités sur l'approche microservices	8
4.1.1. Architecture microservice vs architecture monolithique	8
4.1.2 Principes fondamentaux des microservices	9
4.1.3 Avantages des microservices	10
4.1.4 Inconvénients de l'architecture microservice	10
4.2. Définition et Rôle d'une API Gateway	11
4.3. Solutions Existantes	11
4.4. Justification des Choix Techniques pour le Projet.....	12
Chapitre 2 : Analyse et Conception	14
1. Architectures	14

1.1 Architecture hexagonale	14
1.1.1 Acteurs de l'Architecture Hexagonale	14
1.1.2 Principes de l'Architecture Hexagonale	15
1.1.3 Utilité de l'Architecture Hexagonale dans le Contexte de l'API Gateway	15
1.2. Schéma global de l'architecture du projet	16
1.3. Communication entre les différents blocs	18
1.3.1. Interaction entre les clients et l'API Gateway	18
1.3.2. Communication entre l'API Gateway et les Microservices	18
1.3.3. Gestion de la charge avec Load Balancer	18
1.3.4. Enregistrement des Logs et Monitoring	19
Chapitre 3 : Implémentation	21
1. Technologies utilisées	21
1.1. Environnement de développement intégré	21
1.2. Framework et langages de programmation	21
1.3. Dépendances du projet	21
1.3.1. Dépendances springboot	21
1.3.2. Dépendances springcloud	21
1.3.3. Dépendances JSON Web Token	22
1.3.4. Dépendance pour la gestion des objets	22
1.3.5. Dépendance pour les tests	22
2. Configurations et implémentation	23
2.1. Configuration du Projet	23
2.1.1. Initialisation du projet avec Spring Initializr	23
2.1.2. Importation du dossier téléchargé dans l'IDE et mise à jour des dépendances	24
2.1.3. Configuration des différents modules	24
2.2. Architecture interne du système	30
2.3. Développement des modules de l'API Gateway	33
2.3.1. Validation des Paramètres	33
2.3.2 Vérification d'appartenance à la Liste Blanche	33
2.3.3 Transformation de la Requête	34
2.3.4 Service d'Authentification	34
2.3.5 Service de Découverte (Eureka)	34
2.3.6 Routing	34



2.3.7 Cache.....	34
2.3.8 Gestion des Erreurs	35
2.3.9 Circuit Breaker.....	35
2.3.10 Enregistrement des Logs.....	35
3. Tests, validation et résultats	35
Chapitre 4 : Guide de déploiement	42
1. Guide de déploiement pour l'API Gateway	42
Etape 1 : Préparer l'environnement	42
Outils nécessaires :	42
Etape 2 : Construire le projet	42
Construire le projet :	42
Etape 3 : Conteneuriser l'API Gateway avec Docker	42
Modifier le Dockerfile :	42
Etape 4 : Lancer le serveur Eureka	42
Etape 5 : Lancer le serveur de configuration	43
Etape 6 : Lancer l'API Gateway	43
Etape 7 : Lancer le service d'autorisation.....	43
Etape 8 : Vérifier le déploiement	44
2. Limites et perspectives.....	45
2.1. Limites	45
2.2 Perspectives.....	45
Conclusion	47
Références Bibliographique.....	48



Table des tableaux

Tableau 1. Répartition des tâches	7
Tableau 2. Présentation des solutions existantes	11
Tableau 3 . Tableau récapitulatif	22

Liste des figures

Figure 1. Diagramme de Gantt.....	7
Figure 2. Microservices vs Monolithique	8
Figure 3. Architecture hexagonale	14
Figure 4. Architecture globale d'interaction de l'API Gateway	17
Figure 5. Architecture détaillée.....	17
Figure 6 . Initialisation avec spring initializr	23
Figure 7. Service d'authentification	24
Figure 8. Configuration du serveur	27
Figure 9. Serveur Eureka	27
Figure 10. API Gateway	28
Figure 11. Configuration docker.....	29
Figure 12. Architecture du Système.....	30
Figure 13. APIFlow	32
Figure 14. Swagger du service de l'autorisation	36
Figure 15: la fonction check du service d'autorisation	37
Figure 16: la fonction "toggle" du service d'autorisation.....	38
Figure 17. Interface d'Eureka pour la découverte des applications	39
Figure 18. Swagger du service gestion des utilisateurs	39
Figure 19. Gestion des topic Kafka.....	40



Abbreviations

API : Application Programming Interface

IoT : Internet of Things

JWT : JSON Web Token

AWS : Amazon Web Services

HTTP : HyperText Transfer Protocol

CORS : Cross-Origin Resource Sharing

JSON : JavaScript Object Notation

HTML : HyperText Markup Language

Résumé

Ce projet a porté sur la conception et l'implémentation d'une API Gateway robuste, avec une emphase particulière sur la sécurité à travers l'intégration de services d'autorisation et d'identification d'une part et être l'intermédiaire entre les communications microservices vers microservice et clients vers le système. Dans un contexte d'architectures microservices de plus en plus répandues, la nécessité de centraliser et de sécuriser l'accès aux différents services est devenue primordiale. L'API Gateway développée dans ce projet remplit cette fonction en agissant comme un point d'entrée unique pour toutes les requêtes, tout en assurant la gestion de l'authentification et de l'autorisation des différents services.

Pour atteindre cet objectif, une approche modulaire a été adoptée, permettant une séparation claire des responsabilités. Le cœur de l'API Gateway repose sur une architecture basée sur des filtres, offrant une grande flexibilité pour la mise en œuvre de fonctionnalités telles que le routage dynamique des requêtes, la transformation des données et la gestion des erreurs.

En ce qui concerne la sécurité, le projet a intégré un système d'identification robuste, permettant aux utilisateurs de s'authentifier de manière sécurisée et d'obtenir des jetons d'accès pour accéder aux ressources protégées. Un service d'autorisation a également été développé pour contrôler l'accès aux différentes API en fonction des rôles et des permissions des utilisateurs.

Les technologies clés utilisées dans ce projet incluent Spring Cloud Gateway pour la mise en œuvre de l'API Gateway, Spring Security pour la gestion de la sécurité, et un système de stockage de données pour la persistance des informations d'utilisateur et des rôles.

Le projet a été validé à travers une série de tests unitaires et d'intégration, garantissant le bon fonctionnement de l'API Gateway et de ses services de sécurité. Les résultats ont démontré la capacité de l'API Gateway à gérer efficacement les requêtes, à assurer la sécurité des accès et à s'intégrer de manière transparente avec les autres composants du système.

En conclusion, ce projet a permis de développer une API Gateway robuste et sécurisée, offrant une solution centralisée pour la gestion des accès dans une architecture de microservices. L'intégration des services d'identification et d'autorisation garantit la protection des ressources et la conformité aux exigences de sécurité.

Introduction

Dans un contexte où les architectures microservices sont de plus en plus adoptées, la gestion des interactions entre les clients et les différents services devient un enjeu majeur. Une API Gateway joue un rôle central en servant de point d'entrée unique pour toutes les requêtes, assurant ainsi la gestion du routage, de l'authentification, de la sécurité et du monitoring des services.

L'objectif de ce projet est de concevoir et implémenter une API Gateway afin de centraliser les requêtes des clients vers divers microservices. Cela permettra d'améliorer la scalabilité, la sécurité et la maintenabilité de l'architecture globale du système.

Pour atteindre cet objectif, nous avons choisi d'utiliser Spring Cloud Gateway, qui offre une solution flexible et performante pour la gestion des API. L'enregistrement et la découverte des services seront assurés par Eureka, tandis que Spring Boot servira de base pour le développement. Enfin, Kafka sera utilisé pour la communication asynchrone avec les microservices afin d'optimiser la transmission des événements.

Ce rapport détaille la conception, l'implémentation et l'évaluation de notre API Gateway, en mettant en avant l'architecture générale de notre API Gateway suivi des différents modules qui la compose et les étapes de développement.



Chapitre 1 : PRESENTATION GENERALE DU PROJET

Chapitre 1 : Présentation générale du projet

1. Contexte et problématique

1.1 Contexte

Dans le cadre de la Plateforme IoT Agricole, l'API Gateway joue un rôle central en tant que point d'entrée unique pour toutes les requêtes provenant des clients (applications web, mobiles, IoT, etc.). Étant donné que la plateforme intègre plusieurs microservices (gestion des agriculteurs, marketplace, analyse de données IoT, blockchain, etc.), l'API Gateway est essentielle pour :

- Centraliser le routage des requêtes : Diriger les requêtes vers les microservices appropriés en fonction des besoins (ex : requêtes vers le service de gestion des agriculteurs, le service de paiement, ou le service d'analyse de données IoT).
- Gérer la sécurité : Assurer l'authentification et l'autorisation des utilisateurs (agriculteurs, intermédiaires, consommateurs) avant de permettre l'accès aux microservices.
- Optimiser les performances : Implémenter des mécanismes de mise en cache, de limitation de débit (rate limiting), et de gestion des erreurs pour garantir une expérience utilisateur fluide.
- Faciliter le monitoring : Fournir des outils de logging pour suivre les performances et diagnostiquer les problèmes rapidement.

1.2 Problématique

En l'absence d'une API Gateway, la gestion des communications entre les clients et les microservices deviendrait complexe et inefficace. Chaque microservice devrait gérer individuellement des fonctionnalités transversales comme la sécurité, la gestion des erreurs, et le routage, ce qui entraînerait des redondances et des risques accrus de failles de sécurité [1]. Ci-dessous sont présentés des problèmes majeurs auxquels les développeurs et les architectes sont confrontés :

- Complexité du routage : Dans une architecture microservices, les clients doivent souvent interagir avec plusieurs services pour accomplir une tâche. Sans une API Gateway, les clients doivent connaître les adresses de chaque microservice, ce qui complique la gestion des requêtes et rend le système moins flexible.

- Sécurité : Chaque microservice doit gérer individuellement l'authentification et l'autorisation, ce qui peut entraîner des failles de sécurité et des redondances.
- Performance : La gestion des erreurs, la mise en cache, et la limitation de débit doivent être implémentées de manière cohérente pour garantir une expérience utilisateur fluide.
- Découverte de services : Dans un environnement dynamique où les services peuvent être déployés ou mis à jour fréquemment, il est essentiel de localiser les services disponibles.

1.3 Pourquoi une API Gateway est une solution pertinente [2] ?

Une API Gateway répond à ces problématiques en offrant une solution centralisée pour la gestion des microservices. Elle agit comme un point d'entrée unique, routant les requêtes vers les microservices appropriés sans que les clients aient besoin de connaître les adresses de chaque service. Elle gère l'authentification et l'autorisation de manière centralisée, réduisant les risques de failles de sécurité et les redondances, par exemple l'utilisation de tokens JWT (JSON Web Tokens) pour sécuriser les communications entre le client et les microservices. Enfin, elle intègre des registres de services comme Consul ou Eureka pour une découverte dynamique des services.

2. Objectifs du projet

Les objectifs spécifiques à l'API Gateway sont les suivants :

- Centralisation du routage : Implémenter un mécanisme de routage intelligent pour diriger les requêtes vers les microservices appropriés.
- Gestion de la sécurité : Implémenter des mécanismes d'authentification et d'autorisation centralisés (ex : OAuth2, JWT).
- Optimisation des performances :
 - Mettre en place des mécanismes de mise en cache pour réduire la latence et la charge sur les microservices.
 - Implémenter une limitation de débit (rate limiting) pour éviter la surcharge des services.
- Gestion des erreurs et résilience :
 - Implémenter des mécanismes de gestion des erreurs pour garantir que les défaillances des microservices n'affectent pas l'ensemble du système.
 - Prendre en charge des patterns de résilience tels que le circuit breaker et les retries.
- Monitoring et logging :

- Fournir des outils de monitoring pour suivre les performances des API en temps réel.
- Implémenter un système de logging centralisé pour faciliter le diagnostic des problèmes.
- Scalabilité et haute disponibilité :
 - Concevoir l'API Gateway pour qu'elle soit scalable et capable de gérer une augmentation du trafic.
 - Assurer une haute disponibilité grâce à des mécanismes de réplication et de load balancing.

3. Planification du projet

3.1. Etapes de réalisation

3.1.1. Initialisation du Projet

➤ Configuration de Spring Boot et des Dépendances

- Objectifs :
 - Créer une nouvelle application Spring Boot.
 - Configurer les dépendances nécessaires, notamment Spring Cloud pour la gestion des microservices.
- Tâches :
 - Création du projet via Spring Initializr.
 - Ajout des dépendances dans le fichier pom.xml :
 - Spring-boot-starter-web
 - Spring-cloud-starter-netflix-eureka-server
 - Spring-cloud-starter-gateway
 - Spring-cloud-starter-circuitbreaker-resilience4j
 - Spring-boot-starter-data-redis

3.1.2. Implémentation d'Eureka Server

➤ Mise en Place du Serveur de Découverte

- Objectifs :
 - Installer et configurer Eureka Server pour permettre la découverte des services.
- Tâches :
 - Annoter la classe principale avec @EnableEurekaServer.
 - Configurer les propriétés dans application.yml pour définir le port et le contexte de l'application.
 - Tester la découverte des services en ajoutant des microservices sous Eureka.

3.1.3. Implémentation de l'API Gateway

- Configuration du Routage et des Filtres
- Objectifs :
 - Mettre en place un API Gateway pour centraliser les appels aux microservices.
- Tâches :
 - Créer une classe de configuration pour définir les routes.
 - Configurer les filtres pour la gestion des requêtes et des réponses.
 - Tester le routage des requêtes vers différents microservices.

3.1.4. Ajout du Rate Limiter

- Intégration et Configuration de Redis Rate Limiter
- Objectifs :
 - Implémenter un système de limitation de débit pour protéger les microservices contre les surcharges de trafic.
- Tâches :
 - Ajouter la dépendance Redis.
 - Configurer le Rate Limiter dans l'API Gateway.

3.1.5. Ajout du Circuit Breaker

- Mise en Place avec Resilience4J
- Objectifs :
 - Implémenter un Circuit Breaker pour gérer les pannes des microservices et améliorer la résilience de l'application.
- Tâches :
 - Configurer Resilience4J dans l'API Gateway.
 - Définir les règles de seuil pour l'ouverture et la fermeture du Circuit Breaker.

3.1.6. Tests

- Vérification des Fonctionnalités
- Objectifs :
 - Assurer que toutes les fonctionnalités sont opérationnelles avant le déploiement.

3.2. Diagramme de Gantt

Présentation du diagramme de Gantt :

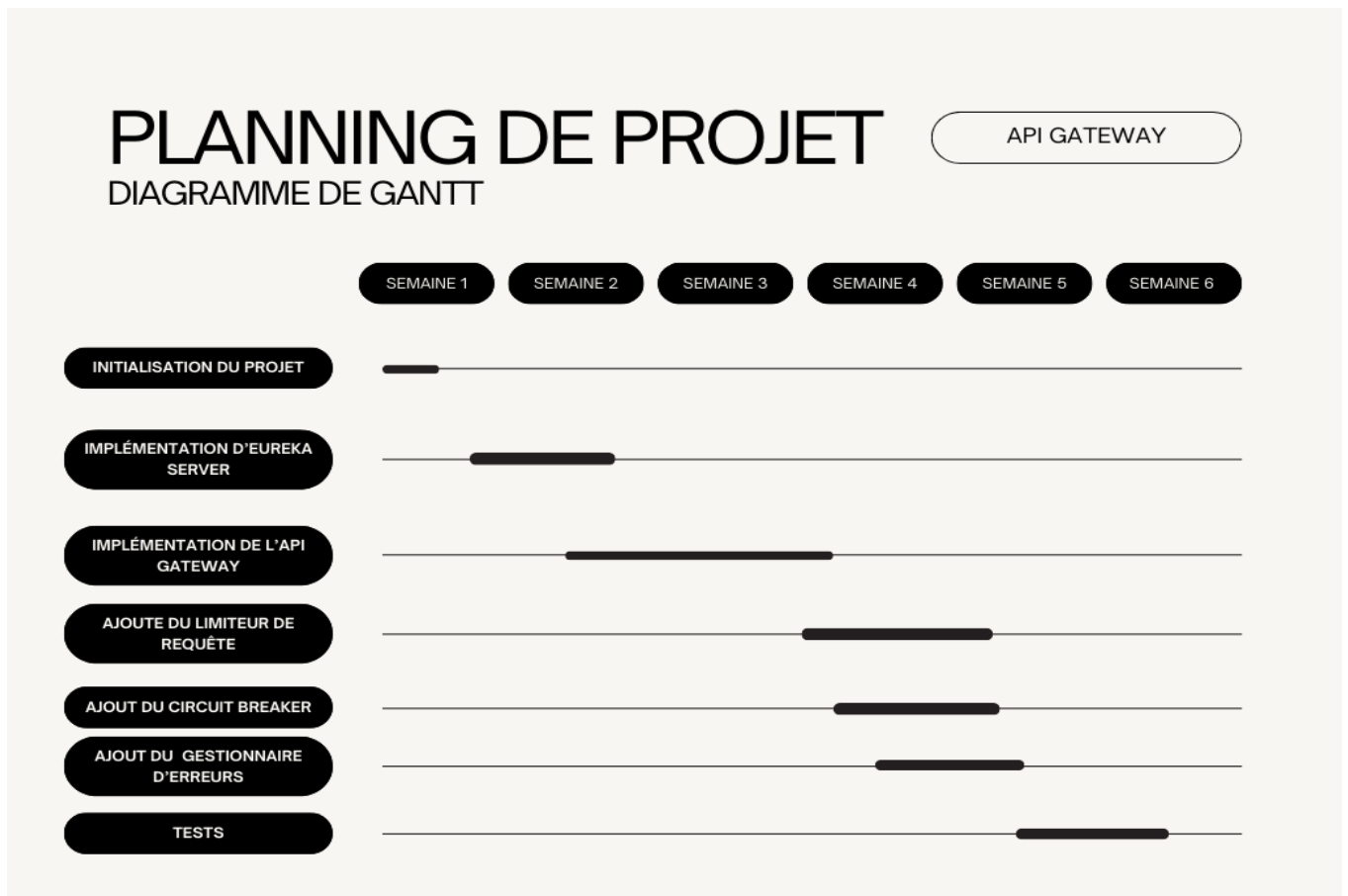


Figure 1. Diagramme de Gantt

3.3. Répartition des tâches

Présentation des répartitions des tâches :

Tableau 1. Répartition des tâches

Tâches	Responsables	Echéance
Initialisation du Projet	TAZO Belle	2 Jours
Implémentation d'Eureka Server	DJIFACK Line, ALETANOU	2 Semaines
Implémentation de l'API Gateway	FOTSO NANA	3 Semaines
Ajout du Rate Limiter et	BELINGA	1 Semaine

du Circuit Breaker		
Ajout du Gestionnaire d'erreurs	BEKALE, TAZO Belle	2 Semaine
Tests	TOUT LE MONDE	1 Semaine

4. Revue de la littérature

Dans cette section, nous explorons les travaux antérieurs, les technologies existantes, et les approches adoptées par d'autres projets similaires afin de justifier les choix techniques et architecturaux pour le projet de la Plateforme IoT Agricole.

4.1. Généralités sur l'approche microservices

L'architecture de microservices (ou microservices) désigne un style d'architecture utilisé dans le développement d'applications. Elle permet de décomposer une application volumineuse en composants indépendants, chaque élément ayant ses propres responsabilités.[3]

4.1.1. Architecture microservice vs architecture monolithique

Une architecture de microservices permet d'apporter des modifications à des unités métier individuelles tout en maintenant une fonctionnalité cohérente. En revanche, dans une approche monolithique, toutes les fonctionnalités sont déployées ensemble dans un seul processus, nécessitant également des modifications dans d'autres couches. Dans une architecture de microservices, les composants fonctionnent indépendamment et ont des limites de service établies, ce qui réduit le risque de défaillance en isolant rapidement les problèmes.

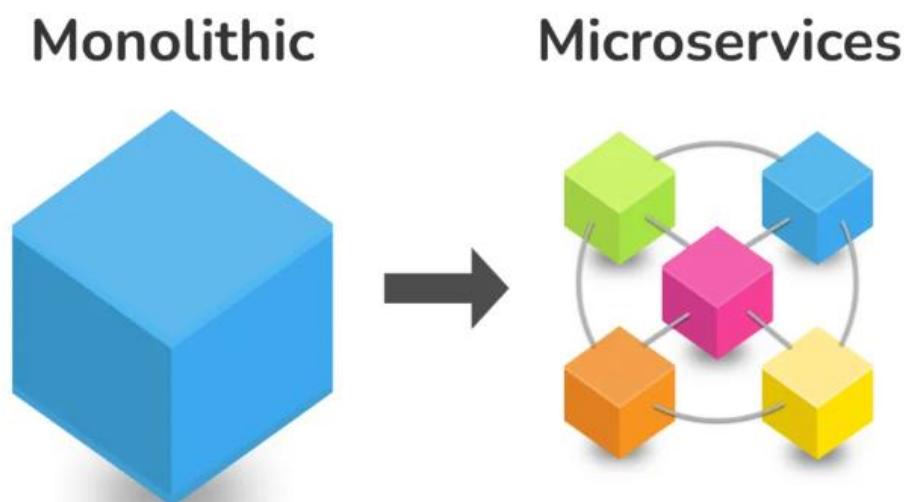


Figure 2. Microservices vs Monolithique

4.1.2 Principes fondamentaux des microservices

4.1.2.1. Communication entre microservices

La communication entre microservices a un style ou un mélange selon le contexte opérationnel spécifique et les besoins, la latence, la sécurité et le volume (charges utiles).

- **Communication asynchrone (courtier de messages)** : permet aux microservice de maintenir plusieurs communications simultanément. Une requête est envoyée sous forme de message (courtier de messages) qui est conservé dans la file d'attente jusqu'à ce qu'il soit consommé (traité) ; il répond lorsqu'il est prêt et route ensuite une réponse. Cela permet au microservice de gérer plusieurs requêtes, tant qu'il peut associer (état associé) la requête à l'expéditeur d'origine. La gestion de la logique de communication interne nécessite de construire du code.
- **Communication synchrone (appels de requête-réponse)** : Il s'agit d'une communication bidirectionnelle utilisée avec des microservices simples qui communiquent (requête) dans la même connexion réseau et des instances qui établissent une connexion réseau (envoient un appel) directement au microservice et attendent sa réponse. Dans les connexions synchrones, un faible couplage est impliqué. De nombreuses connexions indépendantes simultanées entre les appels de réseau de microservices pourraient submerger le système, déclenchant des problèmes créant des congestions ou des défaillances.

4.1.2.2. Caractéristiques principales

- **Indépendance et modularité** : chaque microservice est un composant autonome qui peut être développé, déployé et mis à jour indépendamment des autres
- **Spécialisation métier** : chaque service gère un domaine métier bien défini, ce qui facilite la compréhension et la maintenance
- **Communication via API** : les microservices interagissent généralement via des API REST, GraphQL ou des messages asynchrones via Kafka etc.
- **Évolutivité** : chaque microservice peut être mis à l'échelle indépendamment des autres, optimisant ainsi l'utilisation des ressources
- **Résilience et tolérance aux pannes** : en cas de panne d'un microservice, le reste du système continue à fonctionner avec des mécanismes de callback ou de circuit breaker.

4.1.2.3. Modèles d'architecture de microservices

- **Request-response** : Le modèle de requête-réponse fonctionne en envoyant une requête à un autre microservice pour effectuer une action, puis en recevant une réponse dans une approche synchrone ou asynchrone. Ce modèle de microservices est utilisé lorsqu'une réponse est requise avant tout traitement ultérieur. Il y a un couplage élevé, car l'expéditeur connaît la fonctionnalité du destinataire (couplage de domaine).

- **Pilotage d'événements** : L'interaction pilotée par les événements est une interaction asynchrone dans laquelle un événement se produit ou est émis par un microservice qui est requis et sera utilisé par un autre microservice. Ces interactions sont faiblement couplées (interdépendances réduites) car l'émetteur ne connaît pas l'intention du destinataire.
- **Données communes (fichiers de données – système de fichiers)** : L'entreposage de données et les lacs de données fonctionnent avec ce modèle, tirant des données d'une direction à une autre partie qui lit ou écrit le fichier ou la base de données. En conséquence, ce modèle de microservices peut gérer de grands volumes de données et inclut l'interopérabilité avec des systèmes hérités.

4.1.2.4. Protocoles et technologies d'implémentation

La mise en œuvre des technologies pour une architecture de microservices nécessite une observation attentive de la compatibilité.

- HTTP
- L'API de microservices
- Courtiers de messages (middleware) : C'est un mécanisme qui traite les événements asynchrones (publication et abonnement) en utilisant du middleware (Apache Kafka). Kafka est souvent utilisé pour des interactions basées sur des événements, diffusant de grandes quantités de données en temps réel.
- **Base de données** : Apache Cassandra, MongoDB, ElasticSearch, PostgreSQL.
- Technologie de système de fichiers

4.1.3 Avantages des microservices

- **Flexibilité** : Les équipes peuvent travailler sur différents services simultanément sans interférer les uns avec les autres.
- **Meilleure gestion des erreurs** : Les erreurs dans un service n'affectent pas nécessairement l'ensemble de l'application.
- **Déploiements flexibles** : permet l'intégration continue et les mises à jour fréquentes sans interruption.
- **Facilité de maintenance** : chaque service étant indépendant, les équipes peuvent travailler en parallèle sans interférences.
- **Technologies variées** : chaque microservice peut être développé avec le langage et la technologie les plus adaptés à son besoin.

4.1.4 Inconvénients de l'architecture microservice

- **Complexité accrue** : la gestion des services, des communications et des données devient plus difficile
- **Surcoût en infrastructure** : chaque microservice nécessitant des ressources dédiées, cela peut entraîner des coûts supplémentaires

- Débogage et monitoring plus difficiles : il est nécessaire de mettre en place des outils avancés comme Grafana pour observer l'état du système
- Problèmes de latence : la communication entre services peut entraîner des retards si elle n'est pas bien optimisée.

4.2. Définition et Rôle d'une API Gateway

Une API Gateway est un composant central dans les architectures microservices. Elle agit comme un point d'entrée unique pour toutes les requêtes clientes, les routant vers les microservices appropriés [4] tout en fournissant des fonctionnalités transversales telles que la sécurité, la gestion des erreurs, et le monitoring. Selon Richardson (2018), une API Gateway est essentielle pour :

- Simplifier le routage : Les clients n'ont pas besoin de connaître les adresses des microservices individuels.
- Centraliser la sécurité : L'authentification et l'autorisation sont gérées de manière centralisée, réduisant les risques de failles de sécurité.
- Améliorer les performances : La mise en cache et la limitation de débit sont implémentées de manière cohérente.

4.3. Solutions Existantes

Plusieurs solutions d'API Gateway sont disponibles sur le marché, chacune avec ses avantages et ses limites. Nous avons fait une analyse des principales solutions :

Tableau 2. Présentation des solutions existantes

Solutions d'API Gateway	Kong Gateway	Apigee (Google Cloud)	AWS API Gateway	Spring Cloud Gateway
Avantages	Open source, extensible via des plugins, supporte la mise en cache, la limitation de débit et l'authentification OAuth.	Plateforme cloud complète avec des outils d'analyse et de monitoring avancés.	Intégration native avec les services AWS, supporte la mise en cache et la limitation de débit.	Basé sur Spring Boot, facile à intégrer avec d'autres services Spring, supporte les filtres personnalisés.
Limites	Nécessite une configuration complexe pour les environnements distribués.	Coût élevé pour les petites et moyennes entreprises.	Verrouillage avec l'écosystème AWS, coût potentiellement élevé pour un trafic important.	Nécessite une expertise en Java et Spring.



Cas d'utilisation	Utilisé par des entreprises comme Uber et Sony pour gérer des millions de requêtes par jour [5].	Adopté par des entreprises comme Walmart et Burger King.[6]	Utilisé par Netflix et Airbnb [7]	Populaire dans les environnements Java pour les applications d'entreprise.
-------------------	--	---	-----------------------------------	--

4.4. Justification des Choix Techniques pour le Projet

Étant donné que $\frac{3}{4}$ des exemples d'API citées plus haut sont toutes coûteuses et déjà entièrement développées, cela entraîne une certaine dépendance vis-à-vis du concepteur.

De ce fait, pour la Plateforme IoT Agricole, nous opterons pour **Spring Cloud Gateway** : Choisi pour sa compatibilité avec l'écosystème Spring Boot, sa flexibilité, et sa capacité à gérer des fonctionnalités avancées comme le routage, la sécurité, et la résilience. Nous pourrions donc facilement la maintenir et assurer son évolutivité.



Chapitre 2 : ANALYSE ET CONCEPTION

Chapitre 2 : Analyse et Conception

1. Architectures

1.1 Architecture hexagonale

L'architecture hexagonale, également connue sous le nom d'architecture ports et adaptateurs, est un modèle de conception logicielle qui vise à séparer la logique métier des préoccupations techniques et des détails d'implémentation. Ce modèle permet de créer des applications modulaires, maintenables et évolutives, facilitant ainsi l'intégration avec différents types de clients et de services externes [8].

1.1.1 Acteurs de l'Architecture Hexagonale

Dans ce modèle, trois acteurs principaux interagissent : le User-Side, la Business Logic et le Server-Side. Le User-Side représente la façade de l'application, là où les utilisateurs ou les systèmes externes se connectent. Il englobe le code d'interface utilisateur, les routes API et les mécanismes de communication, facilitant ainsi les interactions avec l'application.

Au cœur de l'architecture se trouve la Business Logic, qui encapsule la logique métier et les règles spécifiques à l'application. Cette partie est conçue pour être indépendante des détails d'implémentation, ce qui permet une meilleure lisibilité et une maintenance facilitée. Enfin, le Server-Side gère les interactions avec l'infrastructure, incluant la gestion des bases de données, des fichiers et des appels à d'autres services. Il est piloté par la logique métier, garantissant ainsi une séparation claire des préoccupations.

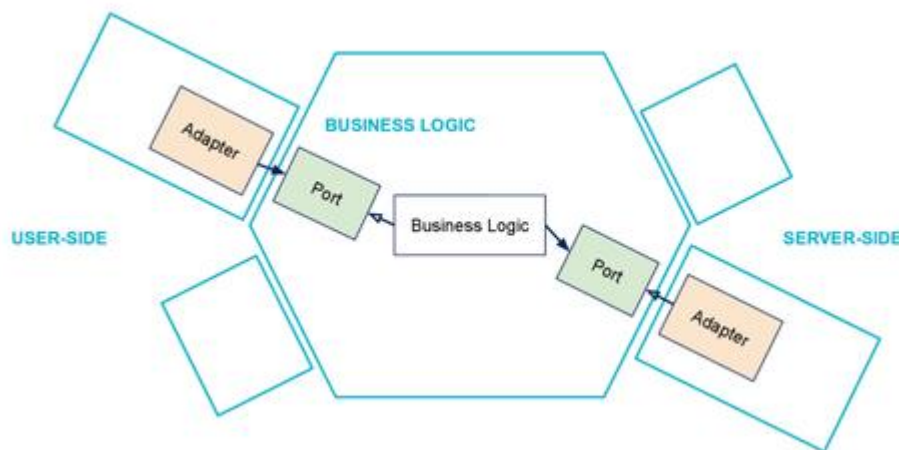


Figure 3. Architecture hexagonale

Le schéma ci-dessus illustre comment ces trois acteurs interagissent au sein de l'architecture hexagonale, mettant en évidence l'importance des ports et des adaptateurs pour assurer une communication fluide entre les différentes couches de l'application.

1.1.2 Principes de l'Architecture Hexagonale

L'architecture hexagonale repose sur des principes fondamentaux qui favorisent la modularité et la maintenabilité des applications. Ces principes permettent de structurer l'application de manière à isoler les différentes préoccupations et à faciliter les interactions entre les composants.

La **séparation des zones** est l'un des principes clés, où les trois zones — **User-Side**, **Business Logic** et **Server-Side** — sont clairement définies. Cette distinction permet une meilleure compréhension du système et une isolation efficace des problèmes.

Un autre principe important est celui des **dépendances vers l'intérieur**. Dans ce modèle, toutes les dépendances doivent diriger vers la Business Logic. Cela garantit que les côtés User-Side et Server-Side dépendent de la logique métier, tout en évitant que cette dernière soit influencée par des détails d'implémentation externes.

Enfin, **l'isolement par interfaces** est essentiel pour structurer les interactions entre les différentes zones. Les communications se font à travers des interfaces, également appelées ports, qui agissent comme des barrières protectrices entre l'intérieur (Business Logic) et l'extérieur (User-Side et Server-Side). Ce mécanisme renforce l'indépendance de la logique métier et facilite les tests et les évolutions de l'application.

1.1.3 Utilité de l'Architecture Hexagonale dans le Contexte de l'API Gateway

L'architecture hexagonale, en conjonction avec l'API Gateway, centralise l'accès à l'application. L'API Gateway agit comme un point d'entrée unique pour tous les clients, simplifiant ainsi la gestion des requêtes et la communication entre les clients et les services backend.

De plus, l'API Gateway permet une gestion efficace des responsabilités. Elle peut gérer **l'authentification** et **l'autorisation**, équilibrer la charge entre les services, et transformer les requêtes et réponses pour assurer une communication fluide.

L'architecture hexagonale facilite également l'intégration de services externes ou d'applications tierces. Grâce à la définition claire des ports, il est possible d'ajouter ou de modifier des adaptateurs sans perturber la logique métier.

Voici comment l'architecture présentée ci-dessus peut répondre au modèle architectural de l'API gateway de la figure 4 :

- **Les composants de l'architecture hexagonale [9] :**

Elle compose de plusieurs éléments clés qui interagissent pour créer un système robuste et flexible, à savoir :

Ports : Les ports représentent les interfaces par lesquelles les interactions avec le système se font. Dans notre cas, l'**API Gateway** agit comme un port, permettant aux clients (front-end et services tiers) d'interagir avec le backend.

Adaptateurs : Les adaptateurs sont les implémentations concrètes des ports. Par exemple, le **Load Balancer** et les **services de backend** agissent comme des adaptateurs qui répondent aux requêtes envoyées via l'API Gateway.

Centre de l'architecture : La logique métier et les règles de l'application se trouvent au centre, séparées des détails d'implémentation. Il s'agit de la logique des différents micro services : IOT service, ML service, Notification service...)

➤ L'interaction avec les systèmes externes :

Elle se présente comme-suit :

Frontend et services tiers : Les applications front-end et les intégrations tierces communiquent avec le backend via des appels réseau, ce qui est conforme à l'idée de ports permettant des interactions externes.

Eureka et configuration : **Eureka** en tant que service de découverte permet aux différentes parties de l'application de se trouver et de communiquer entre elles sans être directement couplées, ce qui est un aspect fondamental de l'architecture hexagonale.

Le schéma de l'application :

Pour visualiser cela dans un schéma hexagonal :

Côté extérieur : Incluez le **frontend** et les **services tiers**.

Ports : Représentez l'API Gateway comme un port.

Adaptateurs : Les services de backend et le Load Balancer doivent être indiqués comme adaptateurs.

Logique métier : Au centre, placez les services ou microservices qui contiennent la logique métier (comme **User Service**, **Payment Service**, etc.).

L'architecture hexagonale, combinée à l'API Gateway, offre une approche robuste et flexible pour le développement d'applications modernes. Elle répond aux besoins croissants de modularité, de maintenabilité et de scalabilité dans un environnement de développement dynamique.

1.2. Schéma global de l'architecture du projet

L'architecture de notre API Gateway repose sur l'interconnexion de plusieurs composants qui travaillent ensemble pour assurer un acheminement optimal des requêtes entre les clients et les microservices.

Présenté ci-dessous l'architecture globale de l'application : **Plateforme IoT Agricole Web et Mobile multilingue et multidevise**. L'architecture suit un schéma où les requêtes

des clients transitent par plusieurs couches avant d'être traitées par les microservices. Le schéma illustratif est le suivant :

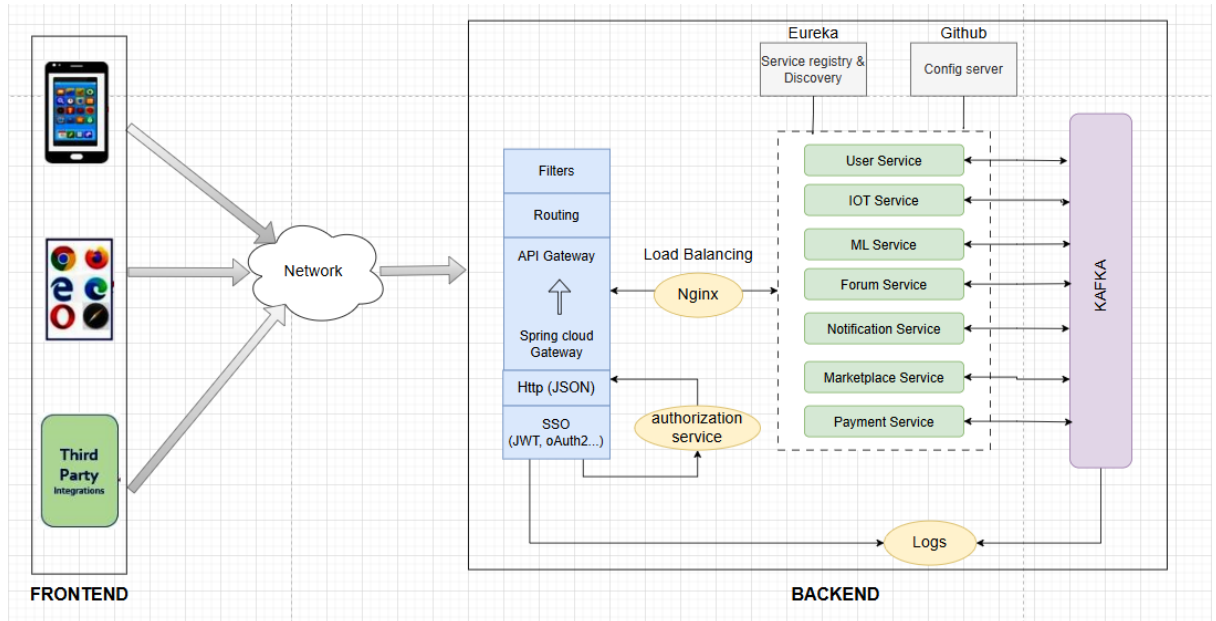


Figure 4. Architecture globale d'interaction de l'API Gateway

Nous avons par la suite ressorti les différents modules intervenant dans notre API Gateway afin de présenter une vue plus approfondie de ce qu'elle est censée faire :

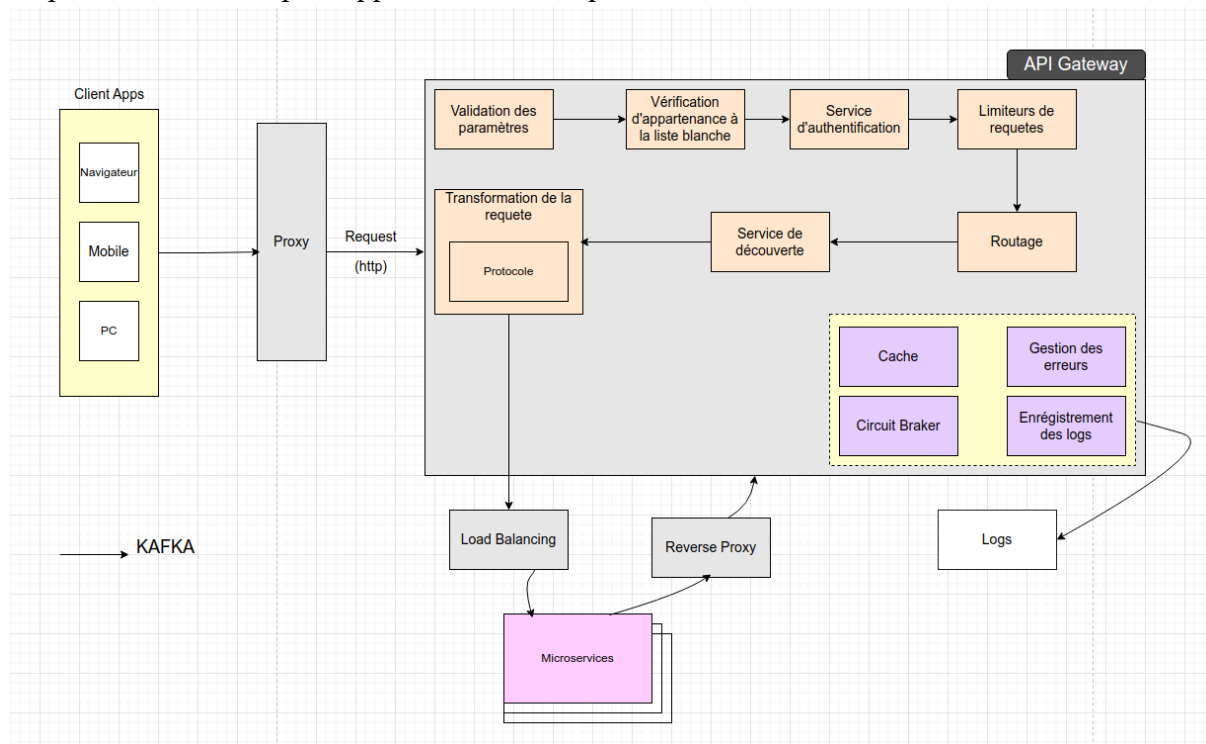


Figure 5. Architecture détaillée

Dans cette architecture, une requête provenant d'un microservice passe successivement par les modules suivants :

- Le module de validation des paramètres :
- Le module de l'authentification :
- Le module de l'autorisation :
- Le limiteur de requêtes :
- Le module de découverte des services :

1.3. Communication entre les différents blocs

1.3.1. Interaction entre les clients et l'API Gateway

Lorsqu'une application cliente, qu'il s'agisse d'un site web, d'une application mobile ou d'un service tiers, envoie une requête, celle-ci est d'abord dirigée vers un proxy. Ce dernier agit comme un premier filtre permettant d'optimiser le trafic, de gérer la mise en cache et d'assurer une protection contre les attaques. Une fois la requête validée par le proxy, elle est transmise à l'API Gateway, qui intercepte la requête et effectue plusieurs validations : vérification du format des paramètres, contrôle des droits d'accès et gestion des quotas d'utilisation.

Si la requête est conforme, l'API Gateway interroge Eureka Client pour récupérer l'adresse du microservice cible.

1.3.2. Communication entre l'API Gateway et les Microservices

Lorsqu'un client envoie une requête à l'API Gateway, celle-ci ne l'achemine pas directement à un microservice spécifique. Au lieu de cela, elle publie la requête dans un topic Kafka dédié, qui agit comme une file d'attente persistante. Cette approche permet d'absorber les pics de charge et d'éviter la surcharge immédiate des microservices.

Les microservices concernés ne reçoivent pas les requêtes directement, mais les consomment à partir du topic Kafka de manière asynchrone. Chaque microservice abonné au topic récupère les messages en fonction de sa capacité de traitement, ce qui permet une gestion efficace des ressources et un équilibrage dynamique de la charge. Une fois la requête traitée, le microservice génère une réponse qui n'est pas envoyée immédiatement à l'API Gateway. Au lieu de cela, la réponse est publiée dans un autre topic Kafka dédié aux réponses.

L'API Gateway, qui est elle-même un consommateur du topic des réponses, surveille en permanence les nouvelles réponses disponibles. Lorsqu'une réponse est publiée par un microservice, elle est récupérée par l'API Gateway, qui l'enrichit si nécessaire, puis la renvoie au client initial.

1.3.3. Gestion de la charge avec Load Balancer

Pour éviter qu'un microservice unique ne soit submergé par un trop grand nombre de requêtes, un Load Balancer intervient afin de répartir équitablement la charge entre plusieurs instances du même service. Cette répartition permet non seulement d'améliorer la disponibilité des



services, mais aussi d'assurer une meilleure réactivité aux variations de trafic. En cas de panne d'une instance, le Load Balancer redirige automatiquement les requêtes vers une autre instance fonctionnelle, garantissant ainsi la continuité du service.

1.3.4. Enregistrement des Logs et Monitoring

Toutes les interactions au sein de cette architecture sont enregistrées dans un système de logs afin d'assurer un suivi précis des requêtes et des réponses. Ces logs permettent d'analyser le fonctionnement du système, de détecter d'éventuelles anomalies et d'optimiser les performances globales.

A large, dark blue ribbon banner with a 3D effect, featuring a central rectangular section and two flared ends. The text is centered on the central section.

Chapitre 3 : IMPLEMENTATION

Chapitre 3 : Implémentation

1. Technologies utilisées

1.1. Environnement de développement intégré

Nous avons utilisé VsCode comme environnement de développement intégré. Il s'agit d'un éditeur de code extensible développé par Microsoft pour Windows, Linux et MacOS. Il peut être utilisé pour de nombreux langages de programmation notamment Java, JavaScript, Go, Node.js...

1.2. Framework et langages de programmation

Notre API est faite à l'aide du Framework springboot et donc utilise le langage de programmation JAVA. Springboot permet de créer des applications spring autonomes, fournit des dépendances de démarrage avisées pour simplifier la configuration du build. Permet également de configurer automatiquement Spring et les bibliothèques tierces chaque fois que possible.

1.3. Dépendances du projet

1.3.1. Dépendances springboot

- **Spring Boot Starter WebFlux**

Fournit un support pour les applications Web réactives en utilisant Spring WebFlux, basé sur Project Reacto, permettant ainsi de construire des API réactives et asynchrones.

- **Spring Boot Starter OAuth2 Resource Server**

Implémente un serveur de ressources OAuth2 qui protège les API avec des tokens JWT, opaque, ou introspection, permettant de sécuriser les endpoints API avec OAuth2.

- **Spring Boot Starter Actuator**

Fournit des endpoints pour monitorer et gérer l'application (métriques, état de santé, journaux...), essentiel pour l'observabilité et l'administration d'une application.

1.3.2. Dépendances springcloud

- **Spring Cloud Gateway**

Implémente un API Gateway pour rediriger et sécuriser le trafic vers les microservices, permet la gestion des routes, load balancing, authentification, et transformations des requêtes.

- **Spring Cloud Netflix Eureka Client**

Permet l'enregistrement et la découverte des services via Eureka, un service de Service Discovery de Netflix. Les microservices peuvent donc s'enregistrer dynamiquement et être découverts sans configuration manuelle.

1.3.3. Dépendances JSON Web Token

- **JWT API, Impl et Jackson**

Gère les JSON Web Tokens (JWT) pour l'authentification et l'autorisation. Permet de signer, parser et valider des tokens JWT.

1.3.4. Dépendance pour la gestion des objets

- **Lombok**

Génère automatiquement des getters, setters, constructeurs et autres méthodes utiles. Évite d'écrire du code répétitif et améliore la lisibilité.

1.3.5. Dépendance pour les tests

- **Spring-boot-starter-test**

Fournit une suite complète d'outils pour tester l'application (JUnit, Mockito, Spring Boot Test...). Permet d'écrire des **tests unitaires et d'intégration**.

Tableau 3 . Tableau récapitulatif

Dépendance	Utilité
Spring Boot Starter WebFlux	Développer des applications web réactives
Spring Boot Starter OAuth2 Resource Server	Sécuriser les API avec OAuth2
Spring Boot Starter Actuator	Monitorer et administrer l'application
Spring Cloud Gateway	Gérer un API Gateway
Spring Cloud Netflix Eureka Client	Découverte dynamique des services
Spring Cloud Netflix Eureka Service	Maintient une liste des services disponibles
Jjwt-api, jjwt-impl, jjwt-jackson	Gérer les tokens JWT
Lombok	Réduire le code répétitif
Spring Boot Starter Test	Tester l'application

Spring Cloud Dependencies	Gérer les versions des dépendances Spring Cloud
Spring Boot Maven Plugin	Compiler et exécuter l'application avec Maven

2. Configurations et implémentation

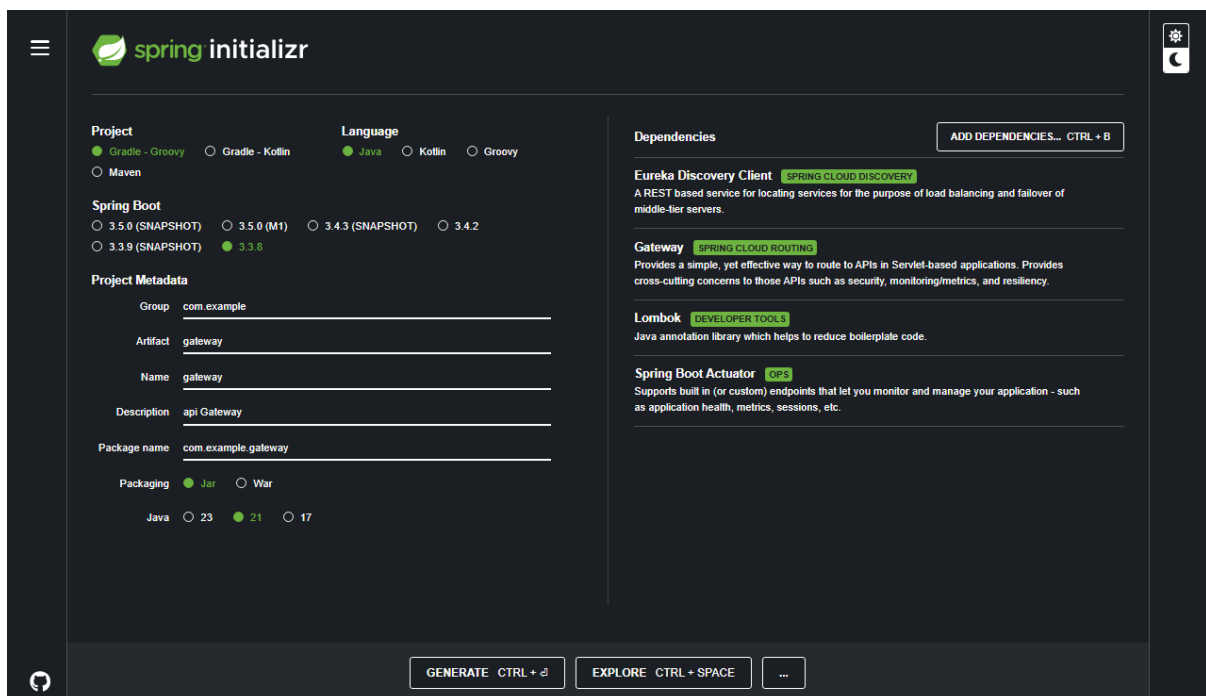
2.1. Configuration du Projet

2.1.1. Initialisation du projet avec Spring Initializr

Pour démarrer le projet, utilisez **Spring Initializer** (<https://start.spring.io/>). Choisissez les options suivantes :

- **Projet** : Maven ou Gradle selon votre préférence.
- **Langage** : Java
- **Spring Boot** : Sélectionnez la version stable la plus récente.
- **Dependencies** : Ajoutez les dépendances énumérées ci-dessus

Cliquez sur **Generate** pour télécharger le projet.



The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '3.3.8 (SNAPSHOT)' is selected. The 'Project Metadata' section includes fields for Group (com.example), Artifact (gateway), Name (gateway), Description (api Gateway), and Package name (com.example.gateway). Under 'Packaging', 'Jar' is selected, and under 'Java', '21' is selected. On the right, the 'Dependencies' section lists several dependencies: 'Eureka Discovery Client' (Spring Cloud Discovery), 'Gateway' (Spring Cloud Routing), 'Lombok' (Developer Tools), and 'Spring Boot Actuator' (Ops). At the bottom, there are buttons for 'GENERATE' (CTRL + G), 'EXPLORE' (CTRL + SPACE), and a menu icon.

Figure 6 . Initialisation avec spring initializr

2.1.2. Importation du dossier téléchargé dans l'IDE et mise à jour des dépendances

Après avoir généré le projet, décompressez-le et importez-le dans votre IDE, ici VScode. Assurez-vous que le projet est correctement configuré et que toutes les dépendances sont résolues. S'assurer que la mise à jour des dépendances Maven soit fonctionnelle.

2.1.3. Configuration des différents modules

Nous présentons ci-dessous le fonctionnement des dépendances des différents modules :

1.Service Autorisation (authorization-service)

La présentation du service autorisation (authorization-service):

```
# Application configuration
server.port=8085
spring.application.name=authorization-service

# Eureka configuration
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.client.fetch-registry=true
eureka.client.register-with-eureka=true
eureka.instance.prefer-ip-address=true
eureka.instance.hostname=localhost

# Logging
logging.level.com.netflix.discovery=DEBUG
logging.level.com.netflix.eureka=DEBUG

# Swagger UI Configuration
springdoc.swagger-ui.path=/swagger-ui.html
springdoc.api-docs.path=/v3/api-docs
springdoc.swagger-ui.operations-sorter=method
springdoc.swagger-ui.try-it-out-enabled=true
```

Figure 7. Service d'authentification

Explication :

- **server.port=8085** : Définit le port sur lequel le service d'autorisation sera accessible.
- **spring.application.name=authorization-service** : Définit le nom de l'application pour la découverte via Eureka.
- **eureka.client.service-url.defaultZone=http://localhost:8761/eureka/** : Spécifie l'URL du serveur Eureka auquel le service doit s'enregistrer.
- **eureka.client.fetch-registry=true** : Permet au service de récupérer la liste des autres services enregistrés dans Eureka.
- **eureka.client.register-with-eureka=true** : Indique si l'application doit s'enregistrer elle-même auprès d'Eureka.
- **eureka.instance.prefer-ip-address=true** : Force l'utilisation de l'adresse IP plutôt que du nom d'hôte lors de l'enregistrement et la découverte du service.



- **eureka.instance.hostname=localhost** : Définit localhost comme nom d'hôte sous lequel le service sera enregistré dans **Eureka**.
- **logging.level.com.netflix.discovery=DEBUG** et **logging.level.com.netflix.eureka=DEBUG** : le mode **DEBUG** affiche des messages détaillés pour aider au débogage des interactions avec Eureka.

Les autres configurations assurant le bon fonctionnement du service autorisation se présentent comme-suit :

La configuration de swagger: Ce fichier configure Swagger, un outil de documentation d'API, pour votre service d'authentification.

Explication:

- **Annotations:**
 - **@Configuration** : Indique que cette classe contient des méthodes de configuration Spring.
 - **@Bean** : Marque une méthode comme une source de bean Spring.
- **Méthode openAPI :**
 - Configure les informations de l'API (titre, description, version, licence) à partir des propriétés de l'application.
 - Ajoute une exigence de sécurité pour l'authentification par jeton JWT.
- **Méthode create API Key Scheme :**
 - Créer un schéma de sécurité pour l'authentification par jeton (Bearer), spécifiant qu'il utilise le format JWT.

La gestion des erreurs : Il s'agit ici de configurer un client HTTP, pour gérer les erreurs.

Explication

- **Méthode erreur Decoder :**
 - Retourne un nouvel objet CustomError Decoder, qui gère les erreurs lors des appels HTTP effectués par Feign. Cela permet de centraliser la logique de gestion des erreurs.

La configuration de la sécurité de l'application : Il s'agit ici d'exposer l'authentification et les règles CORS.

Explication



- **Méthode security FilterChain :**
 - Configure les règles de sécurité, désactive CSRF, permet l'accès public à certaines routes (authentification et documentation Swagger) et exige une authentification pour d'autres requêtes.
- **Méthode authenticationManager :**
 - Fournit un gestionnaire d'authentification pour gérer les processus d'authentification des utilisateurs.
- **Méthode passwordEncoder :**
 - Retourne un encodeur de mot de passe utilisant BCrypt, pour sécuriser les mots de passe des utilisateurs.
- **Méthode corsConfigurer :**
 - Configure CORS (Cross-Origin Resource Sharing) pour autoriser toutes les méthodes sur toutes les routes, permettant ainsi aux applications front-end d'accéder à votre API.

2. Serveur de Configuration (Config Server)

La présentation du serveur de configuration (Config Server):

La configuration liée au serveur :

```
server:
  port: 8888

spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/MarcSab20/api_gateway
          default-label: main
          search-paths: config

management:
  endpoints:
    web:
      exposure:
        include: "*"
  endpoint:
    health:
      show-details: always
```

Figure 8. Configuration du serveur

Explication :

- `server.port=8888` : Définit le port du serveur de configuration.
- `spring.cloud.config.server.git.uri` : Indique l'URL du dépôt Git où les configurations sont stockées.
- `management.endpoints.web.exposure.include="*"` : Expose tous les endpoints de gestion, ce qui est utile pour la surveillance

3. Service Eureka

Créez un fichier `application.yml` dans le répertoire `src/main/resources` :

```
1  server.port=8761
2  eureka.client.register-with-eureka=false
3  eureka.client.fetch-registry=false
4
```

Figure 9. Serveur Eureka

Explication :

- `server.port=8761` : Définit le port sur lequel le serveur Eureka est accessible.

- `eureka.client.register-with-eureka=false` et `fetch-registry=false` : Ces paramètres désactivent l'enregistrement et la récupération des services pour ce serveur.

4. API Gateway

Créez également un fichier `application.yml` dans le répertoire `src/main/resources` :

```
1  server:
2    port: 8080
3
4  spring:
5    application:
6      name: api-gateway
7    cloud:
8      gateway:
9        discovery:
10         locator:
11           enabled: true
12       httpclient:
13         connect-timeout: 60000
14         response-timeout: 60s
```

Figure 10. API Gateway

Explication :

- `server.port=8080` : Définit le port de l'API Gateway.
- `spring.cloud.gateway.discovery.locator.enabled=true` : Active la découverte des services via l'API Gateway.
- `connect-timeout : 60000` (60 secondes) définit le délai maximum pour établir une connexion avec un service en aval.
- `response-timeout : 60s` définit le délai maximum d'attente pour une réponse après l'établissement de la connexion. Cependant, l'unité 60s pourrait ne pas être interprétée correctement, il faudrait utiliser 60000ms pour assurer la cohérence.

5. Docker

Docker Compose Configuration

Créez un fichier `docker-compose.yml` à la racine de votre projet :

```

1  version: "3.5"
2
3  > Run All Services
4  services:
5    > Run Service
6    postgres:
7      container_name: postgres_marc
8      image: postgres:latest
9      environment:
10       POSTGRES_USER: postgres
11       POSTGRES_PASSWORD: 55
12       POSTGRES_DB: microservice
13      volumes:
14       - postgres_data:/var/lib/postgresql/data
15      ports:
16       - "5432:5432"
17
18    > Run Service
19    zookeeper:
20      container_name: zookeeper_marc
21      image: "docker.io/bitnami/zookeeper:3"
22      ports:
23       - "2181:2181"
24      volumes:
25       - "zookeeper_data:/bitnami"
26      environment:
27       - ALLOW_ANONYMOUS_LOGIN=yes
28
29    > Run Service
30    kafka:
31      container_name: kafka_marc
32      image: "docker.io/bitnami/kafka:2-debian-10"
33      ports:
34       - "9092:9092"
35      expose:
36       - "9093"
37      volumes:
38       - "kafka_data:/bitnami"
39      environment:
40       - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
41       - ALLOW_PLAINTEXT_LISTENER=yes
42       - KAFKA_ADVERTISED_LISTENERS=INSIDE://kafka:9093,OUTSIDE://localhost:9092
43       - KAFKA_LISTENER_SECURITY_PROTOCOL_MAP=INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
44       - KAFKA_LISTENERS=INSIDE://0.0.0.0:9093,OUTSIDE://0.0.0.0:9092
45       - KAFKA_INTER_BROKER_LISTENER_NAME=INSIDE
46       - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
47      depends_on:
48       - zookeeper
49
50    > Run Service
51    kafka-ui:
52      container_name: kafka-ui_marc
53      image: provectuslabs/kafka-ui
54      ports:
55       - "9090:8080"
56      restart: always
57      environment:
58       - KAFKA_CLUSTERS_0_NAME=local
59       - KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS=kafka:9093
60       - KAFKA_CLUSTERS_0_ZOOKEEPER=localhost:2181
61
62    volumes:
63      zookeeper_data:
64        driver: local
65      kafka_data:
66        driver: local
67      postgres_data:
68        driver: local

```

Figure 11. Configuration docker

Explication :

Ce fichier **docker-compose.yml** configure une infrastructure composée de **PostgreSQL**, **Zookeeper**, **Kafka** et **Kafka UI**. PostgreSQL est utilisé comme base de données, avec des identifiants définis et un volume pour la persistance des données. Zookeeper agit comme un service de coordination pour Kafka, permettant une gestion stable des brokers. Kafka est configuré pour utiliser Zookeeper, exposer ses ports pour

la communication interne et externe, et stocker ses données dans un volume. Enfin, Kafka UI fournit une interface web accessible sur le port **8080** pour superviser et gérer les topics Kafka.

Les services sont interconnectés grâce à `depends_on`, assurant que Kafka démarre après Zookeeper. Tous les volumes (`postgres_data`, `zookeeper_data`, `kafka_data`) garantissent la persistance des données même après l'arrêt des conteneurs. L'authentification sur Zookeeper et Kafka est désactivée (`ALLOW_ANONYMOUS_LOGIN=yes` et `ALLOW_PLAINTEXT_LISTENER=yes`), ce qui simplifie le développement mais n'est pas sécurisé pour un environnement de production.

2.2. Architecture interne du système

Présentation de l'architecture du système implémentée :

Docker Environnement

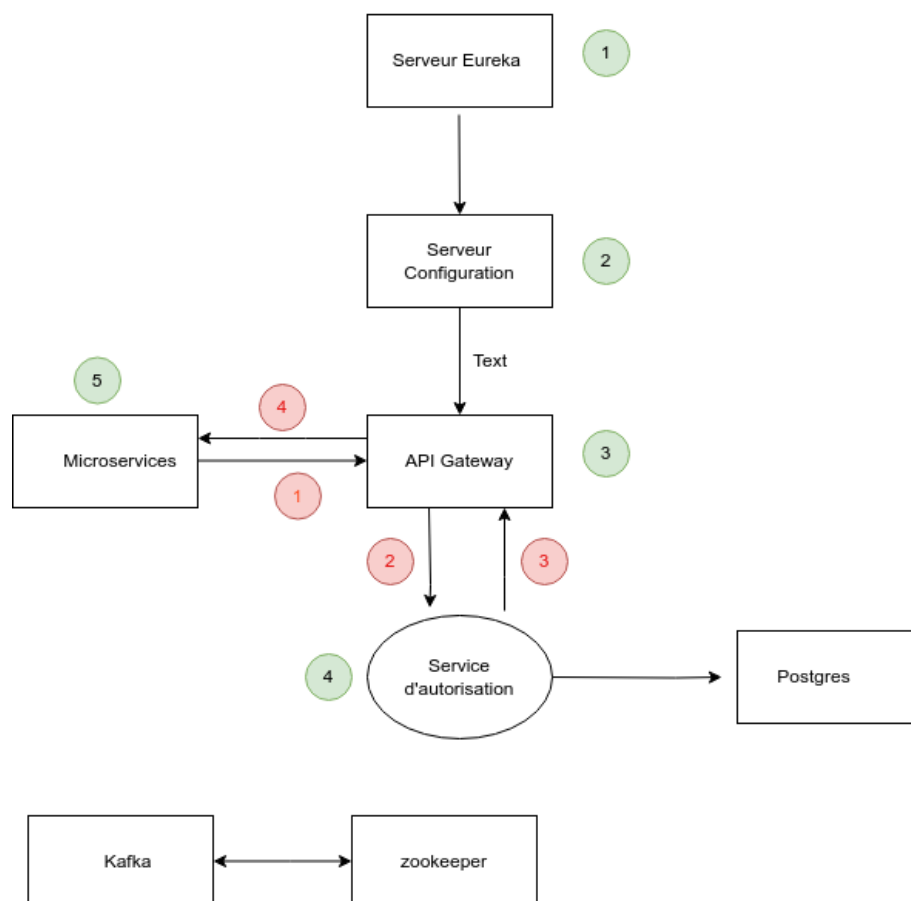


Figure 12. Architecture du Système

Description de l'architecture :



Client à API Gateway : Le client envoie des requêtes HTTP à l'API Gateway, qui sert de point d'entrée unique pour toutes les communications.

API Gateway à Eureka Server : L'API Gateway s'enregistre auprès du serveur Eureka pour découvrir les services disponibles et rediriger les requêtes vers eux.

API Gateway à Configuration Server : Tous les services, y compris l'API Gateway, interrogent le serveur de configuration pour récupérer les paramètres nécessaires à leur fonctionnement.

API Gateway à Auth Service : Les requêtes d'authentification passent par l'API Gateway et sont redirigées vers le service d'authentification.

Ainsi pour déployer ce projet il faut successivement exécuter :

A- Démarrage du Serveur Eureka : Ce serveur agit comme un registre des microservices. Tous les services enregistrés y seront accessibles pour la découverte dynamique. Il doit être lancé en premier.

B- Démarrage du Serveur de Configuration : Il fournit les configurations centralisées aux microservices et autres composants. Il récupère généralement ses configurations depuis un dépôt Git ou une autre source.

C- Démarrage de l'API Gateway : L'API Gateway est un point d'entrée unique pour les clients et agit comme un proxy entre les microservices et les utilisateurs. Elle gère la sécurité, le routage et la limitation des requêtes.

D- Démarrage du Service d'Autorisation : Ce service gère l'authentification et l'autorisation des utilisateurs. Il est essentiel pour sécuriser les interactions avec les microservices.

E- Démarrage des Microservices : Une fois tous les services de base en place (Eureka, Config, API Gateway, Service d'Autorisation), les microservices peuvent être démarrés et enregistrés auprès d'Eureka.

Le flux de communication est multiforme. Il y a plusieurs types de communication.

A- Communication entre l'API Gateway et les Microservices :

- L'API Gateway envoie des requêtes aux microservices enregistrés.
- Si un microservice est indisponible ou mal enregistré, l'API Gateway peut échouer à acheminer les requêtes.

B- Communication entre l'API Gateway et le Service d'Autorisation :

- Toute requête utilisateur passe par l'API Gateway et est validée par le Service d'Autorisation.

- Ce service vérifie les JWT ou autres jetons d'authentification pour valider l'accès aux ressources demandées.

C- Accès du Service d'Autorisation à la base de données PostgreSQL :

- Le service d'autorisation stocke et vérifie les informations des utilisateurs (comptes, rôles, permissions) dans PostgreSQL.

D- Communication entre les Microservices et le Service d'Autorisation :

- Les microservices peuvent interroger le Service d'Autorisation pour vérifier les permissions des utilisateurs avant de répondre aux requêtes.

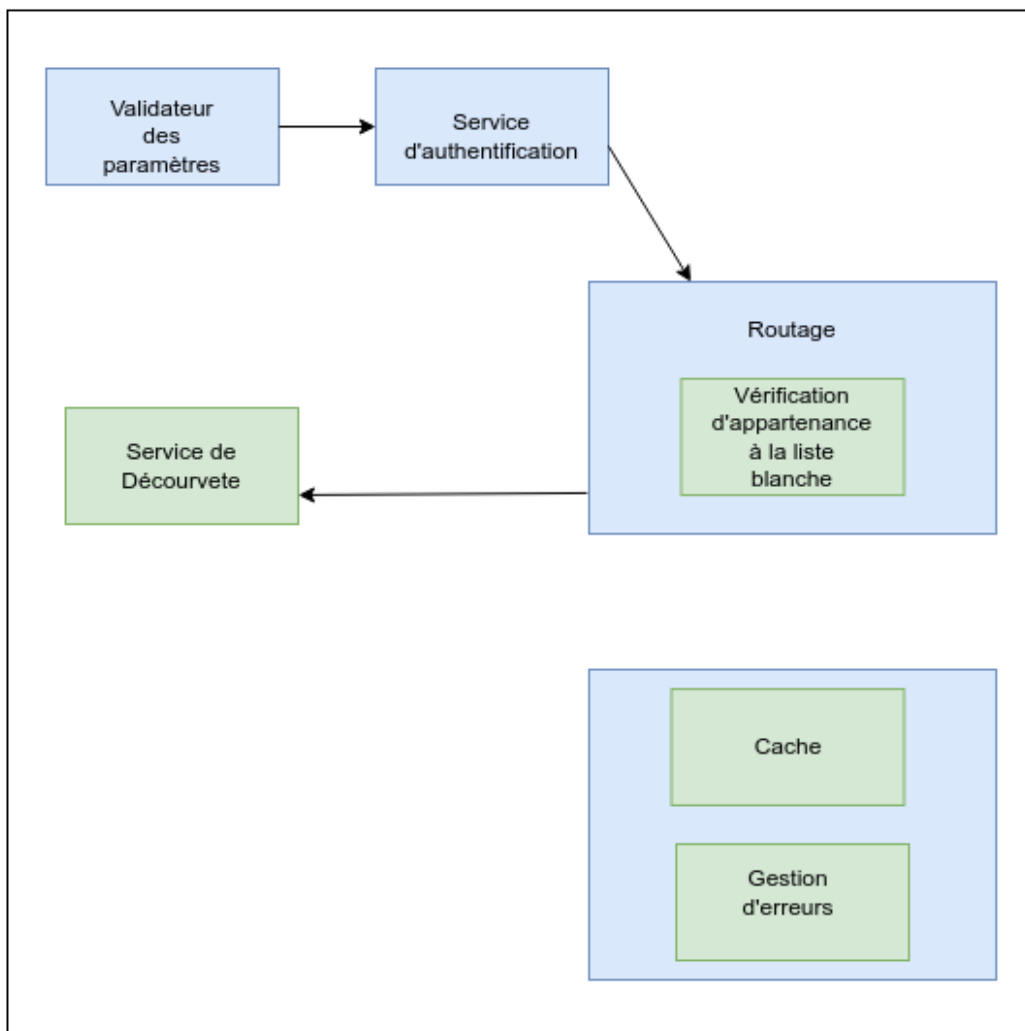


Figure 13. APIFlow

ServicePostgres : Le service d'authentification interroge la base de données Postgres pour stocker et récupérer les informations des utilisateurs.

Kafka à Zookeeper : Kafka utilise Zookeeper pour la gestion de son cluster, garantissant la coordination et l'état des brokers Kafka.

API Gateway à Other Services : L'API Gateway redirige également les requêtes vers d'autres services microservices, en fonction de la logique métier et des routes définies.

Nous avons utilisé une architecture conteneurisée, en effet chacun des blocs de l'architecture communique entre eux, et se présente comme suit :

Présentation des Conteneurs Docker

- **Client** : Peut-être une application frontale déployée dans un conteneur.
- **API Gateway** : Conteneur configuré pour gérer les requêtes et interagir avec les autres services.
- **Eureka Server** : Conteneur pour le service de découverte.
- **Auth Service** : Conteneur pour gérer l'authentification.
- **Postgres** : Conteneur pour la base de données.
- **Configuration Server** : Conteneur fournissant des configurations centralisées.
- **Kafka** : Conteneur pour le service de messagerie.
- **Zookeeper** : Conteneur pour la gestion de Kafka.

2.3. Développement des modules de l'API Gateway

L'API Gateway repose sur plusieurs modules qui assurent la gestion et l'optimisation des requêtes entre les clients et les microservices. Ces modules garantissent la sécurité, la performance et la résilience du système.

2.3.1. Validation des Paramètres

L'un des premiers rôles de l'API Gateway est de s'assurer que les requêtes des clients sont validées avant d'être transmises aux microservices. Ce module vérifie la présence des paramètres requis, s'assure que le format des données est correct et contrôle si les valeurs respectent les plages autorisées (un âge ne peut pas être négatif, un prix doit être supérieur à zéro etc...). Cette validation permet d'éviter d'envoyer des requêtes erronées aux microservices, ce qui réduit les traitements inutiles et prévient les erreurs en aval.

2.3.2 Vérification d'appartenance à la Liste Blanche

Ce module permet de filtrer certaines requêtes en fonction de leur origine ou de leur contenu. Il vérifie notamment si l'adresse IP du client figure dans une liste autorisée, contrôle la validité et l'authenticité d'une clé API ou d'un token d'authentification et restreint l'accès à

certaines fonctionnalités pour des utilisateurs spécifiques. Grâce à ces vérifications, seules les requêtes autorisées peuvent atteindre les microservices, renforçant ainsi la sécurité du système.

2.3.3 Transformation de la Requête

Les microservices ayant des exigences différentes en matière de format et de structure de requêtes, ce module adapte les requêtes en fonction des besoins de chaque service. Il peut convertir les formats de données, par exemple en passant de JSON à XML, ajouter ou supprimer des en-têtes HTTP et modifier le corps de la requête afin d'assurer sa compatibilité avec les services en aval. Cette flexibilité permet de garantir une communication fluide et efficace entre les clients et les microservices.

2.3.4 Service d'Authentification

L'authentification est un élément clé de la sécurité dans une architecture microservices. L'API Gateway intègre un service d'authentification qui vérifie l'identité des utilisateurs à l'aide du mécanisme JWT. Il gère également les permissions et les rôles des utilisateurs, garantissant que seuls ceux qui disposent des autorisations nécessaires peuvent accéder aux services. En centralisant cette gestion, ce module simplifie la sécurité et évite aux microservices d'implémenter individuellement des mécanismes d'authentification.

2.3.5 Service de Découverte (Eureka)

Dans un environnement où les microservices peuvent être fréquemment redéployés et changer d'adresse, il est essentiel de disposer d'un mécanisme de découverte dynamique. Grâce à Eureka, l'API Gateway peut enregistrer et récupérer les adresses des microservices disponibles, assurant ainsi une répartition intelligente du trafic. Ce module permet de maintenir une architecture flexible et évite d'avoir à configurer manuellement les adresses des services, améliorant ainsi la scalabilité et la résilience du système.

2.3.6 Routing

Le module de routing est chargé de rediriger les requêtes des clients vers les microservices appropriés. Il analyse l'URL de la requête et identifie le microservice cible, applique des règles de réécriture d'URL si nécessaire et dirige les requêtes vers plusieurs instances pour équilibrer la charge. En garantissant une distribution efficace du trafic, ce module optimise les performances et améliore la disponibilité des services.

2.3.7 Cache

Afin d'améliorer la rapidité des réponses et de réduire la charge sur les microservices, un système de cache est mis en place. Ce module stocke temporairement les réponses des requêtes fréquentes, ce qui évite d'interroger systématiquement les microservices pour les mêmes données. En minimisant la latence et en réduisant la consommation de ressources, le

cache améliore significativement l'efficacité du système, tout en assurant que les données restent à jour grâce à des mécanismes de rafraîchissement.

2.3.8 Gestion des Erreurs

Les erreurs peuvent survenir à plusieurs niveaux, qu'il s'agisse d'une requête invalide, d'un microservice indisponible ou d'un dépassement de délai. L'API Gateway intègre un module de gestion des erreurs qui fournit des messages explicites aux clients, applique des stratégies de retry pour certaines erreurs temporaires et renvoie des statuts HTTP appropriés (400, 401, 500, etc.).

2.3.9 Circuit Breaker

Lorsqu'un micro service devient indisponible, il est essentiel d'éviter d'envoyer des requêtes répétées qui pourraient aggraver la situation. Ce module implémente le Circuit Breaker, un mécanisme qui détecte automatiquement les pannes et bloque temporairement les requêtes vers un service défaillant. Il permet également de renvoyer une réponse de secours ou un message indiquant l'indisponibilité du service.

2.3.10 Enregistrement des Logs

Pour assurer un suivi efficace du fonctionnement de l'API Gateway, un module d'enregistrement des logs a été mis en place. Il collecte des informations sur les requêtes reçues, les réponses envoyées ainsi que les erreurs et événements système. Ces logs permettent de surveiller l'activité du système, d'identifier rapidement les anomalies et d'optimiser les performances.

3. Tests, validation et résultats

Pour tester ce projet, nous avons utilisé le microservice utilisateur et le microservice des notifications. Nous avons testé ce code à l'aide de Postman :



← → ↻ localhost:8085/swagger-ui/index.html

Swagger
Supported by SMARTBEAR

/v3/api-docs Explore

OpenAPI definition v0 OAS 3.0

/v3/api-docs

Servers

http://localhost:8085 - Generated server url

Service d'Autorisation API pour gérer les autorisations des services

- GET** /api/authorization/check/{serviceName} Vérifier l'autorisation d'un service
- POST** /api/authorization/toggle/{serviceName} Basculer l'état d'autorisation d'un service
- POST** /api/authorization/enable/{serviceName} Activer un service
- POST** /api/authorization/disable/{serviceName} Désactiver un service

Figure 14. Swagger du service de l'autorisation



← → ↻ localhost:8085/swagger-ui/index.html#/Service d'Autorisation

Service d'Autorisation API pour gérer les autorisations des services

GET /api/authorization/check/{serviceName} Vérifier l'autorisation d'un service

Vérifie si un service spécifique est autorisé à communiquer

Parameters Cancel

Name	Description
serviceName * required string (path)	nom du service à vérifier <input type="text" value="serviceName"/>

Execute

Responses

Code	Description	Links
200	Retourne true si le service est autorisé, false sinon Media type <input type="text" value="*/"/> Controls Accept header. Example Value Schema <pre>true</pre>	No links

Figure 15: la fonction check du service d'autorisation



← → ↻ localhost:8085/swagger-ui/index.html#/Service d'Autorisation 🔍 ☆

Service d'Autorisation API pour gérer les autorisations des services

GET /api/authorization/check/{serviceName} Vérifier l'autorisation d'un service

POST /api/authorization/toggle/{serviceName} Basculer l'état d'autorisation d'un service

Inverse l'état d'autorisation actuel du service

Parameters Cancel

Name	Description
serviceName * required string (path)	nom du service à basculer

Execute

Responses

Code	Description	Links
200	État du service basculé avec succès	No links

Media type

Controls Accept header.

Example Value | Schema

```
string
```

Figure 16: la fonction "toggle" du service d'autorisation

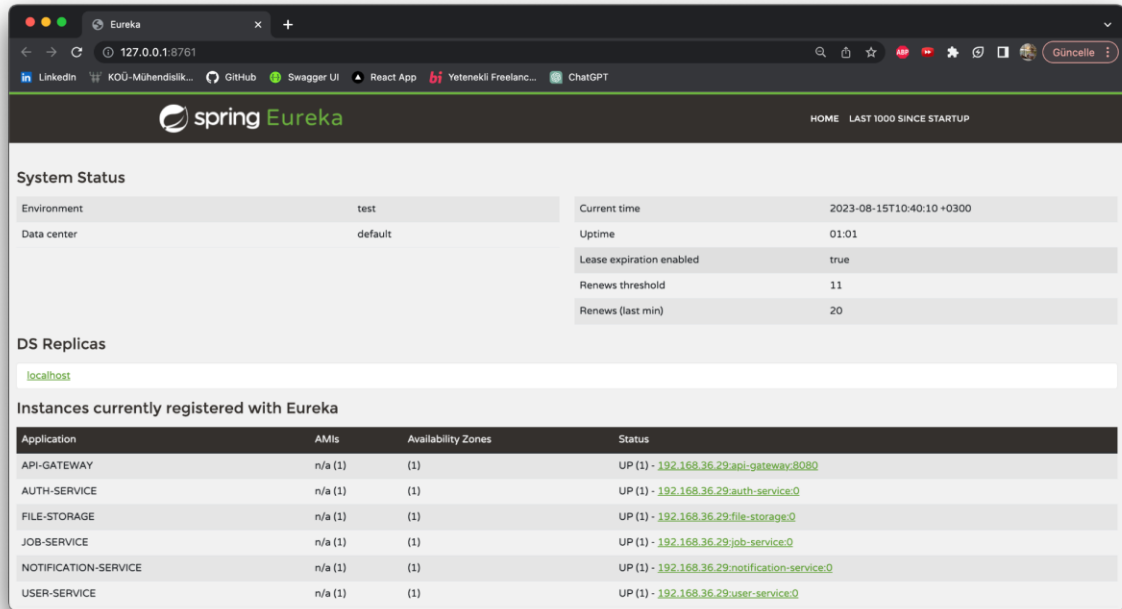


Figure 17. Interface d'Eureka pour la découverte des applications

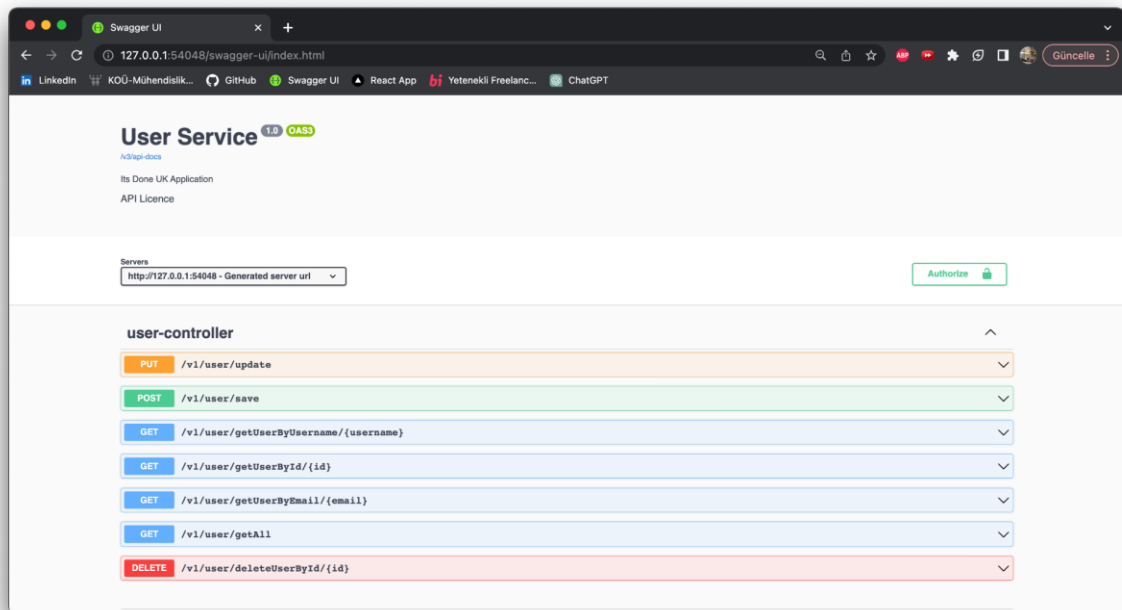


Figure 18. Swagger du service gestion des utilisateurs

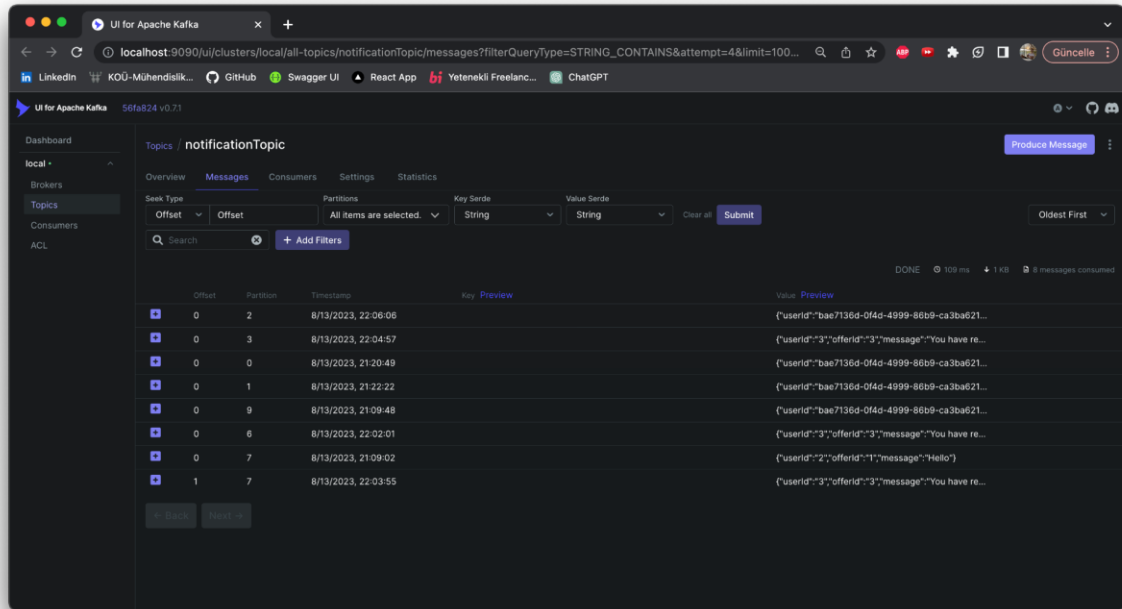


Figure 19. Gestion des topic Kafka



Chapitre 4 : GUIDE DE DEPLOIEMENT

Chapitre 4 : Guide de déploiement

1. Guide de déploiement pour l'API Gateway

Etape 1 : Préparer l'environnement

Outils nécessaires :

- IDE installé (VSCode ou IntelliJ Idea)
- Java 21 (21.0.5) : assurez-vous d'avoir cette version de java 21 installé
- Maven : Pour construire le projet
- Docker : pour conteneuriser l'API Gateway
- Docker compose : Pour orchestrer les conteneurs

Etape 2 : Construire le projet

Construire le projet :

Il faut cloner/télécharger le dossier du projet à partir du lien github : https://github.com/MarcSab20/api_gateway et se placer sur la branche Marco.

Si vous êtes sur VSCode, il faut installer les dépendances springboot Tools, CMake et docker.

Etape 3 : Conteneurisé l'API Gateway avec Docker

Modifier le Dockerfile :

Aller dans le répertoire qui contient les fichiers docker-compose.yml et taper la commande :

```
$ docker compose up
```

Etape 4 : Lancer le serveur Eureka

Pour se faire, il faut :

- Ouvrir un autre terminal dans le projet, aller dans le répertoire où se trouve Eureka :

```
$ cd eureka-server
```

- Lancer le serveur Eureka avec la commande



```
$ mvn spring-boot : run
```

Ou aller dans le fichier ApplicationEurekaServer.java et cliquer sur la petite icone « play ».

Etape 5 : Lancer le serveur de configuration

Pour se faire, il faut :

- Ouvrir un autre terminal dans le projet, aller dans le répertoire où se trouve le serveur de configuration :

```
$ cd config-server
```

- Lancer le serveur de configuration avec la commande

```
$ mvn spring-boot : run
```

Ou aller dans le fichier ConfigServerApplication.java et cliquer sur la petite icone « play ».

Etape 6 : Lancer l'API Gateway

Pour se faire, il faut :

- Ouvrir un autre terminal dans le projet, aller dans le répertoire où se trouve l'API Gateway:

```
$ cd gateway
```

- Lancer l'api Gateway avec la commande

```
$ mvn spring-boot : run
```

Ou aller dans le fichier GatewayApplication.java et cliquer sur la petite icone « play ».

Etape 7 : Lancer le service d'autorisation

Pour se faire, il faut :

- Ouvrir un autre terminal dans le projet, aller dans le répertoire où se trouve le service d'autorisation :

```
$ cd auth-service
```

- Lancer le serveur d'autorisation avec la commande

```
$ mvn spring-boot: run
```

Ou aller dans le fichier AuthServiceApplication.java et cliquer sur la petite icone « play ».



Etape 8 : Vérifier le déploiement

- Eureka Server : Accédez à <http://localhost:8761> pour voir les services enregistrés.
- API Gateway : Testez les endpoints via <http://localhost:8080>
- ConfigServer : Accédez à <http://localhost:8888>
- Le service d'autorisation : Accéder à <http://localhost:8081/swagger-ui/index.html>
- Interface de KAFKA : <http://localhost:9090>
- Microservices : Vérifiez les logs des conteneurs pour vous assurer que tout fonctionne.

2. Limites et perspectives

2.1. Limites

1. Complexité de configuration

- Problème : L'API Gateway nécessite une configuration détaillée pour le routage, la sécurité, la gestion des erreurs, etc. Cela peut devenir complexe à mesure que le nombre de microservices augmente.
- Impact : Une mauvaise configuration peut entraîner des erreurs de routage, des problèmes de sécurité ou des temps de réponse lents.

2. Dépendance à Eureka

- Problème : L'API Gateway dépend d'un service de découverte (Eureka) pour localiser les microservices. Si Eureka tombe en panne, l'API Gateway ne peut plus router les requêtes.
- Impact : Une panne d'Eureka peut rendre l'ensemble du système indisponible.

3. Sécurité

- Problème : L'API Gateway est un point d'entrée unique pour tous les clients. Si elle est compromise, l'ensemble du système est vulnérable.
- Impact : Une faille de sécurité peut entraîner des attaques exemples : DDoS, injection SQL.

2.2 Perspectives

1. Passage à une architecture multi-gateway

- Solution : Au lieu d'une seule API Gateway, utilisez plusieurs gateways pour répartir la charge (ex : une gateway par région ou par groupe de microservices).
- Avantages :
 - Réduit le risque de goulot d'étranglement.
 - Améliore la résilience et la disponibilité.

2. Monitoring et observabilité

- Solution :
 - Intégrez des outils de monitoring comme Prometheus et Grafana pour surveiller les performances de l'API Gateway.
 - Utilisez des outils de tracing comme Jaeger ou Zipkin pour suivre les requêtes entre les microservices.
- Avantages :
 - Permet de détecter rapidement les problèmes de performance ou de disponibilité.
 - Facilite le débogage.



5. Automatisation de la configuration

- **Solution :**
 - Utilisez des outils comme Spring Cloud Config ou Consul pour gérer la configuration de l'API Gateway de manière centralisée.
 - Automatisez le déploiement avec des pipelines CI/CD (ex : Jenkins, GitLab CI).
- **Avantages :**
 - Réduit les erreurs de configuration.
 - Accélère les déploiements.

Conclusion

Il était question pour nous de mettre sur pied une API gateway dans une architecture microservice. IL en ressort qu'une API est essentielle dans une architecture de microservices pour garantir une communication fluide, sécurisée et optimisée entre les clients et les services. Grâce à Spring Cloud Gateway, nous avons pu mettre en place une solution intégrant l'authentification et l'autorisation via JWT, un routage dynamique et des mécanismes de sécurité tels que le Rate limiting. L'intégration avec un service de couverte comme Eureka permet à l'API Gateway d'assurer une redirection efficace des requêtes vers les différents services tout s'adaptant aux changements dynamiques d'états des microservices.

Références Bibliographique

- [1] A. W. "Understanding API Gateways: The Key to Microservices"
[En ligne] <https://www.example.com/understanding-api-gateways>, consulté le 25 Janvier 2025.
- [2] Thorne, Joseph, "The Rise of API Gateways", [En ligne] <https://www.example.com/rise-of-api-gateways>, consulté le 25 Janvier 2025.
- [3] Google Cloud, "Qu'est-ce que l'architecture de microservices ?", [En ligne] <https://cloud.google.com/learn/what-is-microservices-architecture?hl=fr>, consulté le 25 Janvier 2025.
- [4] Richardson, Chris, "Microservices : A Definition of This New Architectural Term", *IEEE Software*, vol. 32, pp. 112-116, 2015.
- [5] Kong, "Uber Streaming APIs : Powering Real-Time Apps at Global Scale", [En ligne] <https://konghq.com/resources/videos/uber-streaming-apis-powering-real-time-apps-global-scale>, consulté le 25 janvier 2025.
- [6] Featured Customers, "Find B2B & SaaS Software & Services - Reviews, Testimonials & Case Studies", [En ligne] <https://www.featuredcustomers.com/vendor/apigee>, consulté le 25 janvier 2025.
- [7] Bacancy Technology, "Case Study on Netflix AWS Migration | Netflix Migration", [En ligne] <https://www.bacancytechnology.com/blog/netflix-aws-migration>, consulté le 25 janvier 2025.
- [8] Jean-Jérôme Lévy, "Le Guide Ultime pour Maîtriser l'Architecture Hexagonale : Focus sur le Domaine", [En ligne] <https://scalastic.io/hexagonal-architecture-domain/>, consulté le 25 janvier 2025.
- [9] Scalastic, "L'Architecture Hexagonale", [En ligne] <https://scalastic.io/hexagonal-architecture/>, consulté le 25 janvier 2025.