Rapport de Migration de Beep vers une Architecture Microservices

Table des Matières

Introduction

Dans le contexte actuel de transformation numérique, les entreprises sont confrontées à des défis croissants en matière de scalabilité, de flexibilité et de rapidité de mise sur le marché. Traditionnellement, les applications logicielles étaient développées sous forme de monolithes, où toutes les fonctionnalités sont intégrées dans une seule base de code. Bien que cette approche ait ses avantages, notamment en termes de simplicité initiale de développement et de déploiement, elle présente des limitations significatives à mesure que l'application évolue.

Le passage d'une architecture monolithique à une architecture microservices représente une réponse stratégique à ces limitations. Les microservices permettent de décomposer une application en plusieurs services indépendants, chacun responsable d'une fonctionnalité spécifique. Cette approche offre plusieurs avantages clés :

Scalabilité : Chaque microservice peut être mis à l'échelle indépendamment en fonction des besoins spécifiques, optimisant ainsi l'utilisation des ressources.

Flexibilité : Les équipes de développement peuvent travailler sur différents services en parallèle, utilisant les technologies les mieux adaptées à chaque cas d'utilisation, ce qui accélère le développement et l'innovation.

Résilience : Une défaillance dans un microservice n'affecte pas nécessairement l'ensemble du système, améliorant ainsi la robustesse globale de l'application.

Déploiement Continu : Les microservices facilitent l'adoption de pratiques DevOps, permettant des déploiements plus fréquents et plus sûrs, et réduisant le temps de mise sur le marché des nouvelles fonctionnalités.

Ce rapport vise à explorer les motivations, les défis et les bénéfices associés à la transition d'une architecture monolithique vers une architecture microservices. Nous examinerons les technologies et les outils utilisés pour faciliter cette transition, ainsi que les meilleures pratiques pour assurer une migration réussie. Enfin, nous évaluerons l'impact de cette transformation sur les performances, la maintenance et l'évolutivité de l'application.

Dans ce document nous décrirons donc la migration de notre application, Beep, d'une architecture monolithique vers une architecture microservices.

1. Découpage Fonctionnel et Microservices

1.1 Redéfinition des Fonctionnalités

- En tant qu'**utilisateur invité**, je veux **m'inscrire et me connecter** afin de **participer aux discussions**.
- En tant qu'**utilisateur enregistré**, je veux **créer un serveur** afin de **gérer ma propre communauté**.
- En tant qu'**utilisateur enregistré**, je veux **personnaliser mon profil** afin de **refléter mon identité et mes intérêts**.

- En tant qu'**utilisateur enregistré**, je veux **voir l'état de présence des autres membres d'un serveur ou de mes amis** afin de **savoir qui est en ligne et disponible pour discuter**.
- En tant qu'**utilisateur enregistré**, je veux **rejoindre un serveur** afin de **interagir avec d'autres utilisateurs**.
- En tant qu'**administrateur de serveur**, je veux **créer des channels de discussion** afin de **organiser les échanges**.
- En tant qu'**administrateur de serveur**, je veux **gérer les rôles et permissions des utilisateurs** afin de **contrôler les accès aux fonctionnalités**.
- En tant qu'**utilisateur**, je veux **envoyer des messages texte et vocaux** afin de **communiquer avec les autres membres**.
- En tant qu'**utilisateur**, je veux **partager des fichiers et des images** afin de **échanger du contenu avec ma communauté**.
- En tant qu'**utilisateur**, je veux **recevoir des notifications** afin de **être informé des nouveaux messages et événements**.
- En tant qu'**administrateur de serveur**, je veux **modérer le contenu** afin de **garantir le respect des règles de la communauté**.
- En tant qu'**utilisateur**, je veux **effectuer des recherches dans les messages** afin de **retrouver rapidement une information**.
- En tant qu'**utilisateur**, je veux **participer à des appels vidéo ou audio** afin de **communiquer en temps réel avec d'autres membres**.

1.2 Quartiers Fonctionnels

- **Authentification & Gestion des Utilisateurs**
- Inscription, Connexion, Gestion des Profils
- Gestion des rôles et permissions
- **Gestion des Serveurs et channels**
- Création et gestion des serveurs
- Création et gestion des channels
- Modération des discussions
- **Messagerie et Communication**
- Envoi de messages texte et vocaux
- Partage de fichiers et médias
- Notifications en temps réel
- **Recherche & Indexation**
- Moteur de recherche pour messages et fichiers

1.3 Schéma d’Architecture Microservices

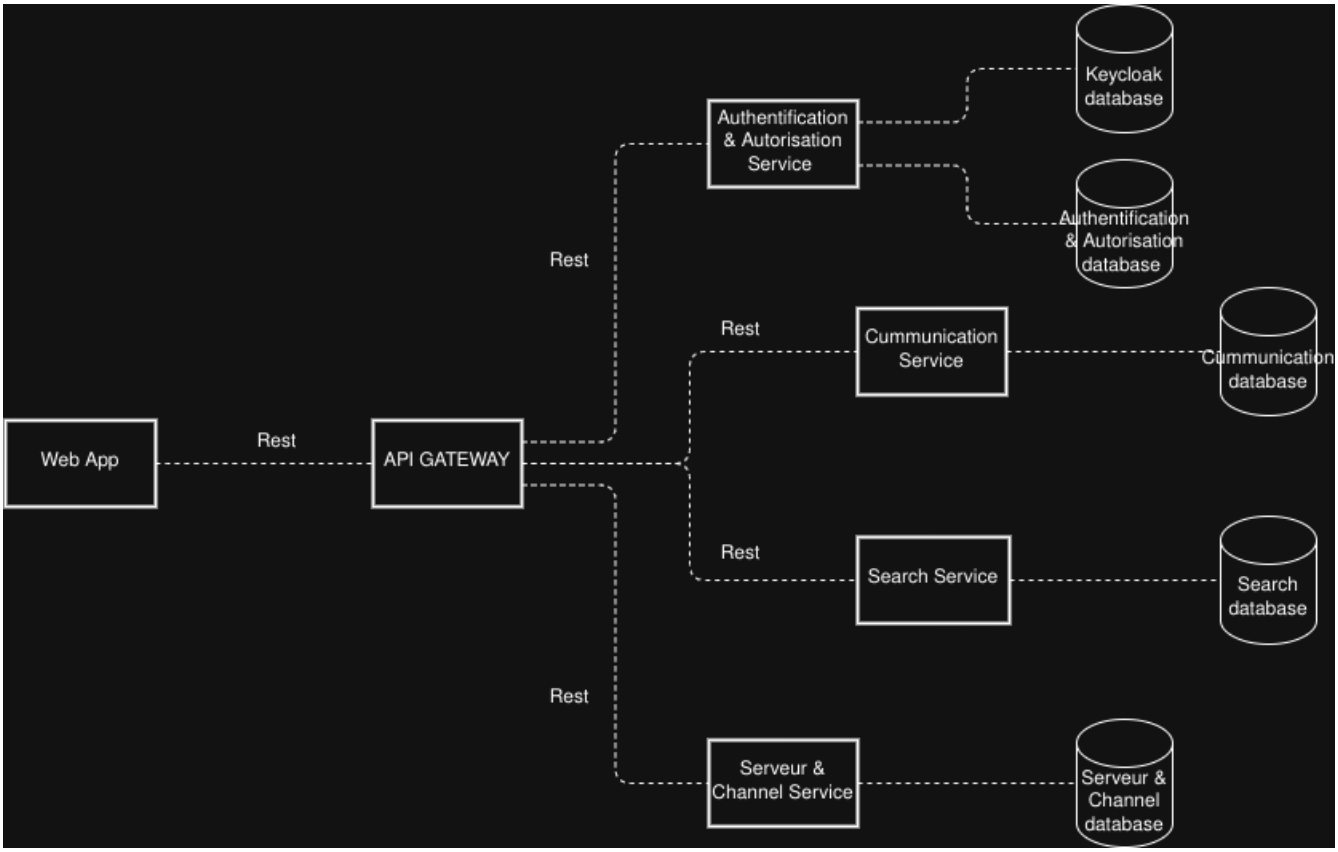

Figure 1. Diagramme d’Architecture

2. Authentification avec OIDC

2.1 Plan d’Intégration OIDC :

L’authentification des utilisateurs repose sur **Keycloak**, un fournisseur OIDC qui gère les différentes méthodes de connexion. Les types d’authentification pris en charge sont : - **Vanilla** : Création et connexion avec un compte Beep (email/mot de passe). - **Polytech** : Authentification via le LDAP de Polytech en utilisant Keycloak comme passerelle OIDC. - **Google** : Connexion via Google OAuth, avec possibilité d’association avec un compte Beep existant.

L’objectif est d’assurer une authentification centralisée, sécurisée et extensible.

2.2 Déploiement de Keycloak :

Le schéma suivant illustre le déploiement des composants impliqués dans l’authentification OIDC :

2.2.1 Composants :

- **Frontend** : Interface utilisateur (React).
- **Backend** : API Beep qui communique avec Keycloak et la base de données.

- **Keycloak** : Gère l'authentification OIDC et communique avec les fournisseurs externes.
- **Database** : Stocke les informations des utilisateurs (hors identifiants).
- **Polytech LDAP** : Service externe pour l'authentification des étudiants et enseignants de Polytech.
- **Google OAuth** : Service externe pour l'authentification Google.

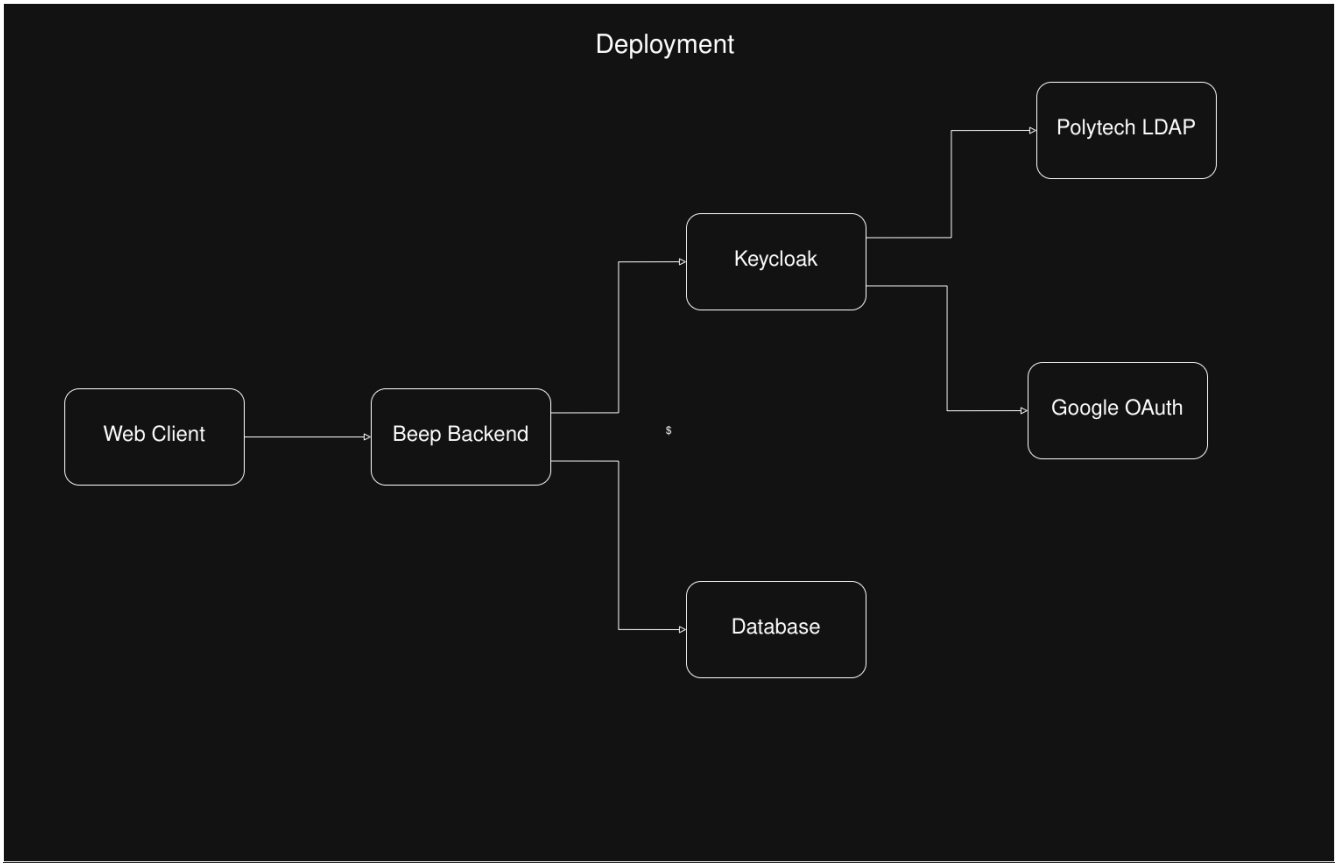

Figure 2. Diagramme de Déploiement

2.2.1 Flux de communication :

1. L'utilisateur initie une connexion ou une inscription via le **Frontend**.
2. Le **Backend** redirige vers **Keycloak** pour authentification.
3. **Keycloak** traite l'authentification selon le fournisseur choisi :
   - Directement pour un compte Vanilla.
   - Via **Polytech LDAP** pour une connexion universitaire.
   - Via **Google OAuth** pour une connexion Google.
4. Une fois authentifié, **Keycloak** retourne un **token JWT** au **Backend**.
5. Le **Backend** vérifie l'identité et accorde une session utilisateur.

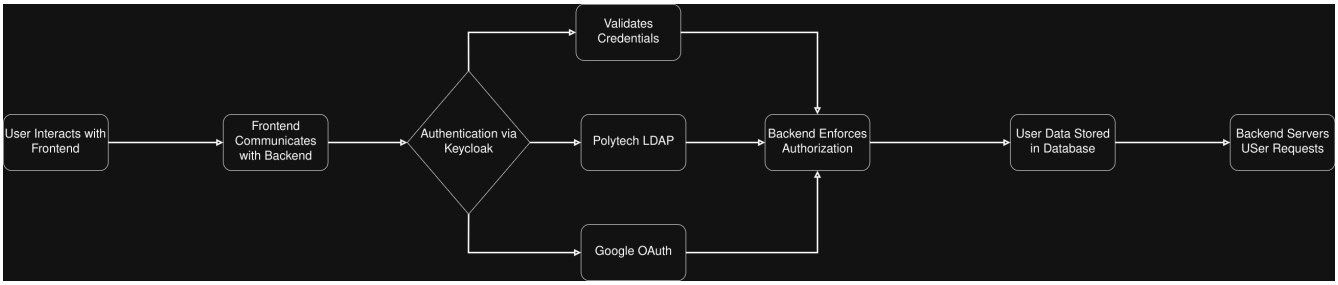

Figure 3. Flux de Communication

2.3 Flux d'Authentification :

Ces diagrammes illustrent les différentes étapes du processus d'authentification selon le type de compte.

2.3.1 Création de compte utilisateur :

- **Vanilla** : L'utilisateur fournit un email et un mot de passe, Keycloak gère la création et stocke les données dans la base de données.
- **Polytech** : Redirection vers **Polytech LDAP** via **Keycloak** pour validation des identifiants.
- **Google** : Redirection vers **Google OAuth**, qui renvoie un jeton validé par **Keycloak**.

2.3.2 Connexion utilisateur :

- **Vanilla** : L'utilisateur soumet ses identifiants, qui sont validés par **Keycloak**.
- **Polytech** : Redirection vers **Polytech LDAP**, qui valide et retourne un jeton à **Keycloak**.

2.3.3 Association d'un compte Google :

1. L'utilisateur connecté initie une liaison avec son compte Google.
2. Redirection vers **Google OAuth** pour validation.
3. Retour du jeton à **Keycloak**.
4. Mise à jour du compte Beep pour inclure l'identité Google.

Ces éléments garantissent une gestion flexible et sécurisée de l'authentification pour Beep.

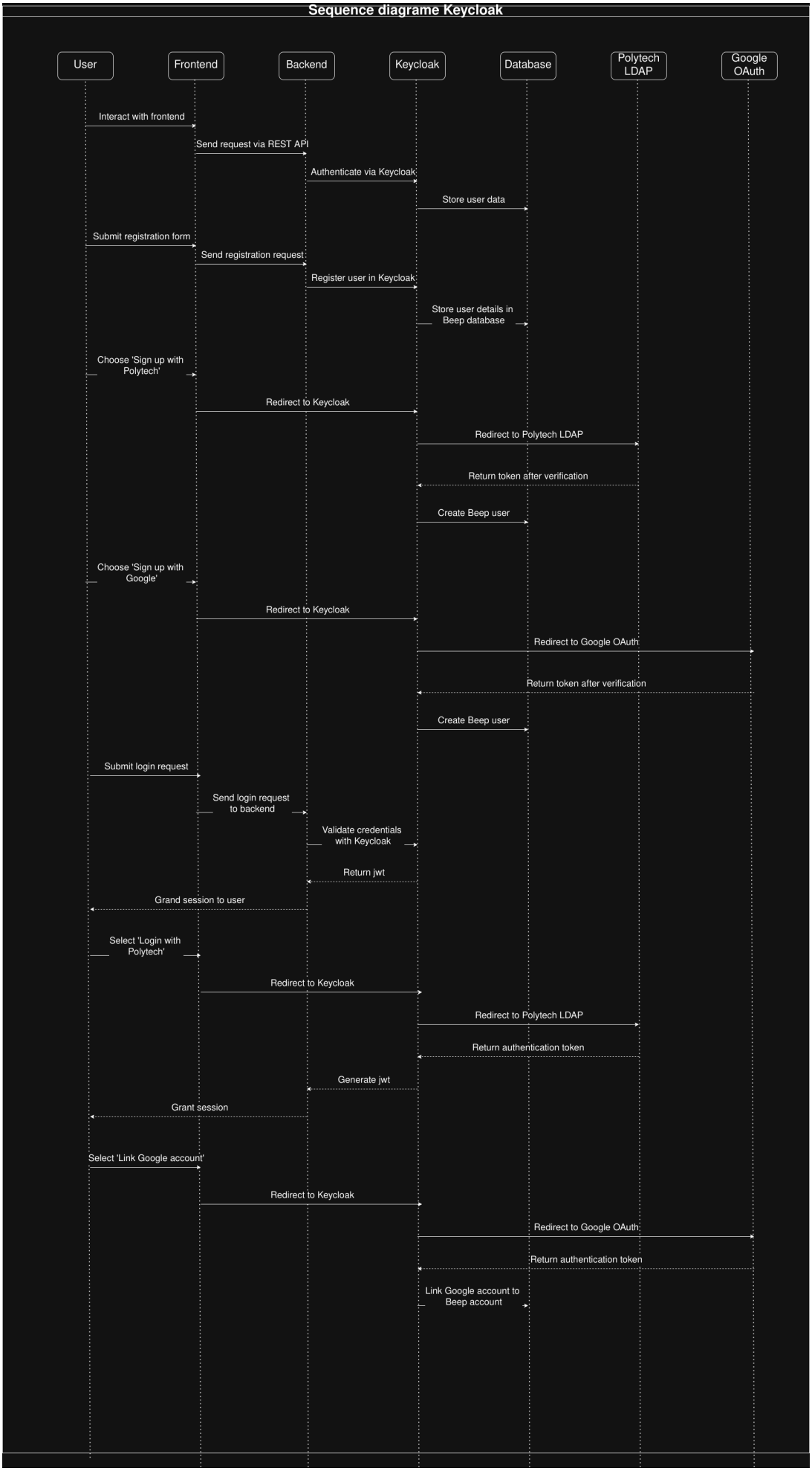

Figure 4. Diagrammes de Séquence

3. Communication Inter-Microservices

3.1 Stratégie de Communication :

Pour une application de messagerie en temps réel comme Beep, l'architecture doit prioriser la faible latence, la haute disponibilité et la scalabilité.

3.1.1 Service d'Authentification:

APIs REST/gRPC : Les APIs REST sont utilisées pour leur simplicité et leur compatibilité avec les navigateurs web, facilitant ainsi l'intégration avec des applications front-end. gRPC est utilisé pour la communication inter-services en raison de sa performance et de sa capacité à gérer des appels de procédure distante de manière efficace.

3.1.2 Service de Messages:

gRPC : gRPC est choisi pour sa performance et sa capacité à gérer des communications bidirectionnelles, ce qui est crucial pour les opérations CRUD sur les messages où la latence doit être minimale.

Redis : Redis est utilisé pour le cache afin de réduire la latence des opérations de lecture fréquentes, améliorant ainsi la réactivité de l'application.

3.1.3 Service de Présence:

WebSockets : Les WebSockets permettent une communication bidirectionnelle en temps réel entre le client et le serveur, essentielle pour maintenir l'état de présence des utilisateurs à jour en temps réel.

Redis Pub/Sub : Redis Pub/Sub est utilisé pour la diffusion d'états de présence entre les instances du service, assurant que toutes les instances ont une vue cohérente de l'état des utilisateurs.

3.1.4 Service de Notifications:

WebSocket/Push : Les WebSockets et les notifications push sont utilisés pour envoyer des notifications en temps réel aux utilisateurs, que ce soit sur le web ou sur des appareils mobiles.

3.1.5 Service de Médias:

API REST : Une API REST est utilisée pour les uploads de médias en raison de sa simplicité et de sa compatibilité avec les navigateurs web, facilitant ainsi l'intégration avec les interfaces utilisateur.

Kafka : kafka est utilisé pour le traitement asynchrone, permettant de découpler le traitement des images et des vidéos de l'upload initial, ce qui améliore la réactivité de l'application.

3.1.6 Service de Serveurs/Channel:

gRPC et REST : gRPC est utilisé pour la communication inter-services en raison de sa performance,

tandis que REST est utilisé pour les interactions avec les clients en raison de sa simplicité et de sa compatibilité.

3.2 Diagramme de Séquence :

Voici le diagramme de séquence résumant les interactions entre les différents microservices lors de l'envoi d'un message par un utilisateur.

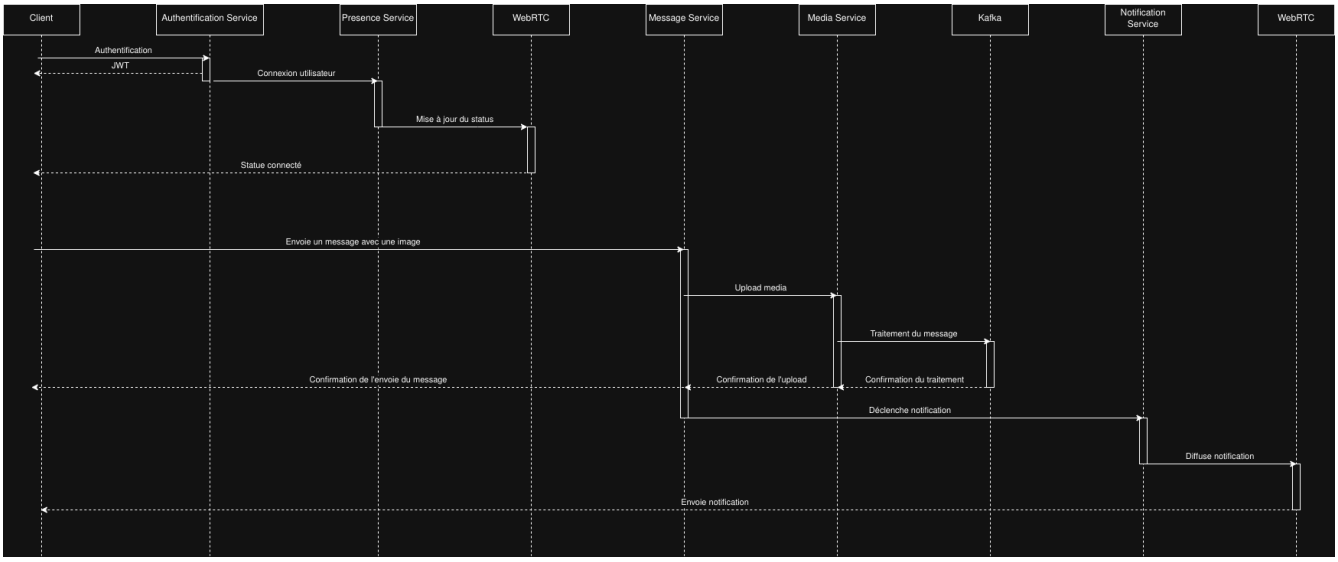

Figure 5. Diagramme de Séquence de Communication

4. Système d'Autorisation

4.1 Définition du Service d'Autorisation

Un **service d'autorisation** est un module permettant de contrôler l'accès aux ressources de l'application en fonction des rôles et permissions des utilisateurs. Il joue un rôle central dans la sécurité et la gestion des accès.

Objectifs du Service d'Autorisation :

- Assurer une gestion granulaire des permissions à différents niveaux :
- **Serveur** (ex : propriétaire, modérateur, membre).
- **Catégorie** (ex : accès à certaines sections d'un serveur).
- **Channel** (ex : lecture/écriture des messages, envoi de fichiers).
- Permettre l'attribution de **permissions globales** aux administrateurs de la plateforme.
- Offrir une **scalabilité** et une **interopérabilité** avec d'autres services via des API sécurisées.

4.2 Architecture Fonctionnelle

L'architecture fonctionnelle repose sur un modèle **RBAC (Role-Based Access Control)**, structuré de la manière suivante :

1. **Gestion des rôles et permissions**
   - Rôles définis par serveur (ex : propriétaire, modérateur, membre).
   - Permissions assignées aux catégories et channels.
   - Gestion des rôles globaux pour les administrateurs de la plateforme.
2. **Validation des accès**
   - Lorsqu'un utilisateur tente d'accéder à une ressource, son rôle et ses permissions sont vérifiés.
   - Un refus d'accès est retourné si l'utilisateur n'a pas les droits nécessaires.
3. **Administration et gestion dynamique**
   - Interface permettant aux administrateurs de modifier les rôles et permissions en temps réel.

4.3 Architecture Technique

L'implémentation technique repose sur une approche **microservices** avec un **service d'autorisation centralisé**.

4.3.1 Composants principaux :

- **Service d'authentification (Keycloak)** : Gestion des utilisateurs et rôles via OpenID Connect (OIDC).
- **Service d'autorisation** : Vérifie les permissions et applique les règles de contrôle d'accès.
- **Base de données des permissions** : Stocke les règles et associations utilisateurs-serveurs-channels.
- **Cache (Redis)** : Accélère la vérification des permissions.

4.3.2 Technologies Recommandées :

- **Base de données** : PostgreSQL
- **Cache** : Redis
- **IAM & OIDC** : Keycloak
- **Communication interservices** : gRPC / REST API

4.3.3 Flux Technique :

1. L'utilisateur effectue une action nécessitant une autorisation.
2. Le service d'autorisation interroge la base des permissions ou le cache.
3. Une réponse est envoyée au backend avec un **statut d'autorisation (autorisé/refusé)**.

4.4 Diagrammes de Séquence

Les diagrammes suivants illustrent le fonctionnement du système dans les cas d'usage principaux.

4.4.1 Création d'un Channel

Acteurs : Utilisateur, Backend, Service d'Autorisation, Base de données. 1. L'utilisateur crée un channel via l'interface. 2. Le backend contacte le service d'autorisation pour initialiser les permissions. 3. Les permissions sont stockées en base et mises en cache.

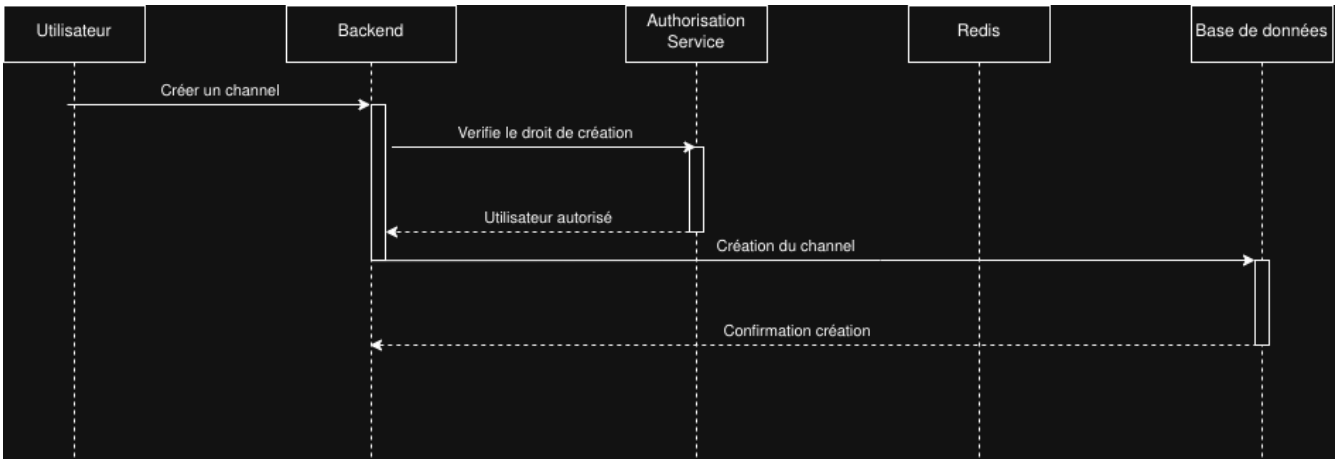

Figure 6. Diagramme de Séquence : Gestion des permission lors de la création d'un Channel

4.4.2 Ajout d'un Utilisateur à un Serveur avec un Rôle Spécifique

Acteurs : Administrateur du serveur, Backend, Service d'Autorisation. 1. L'administrateur attribue un rôle (ex : modérateur) à un utilisateur. 2. Le service d'autorisation met à jour les permissions en base et en cache.

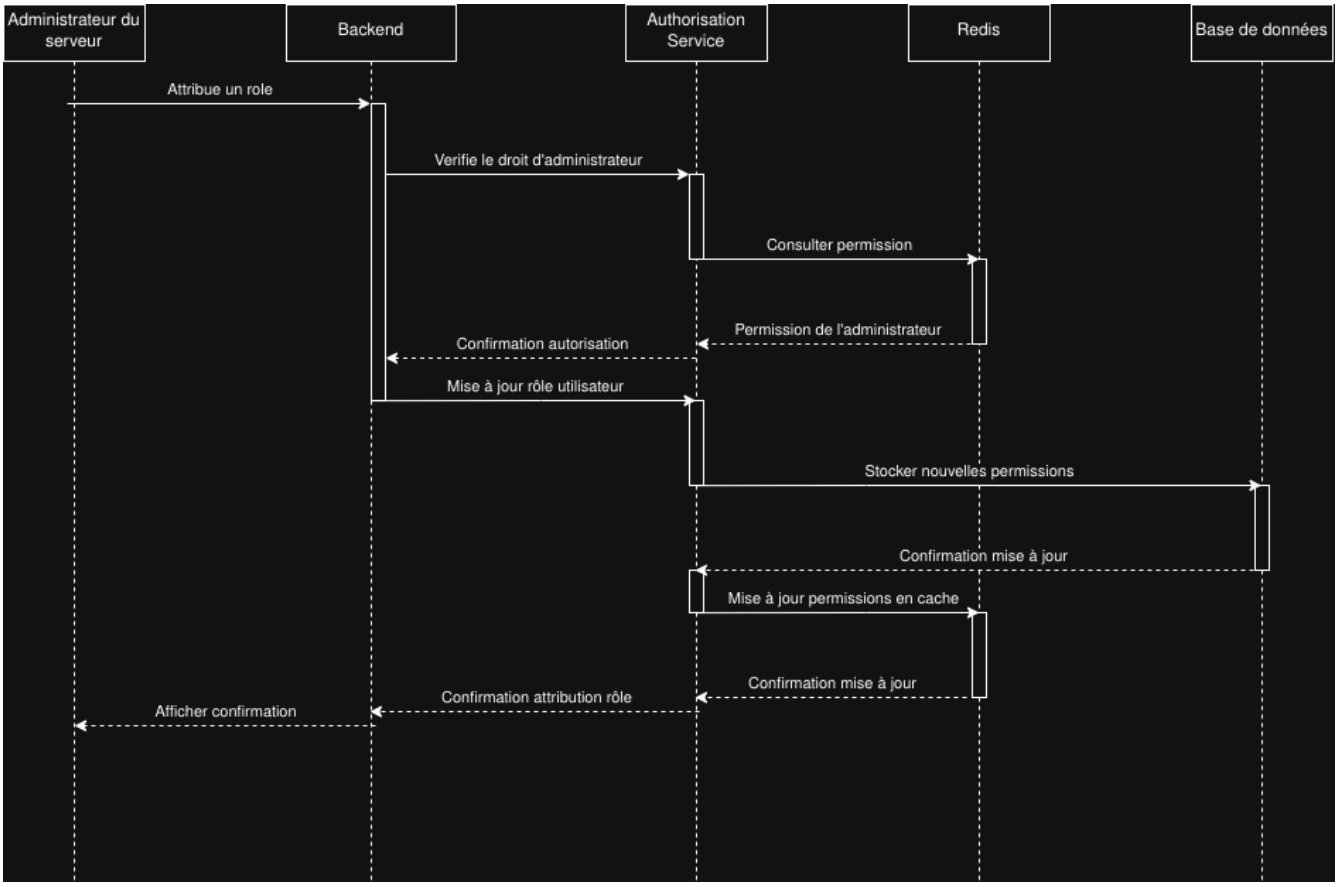

Figure 7. Diagramme de Séquence : Attribution d'un Rôle à un Utilisateur

Architecture du Service d'Autorisation :

- Gestion des permissions par serveur, catégorie et canal.
- Permissions globales pour les administrateurs.

5. Journalisation et Traçabilité

5.1 Plan d'Observabilité :

- Utilisation de la journalisation et de la traçabilité centralisées.
- Définition des logs de sécurité.

5.2 Diagrammes de Déploiement et de Séquence :

(À créer sur Draw.io)

6. Prêt pour la Production

6.1 Sujets Couverts :

- Sécurité des données et sauvegardes.

- Observabilité des services.
- Stratégies de haute disponibilité.

6.2 Diagramme Cible :

(À créer sur Draw.io)

7. Sécurité de l’Infrastructure

7.1 Authentification Mobile :

- Définition du processus d’authentification mobile.

7.2 Architecture de Sécurité :

- Séparation DMZ, cryptographie et protocoles utilisés.

8. Moteur de Recherche

8.1 Proposition Fonctionnelle :

- Mise en place d’une recherche en texte intégral.

8.2 Maquette UI :

(À créer sur Draw.io)

8.3 Diagramme de Séquence :

- Flux de requête de recherche.

Conclusion

Résumé des décisions architecturales clés et prochaines étapes.