TAD - Beep

Louis LABEYRIE

Table of Contents

Chapter 1. Concepts fondamentaux et décisions d'architecture (ADR)

Cette partie du document présente les modèles conceptuels qui sous-tendent l'application Beep, ainsi que les rôles métiers et les premières décisions d'architecture retenues.

Elle sert de fondation pour les vues logiques et techniques décrites dans les sections suivantes.

1.1. Introduction

1.1.1. Objectifs du document

Ce document présente l'architecture technique de l'application **Beep**, une messagerie en temps réel conçue pour offrir une expérience fluide, sécurisée et évolutive, à destination d'un large public.

Il a pour objectif de :

- Décrire les choix structurants de l'architecture de l'application
- Formaliser les contraintes fonctionnelles et techniques
- Proposer une architecture modulaire, maintenable et scalable
- Servir de référence pour les équipes de développement, de déploiement et d'exploitation

Ces objectifs structurent l'ensemble du document et constituent le fil conducteur des choix d'architecture et de modélisation qui seront détaillés dans les sections méthodologiques et techniques à venir.

1.1.2. Portée du document

Le périmètre ainsi défini permet de concentrer le présent document sur les aspects d'architecture logicielle et technique essentiels au bon fonctionnement et à l'évolution de la plateforme.

Ce document couvre :

- L'organisation fonctionnelle de l'application et ses rôles métiers
- Le découpage en services logiques et les interactions inter-domaines
- Les choix technologiques pour l'infrastructure, la sécurité, la supervision et la résilience
- Les mesures d'observabilité, de sécurité et de protection des données
- Les orientations pour le déploiement, la continuité de service et l'évolution

Il ne traite pas :

- Du design graphique ou de l'interface utilisateur
- Des décisions commerciales ou de marketing
- Des détails d'implémentation ou de code

1.1.3. Public cible

Le présent document s'adresse principalement :

- Aux architectes logiciels et techniques
- Aux ingénieurs DevOps et SRE
- Aux développeurs impliqués dans le projet
- Aux responsables de l'infrastructure
- À toute personne participant à l'analyse, au pilotage ou à l'évolution du système

1.1.4. Méthodologie

La démarche de conception et de formalisation de ce document suit une approche structurée, alignée avec les bonnes pratiques d'architecture logicielle.

Le document est organisé en **vues complémentaires**, permettant de couvrir de manière cohérente les différents aspects du système :

- **Vue conceptuelle** : définit les modèles métier fondamentaux, les rôles et les entités manipulées, indépendamment des choix techniques.
- **Vue logique** : décrit le découpage fonctionnel en services et les interactions métier.
- **Vue technique** : détaille les choix technologiques, les architectures de déploiement, les patterns de communication et les mesures de sécurité.

Les décisions d'architecture ont été prises selon les principes suivants :

- **Alignement sur les besoins métier** : répondre efficacement aux cas d'usage attendus par les utilisateurs.
- **Robustesse et sécurité** : garantir un niveau de sécurité élevé dans un contexte multi-tenant et public.
- **Scalabilité maîtrisée** : permettre une montée en charge fluide des services critiques.
- **Observabilité et opérabilité** : assurer la visibilité, la supervision et la maintenabilité du système en production.
- **Conformité aux contraintes de l'exercice pédagogique** : respecter les limitations imposées (absence de broker, architecture REST-first, etc.).

Le document est évolutif : il est enrichi au fil des itérations de conception et de mise en œuvre, avec l'objectif de rester une **référence vivante** pour l'ensemble des parties prenantes du projet.

1.1.5. Historique

Version	Date	Auteur	Remarques
0.1	2025-05-27	Louis Labeyrie	Version initiale de l'introduction

1.2. Modèle conceptuel métier

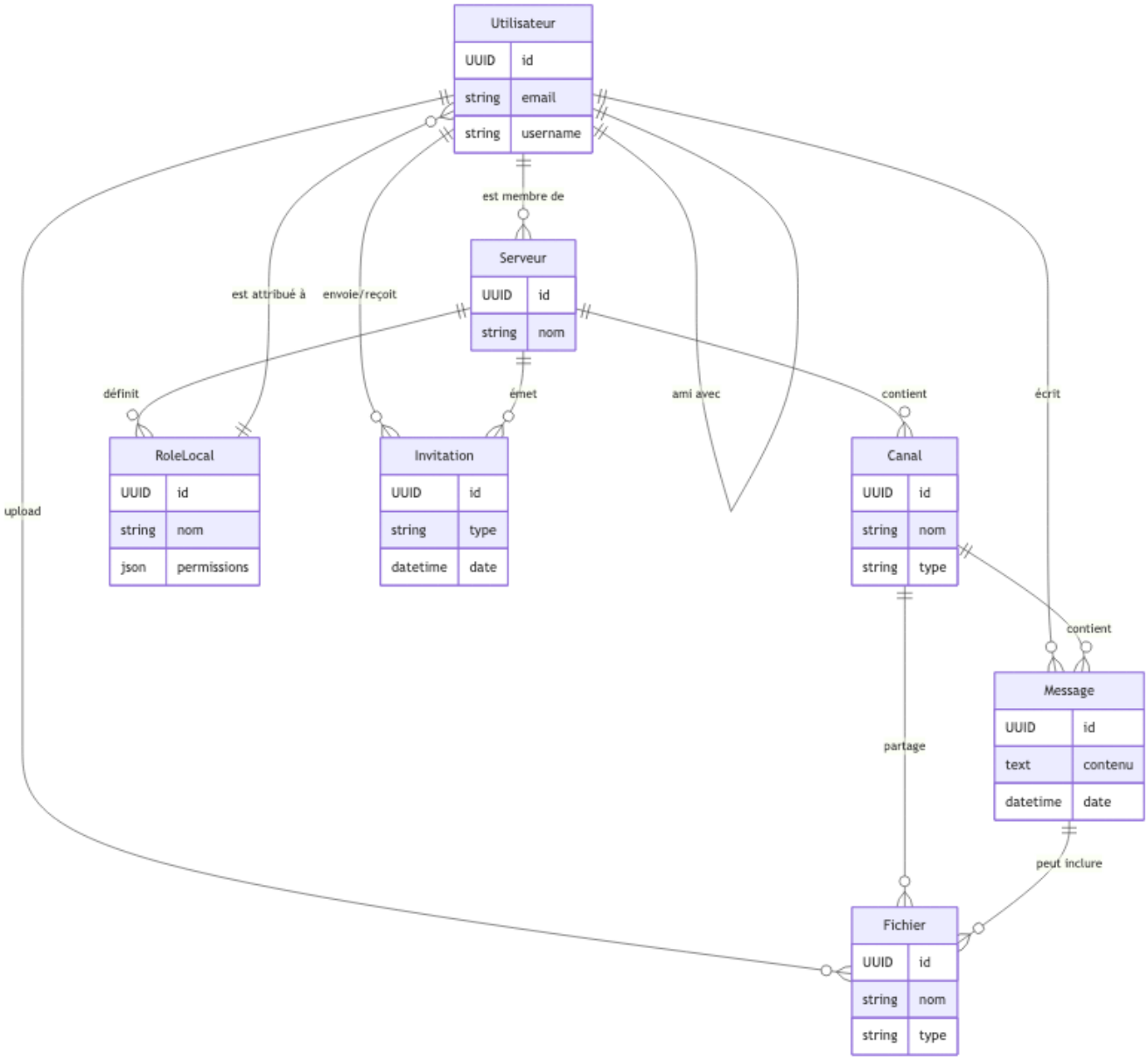
1.2.1. Objectif

Cette section présente les entités métier fondamentales manipulées par l'application Beep, ainsi que leurs relations conceptuelles. L'objectif est d'expliciter les concepts fonctionnels de haut niveau, indépendamment de toute considération technique ou d'implémentation.

1.2.2. Entités principales

Entité	Description
`Utilisateur`	Représente un compte personnel sur la plateforme. Un utilisateur peut rejoindre des serveurs, envoyer des messages, créer des canaux, etc.
`Serveur`	Espace communautaire privé ou public, créé par un utilisateur (owner). Contient des canaux, des membres et une configuration de rôles personnalisée.
`Canal`	Espace de communication (texte, vocal, ou autre) appartenant à un serveur. Peut être public ou restreint à certains rôles.
`Message`	Contenu textuel ou multimédia posté par un utilisateur dans un canal.
`Rôle local`	Ensemble de permissions définies dans un serveur et attribuées à un ou plusieurs utilisateurs.
`Invitation`	Requête d'amitié entre deux utilisateurs, ou d'accès à un serveur.
`Relation d'amitié`	Lien réciproque entre deux utilisateurs permettant une communication directe.
`Fichier`	Élément partagé dans un canal ou en message privé (image, PDF, etc.)

1.2.3. Relations entre entités



1.2.4. Éléments notables

- Un `Utilisateur` peut appartenir à plusieurs `Serveurs`
- Un `Serveur` possède plusieurs `Canaux`, `Messages`, `Rôles`, et `Membres`
- Un `Rôle` est défini dans le périmètre d'un `Serveur` uniquement
- Une `Invitation` peut cibler un autre utilisateur (demande d'amitié) ou un serveur (invitation à rejoindre)
- Les `Messages` peuvent être supprimés, modifiés ou signalés (cf. règles métier)

1.3. Modèle de rôles

1.3.1. Présentation générale

L'application Beep repose sur un système de rôles hiérarchiques permettant de gérer les droits et responsabilités des utilisateurs à deux niveaux distincts :

- des **rôles globaux**, valables sur toute la plateforme
- des **rôles locaux**, propres à chaque serveur d'échange

Cette séparation garantit à la fois la cohérence globale du système et la flexibilité dans la gestion des espaces communautaires.

1.3.2. Rôles globaux (plateforme)

Les rôles globaux définissent les droits d'un utilisateur à l'échelle de l'application Beep. Ils sont peu nombreux et strictement définis :

Rôle	Description
`user`	Rôle par défaut attribué à tout utilisateur authentifié. Donne accès aux fonctionnalités sociales, aux serveurs, à la communication et à la recherche.
`admin`	Rôle attribué à un superviseur de la plateforme. Peut intervenir sur les comptes utilisateurs (support, modération), accéder aux outils d'audit ou de monitoring globaux.

1.3.3. Rôles locaux (serveur)

Chaque **serveur d'échange** fonctionne comme un sous-espace autonome dans lequel les utilisateurs peuvent se voir attribuer des rôles spécifiques. Ces rôles sont :

- Créés et gérés par les utilisateurs disposant des droits adéquats
- Stockés dans la configuration locale du serveur
- Associés à des permissions granulaires

1.3.3.1. Rôles par défaut

Rôle	Description
`owner`	Propriétaire du serveur. Il possède tous les droits, y compris la suppression du serveur, la gestion des utilisateurs et des rôles.
`default`	Rôle attribué automatiquement à tout nouvel arrivant sur le serveur. Ses permissions sont configurables par le `owner`.

1.3.3.2. Rôles personnalisés

Le `owner` peut : - Créer des rôles personnalisés (ex: `moderator`, `content-manager`, `role-manager`) - Associer à chaque rôle un ensemble de permissions : lecture, écriture, gestion des canaux, gestion des rôles, etc. - Déléguer la **gestion des rôles** à certains utilisateurs (ex: via un rôle nommé `role-manager`)

Cette approche permet de : - Créer une **hiérarchie locale propre à chaque communauté** - Appliquer le **principe du moindre privilège** - Faciliter la **modération collaborative** ou thématique

1.3.4. Exemples de rôles locaux

Exemple de rôle	Description
`moderator`	Peut supprimer des messages, gérer les utilisateurs dans un canal
`role-manager`	Peut créer de nouveaux rôles et les attribuer à d'autres membres

1.3.5. Portée et cumul

Un utilisateur peut : - Avoir un rôle global (`user` ou `admin`) - Être membre de plusieurs serveurs avec des rôles locaux différents - Cumuler plusieurs rôles locaux si la configuration du serveur le permet

Chapter 2. Vue applicative

Cette section décrit l'application Beep du point de vue des **usages métier** et de l'expérience utilisateur.

Elle formalise les cas d'usage, les règles fonctionnelles et les interactions attendues, qui servent de référence pour le découpage en services et les choix techniques détaillés ultérieurement.

2.1. Cas d'usage de l'application Beep

2.1.1. Objectif

Cette section décrit les principales fonctionnalités accessibles aux utilisateurs à travers des cas d'usage représentatifs. Ces cas permettent d'illustrer les besoins fonctionnels, les interactions entre l'utilisateur et le système, et de poser les bases du découpage en services.

2.1.2. Authentification et gestion de compte

2.1.2.1. En tant que visiteur (non connecté)

- Je veux créer un compte Beep avec une adresse e-mail et un mot de passe, afin d'accéder à l'application.
- Je veux créer un compte Beep via mon compte Google ou mon compte Polytech (LDAP), afin de simplifier la connexion.
- Je veux me connecter à mon compte Beep, quel que soit le mode de création initial.

2.1.2.2. En tant qu'utilisateur connecté

- Je veux pouvoir modifier mes informations personnelles (email, mot de passe, avatar…).
- Je veux associer mon compte Google ou Polytech (LDAP) à mon compte Beep existant pour faciliter les futures connexions.
- Je veux me déconnecter manuellement ou automatiquement après inactivité.
- Je veux pouvoir supprimer mon compte.

2.1.3. Gestion des amis et invitations

- Je veux envoyer une invitation à un autre utilisateur par son identifiant unique.
- Je veux accepter ou refuser une demande d'amitié.
- Je veux consulter ma liste d'amis, voir leur statut, et entamer une conversation directe.
- Je veux supprimer un ami si je ne souhaite plus interagir avec lui.

2.1.4. Communication via serveurs, canaux et messages

2.1.4.1. Serveurs

- Je veux créer un serveur pour regrouper une communauté.
- Je veux inviter des amis à rejoindre mon serveur via un lien ou un identifiant.
- Je veux configurer les rôles et les permissions dans mon serveur.
- Je veux pouvoir quitter un serveur.

2.1.4.2. Canaux

- Je veux créer des canaux textuels ou vocaux dans un serveur.
- Je veux restreindre l'accès à certains canaux selon le rôle de l'utilisateur.
- Je veux modifier l'ordre et la hiérarchie des canaux.

2.1.4.3. Messages

- Je veux envoyer un message dans un canal auquel j'ai accès.
- Je veux supprimer ou modifier mes propres messages.
- Je veux épingler un message important ou y réagir via des émojis.
- Je veux partager des fichiers dans les canaux.

2.1.5. Gestion des rôles et permissions

- En tant que `owner`, je veux créer des rôles personnalisés dans mon serveur.
- En tant que `owner`, je veux déléguer la gestion des rôles à d'autres utilisateurs via un rôle intermédiaire.
- Je veux attribuer des rôles aux membres de mon serveur selon leurs responsabilités.
- Je veux restreindre l'accès à certaines actions (création de canal, suppression de message…) selon le rôle.
- Je veux pouvoir supprimer un rôle.
- Je veux pouvoir modifier un rôle.
- En tant que `owner`, je veux pouvoir supprimer un serveur.

2.1.6. Notifications et alertes

- Je veux recevoir une notification en temps réel lorsqu'un ami m'envoie un message.
- Je veux recevoir une notification lorsque je reçois une demande d'amitié.
- Je veux recevoir une notification lorsqu'on m'invite à rejoindre un serveur.
- Je veux recevoir une notification lorsqu'un message est épinglé.
- Je veux pouvoir activer ou désactiver certaines notifications (push ou in-app) dans mes préférences.

2.1.7. Recherche de contenu

- Je veux rechercher un utilisateur par son nom ou son identifiant.
- Je veux rechercher un message contenant un mot-clé dans les canaux auxquels j'ai accès.
- Je veux rechercher un fichier partagé dans un serveur donné.

2.1.8. Contraintes fonctionnelles

2.1.8.1. Performance et latence

- Le temps de propagation d'un message entre l'émetteur et les destinataires doit être inférieur à 500ms pour des conditions normales de connexion.
- Le temps de chargement initial de l'application doit être inférieur à 2 secondes.
- La synchronisation des messages doit être quasi-instantanée (moins de 100ms) entre les différents clients connectés.

2.1.8.2. Disponibilité et fiabilité

- L'application doit être disponible 99.9% du temps (hors maintenance planifiée).
- Les messages doivent être persistés de manière fiable, avec une garantie de non-perte.
- En cas de déconnexion temporaire, la reconnexion doit être automatique et transparente.

2.1.8.3. Sécurité

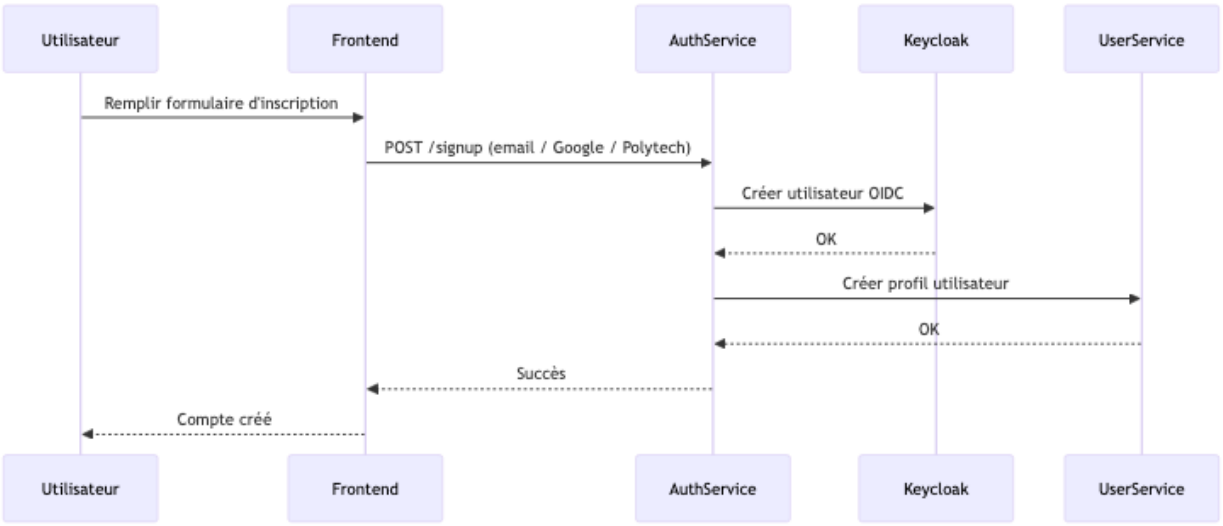
- Les messages privés doivent être chiffrés de bout en bout.
- Les sessions utilisateur doivent expirer après 24 heures d'inactivité.
- Les tentatives de connexion échouées doivent être limitées à 5 par minute par adresse IP.

2.1.8.4. Scalabilité

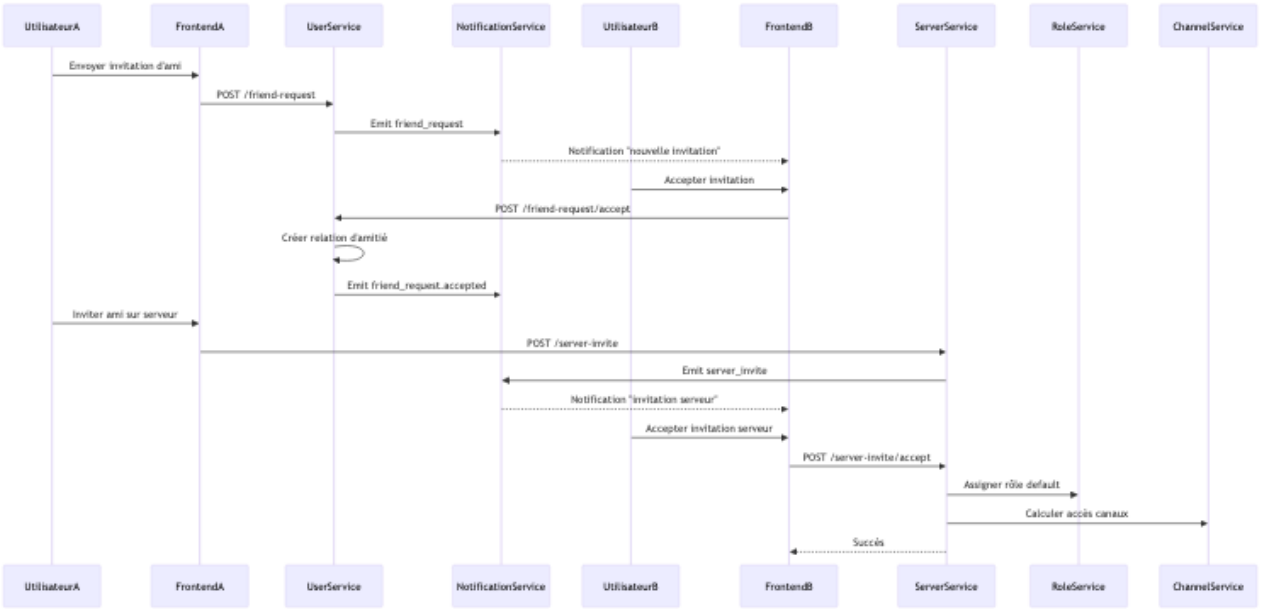
- L'application doit supporter jusqu'à 1000 utilisateurs simultanés par serveur.
- Un serveur peut contenir jusqu'à 100 canaux.

2.1.9. Illustration : Diagrammes de cas d'usage

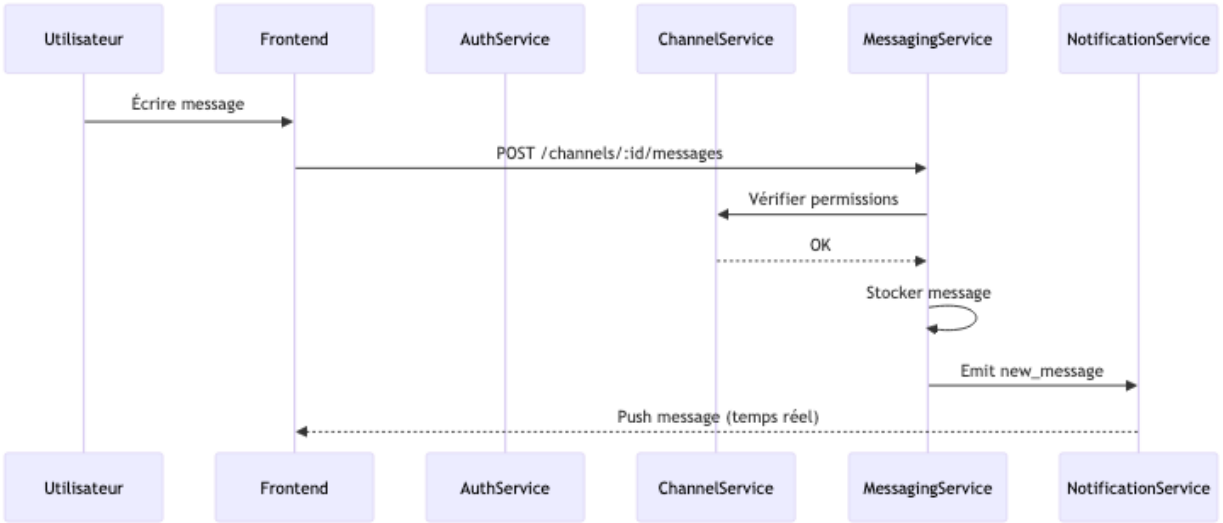
2.1.9.1. Diagramme : Création de compte (email / Google / Polytech)



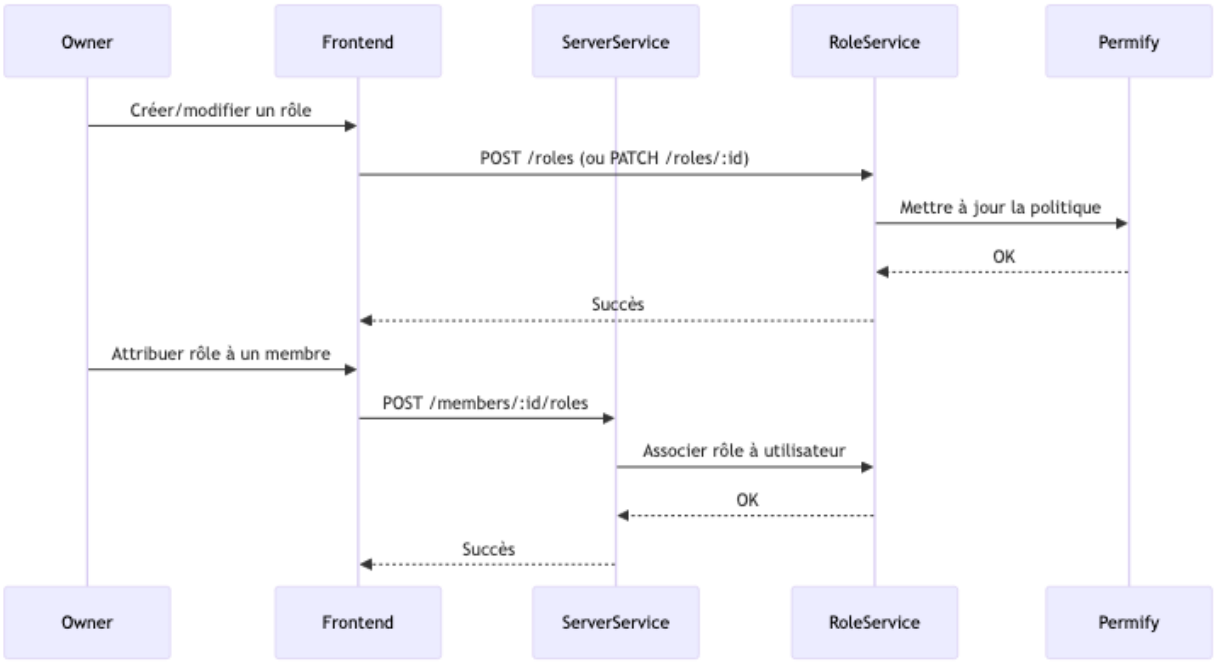
2.1.9.2. Diagramme : Invitation d'amis et ajout à un serveur



2.1.9.3. Diagramme : Envoi de message dans un canal privé



2.1.9.4. Diagramme : Gestion des rôles et des permissions dans un serveur



2.2. Règles métiers de Beep

2.2.1. Objectif

Cette section formalise les règles fonctionnelles qui encadrent les comportements attendus dans Beep. Ces règles visent à garantir la cohérence de l'expérience utilisateur, la sécurité logique de l'application, et la robustesse des interactions.

2.2.2. Règles de gestion des utilisateurs

- Une adresse e-mail ne peut être associée qu'à un seul compte Beep.
- Un utilisateur ne peut être ami qu'avec des utilisateurs ayant accepté son invitation.
- Un utilisateur ne peut pas s'envoyer d'invitation à lui-même.
- La suppression d'un utilisateur entraîne la suppression de ses messages (ou leur anonymisation selon les paramètres de confidentialité).
- Un utilisateur peut associer un seul compte Google à son compte Beep.
- Un utilisateur peut associer un seul compte Polytech (LDAP) à son compte Beep.
- Un compte Google ou Polytech (LDAP) ne peut être associé qu'à un seul compte Beep.

2.2.3. Règles liées aux serveurs

- Un utilisateur peut créer jusqu'à **5 serveurs** (limite initiale configurable).
- Un serveur doit toujours avoir un propriétaire (`owner`).
- La suppression du propriétaire entraîne le transfert de propriété ou la suppression du serveur.
- Un serveur ne peut pas avoir plus de 1000 membres.

- Un serveur ne peut pas avoir plus de 100 canaux.
- Un serveur ne peut pas avoir plus de 50 rôles.

2.2.4. Règles liées aux rôles et permissions

- Un serveur doit toujours contenir au moins deux rôles : `owner` et `default`.
- Les noms de rôles doivent être uniques au sein d'un serveur.
- Le rôle `owner` ne peut être supprimé ni réassigné sans délégation explicite.

2.2.5. Règles sur les canaux et la communication

- Un canal ne peut exister sans serveur parent.
- Seuls les utilisateurs ayant un rôle autorisé peuvent poster dans un canal restreint.
- Un message peut être modifié.
- Un message supprimé est définitivement retiré (pas de corbeille), sauf si archivé par un modérateur.
- Un fichier joint à un message est supprimé si le message est supprimé.

2.2.6. Règles de modération et sécurité

- Tout message signalé plus de 3 fois est caché en attente de validation.
- Seuls les utilisateurs avec le rôle global `admin` ou le rôle local avec la permission adaptée peuvent voir les messages cachés.

2.2.7. Règles transverses

- Les noms d'utilisateurs doivent :
  - Contenir entre 3 et 32 caractères
  - Ne pas contenir de caractères spéciaux (uniquement lettres, chiffres, tirets et underscores)
- Les noms de serveurs et de canaux doivent :
  - Contenir entre 2 et 50 caractères
  - Ne pas contenir de caractères spéciaux (uniquement lettres, chiffres, tirets et underscores)
- Les messages doivent :
  - Ne pas dépasser 2000 caractères
- Les fichiers joints doivent :
  - Ne pas dépasser 50 Mo par fichier

2.2.8. Perspectives d'évolution

Cette section pourra être enrichie ultérieurement avec :

- des règles liées à la notification

- des règles d’automatisation (bots, réponses automatiques, anti-spam…)
- des règles tarifaires ou de plan freemium si le produit évolue

Chapter 3. Vue logique

Cette section présente l'**architecture fonctionnelle** de Beep, en décrivant le découpage en services logiques, les responsabilités métier associées et les principaux flux d'interaction.

Elle constitue le socle de la modélisation technique et facilite l'alignement entre conception métier et implémentation.

3.1. Vue logique – Architecture fonctionnelle

3.1.1. Objectif

Cette section présente le découpage logique de l'application Beep en services ou modules fonctionnels. Elle permet de comprendre comment les responsabilités métier sont distribuées au sein du système, et d'identifier les composants principaux qui interagissent pour délivrer les fonctionnalités aux utilisateurs.

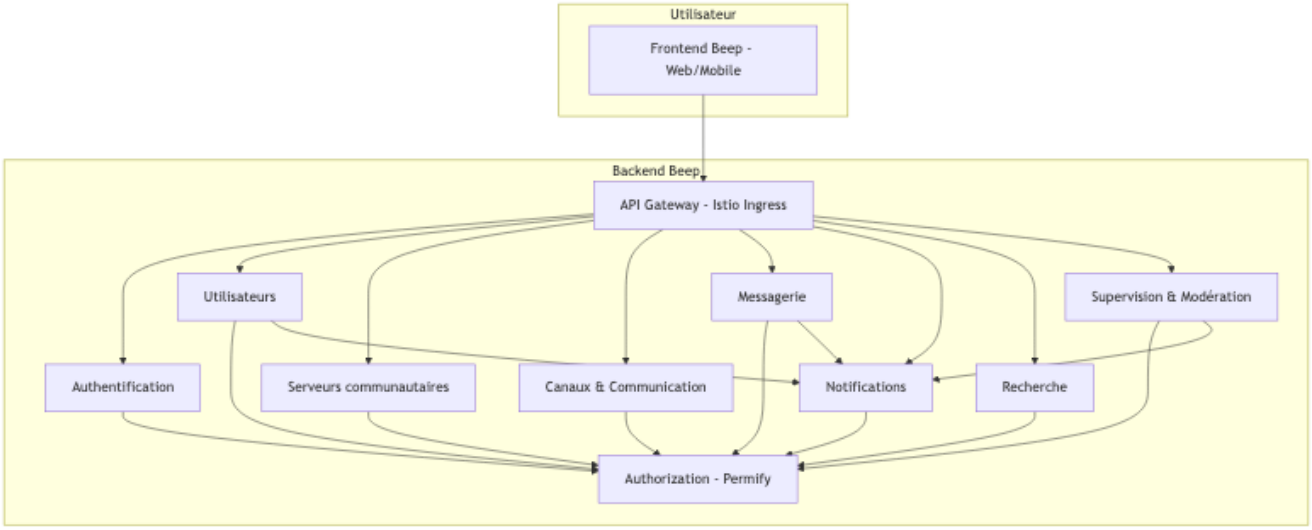
Cette vue sert de base aux vues techniques détaillées (architecture de déploiement, sécurisation, flux inter-services).

3.1.2. Services fonctionnels

Service	Responsabilités	Dépendances critiques
Authentification	Gère l'inscription, la connexion, les sessions, les providers externes (Google, Polytech), les tokens d'accès.	Utilisateurs, Authorization (Permify)
Gestion des utilisateurs	Gère les profils utilisateurs, les avatars, les préférences, la liste d'amis, les relations.	Authentification, Notifications
Serveurs communautaires	Gère la création de serveurs, la configuration des rôles (via Permify), la gestion des membres, les invitations.	Authentification, Utilisateurs, Authorization (Permify)
Canaux et communication	Gère les canaux (texte, vocal), les permissions par rôle, la hiérarchie des canaux.	Serveurs, Authorization (Permify)
Messagerie	Gère l'envoi, la réception, la modification et suppression des messages, ainsi que le partage de fichiers.	Canaux, Utilisateurs, Notifications, Authorization (Permify)
Notifications	Gère les alertes push, les notifications in-app, les statuts (temps réel via WebSocket).	Messagerie, Gestion des utilisateurs, Modération
Recherche	Permet de rechercher des utilisateurs, des serveurs, des messages, ou des fichiers.	Utilisateurs, Messagerie, Authorization (Permify)
Supervision & modération	Permet de signaler, masquer ou modérer des contenus, d'archiver ou supprimer des serveurs inactifs.	Messagerie, Utilisateurs, Serveurs, Notifications

Service	Responsabilités	Dépendances critiques
Authorization (Permify)	Service transverse d'évaluation des droits d'accès (ABAC/RBAC).	Tous les services métiers
Media Management	Gère le stockage et la gestion des fichiers.	Messagerie

3.1.3. Diagramme des composants (vue simplifiée)



3.1.4. Interactions entre services

Les services présentés ci-dessus interagissent selon différents modes de communication, qui seront détaillés dans la section dédiée aux interactions techniques.

- Les interactions synchrones sont majoritairement réalisées via des APIs REST, transitant par l'API Gateway (Istio Ingress).
- Les flux asynchrones (notifications, modération, supervision) reposent sur un mécanisme de publication/consommation d'événements (Redis Pub/Sub).
- Certaines données temporaires (sessions, présence) sont partagées via des caches distribués (Redis).

3.1.5. Bilan de la vue logique

- **Maturité des services** :
- Authentification, Gestion des utilisateurs, Messagerie : définis et stables
- Notifications, Recherche : implémentation initiale en place, approfondissement possible
- Supervision & modération : en cours de formalisation (flux à détailler)
- **Hypothèses technologiques validées en vue technique** :
- REST pour interactions synchrones (via API Gateway)
- Pub/Sub pour événements (Redis Pub/Sub, en conformité avec l'exercice : pas de broker)
- Cache distribué (Redis) pour sessions et présence

- Sécurité inter-service : mTLS, Service Mesh (Istio)

3.2. Flux fonctionnels entre services

3.2.1. Objectif

Cette section décrit les principaux flux d'interaction entre les services logiques de l'application Beep. Elle vise à illustrer les échanges récurrents de données ou d'appels, les dépendances fonctionnelles, et les circuits métier associés aux cas d'usage clés.

Les flux présentés ici préparent le terrain pour les vues techniques détaillées (ex: APIs, messages, événements asynchrones).

3.2.2. Méthodologie

Chaque flux est structuré selon :

- Le **contexte métier**
- Les **services impliqués**
- La **nature des interactions** (appel synchrone, publication d'événement, etc.)
- Un **diagramme de séquence ou de flux** illustratif

Toutes les interactions entre services sont protégées par mTLS via Istio (service mesh). Les flux asynchrones sont implémentés via Redis Pub/Sub (pas de broker central).

3.2.3. Flux 1 : Création de message dans un canal

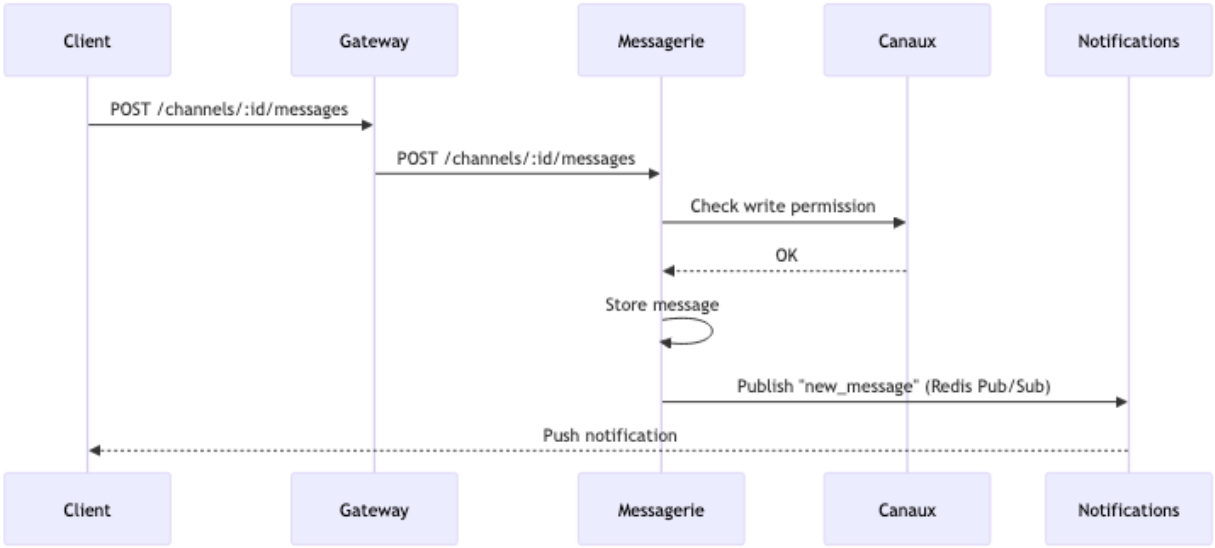
3.2.3.1. Contexte

Lorsqu'un utilisateur envoie un message dans un canal, plusieurs services interviennent pour vérifier les droits, stocker le message et en notifier les membres.

3.2.3.2. Services impliqués

- `API Gateway`
- `Messagerie`
- `Canaux`
- `Notifications`

3.2.3.3. Diagramme



3.2.4. Flux 2 : Ajout d'un ami

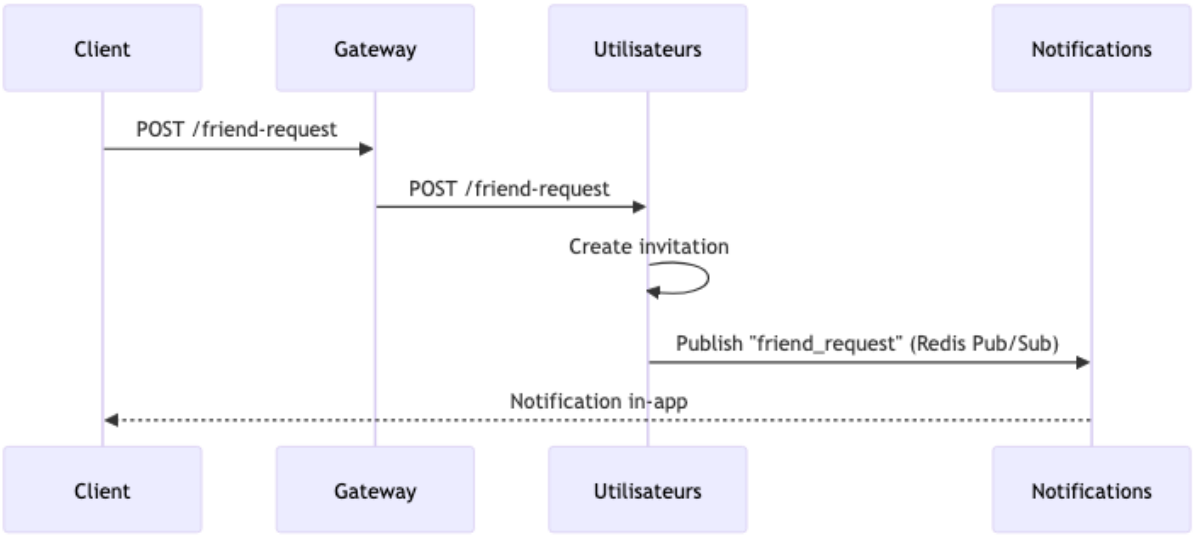
3.2.4.1. Contexte

Un utilisateur invite un autre utilisateur à devenir son ami.

3.2.4.2. Services impliqués

- `API Gateway`
- `Gestion des utilisateurs`
- `Notifications`

3.2.4.3. Diagramme



3.2.5. Flux 3 : Acceptation d'une invitation à un serveur

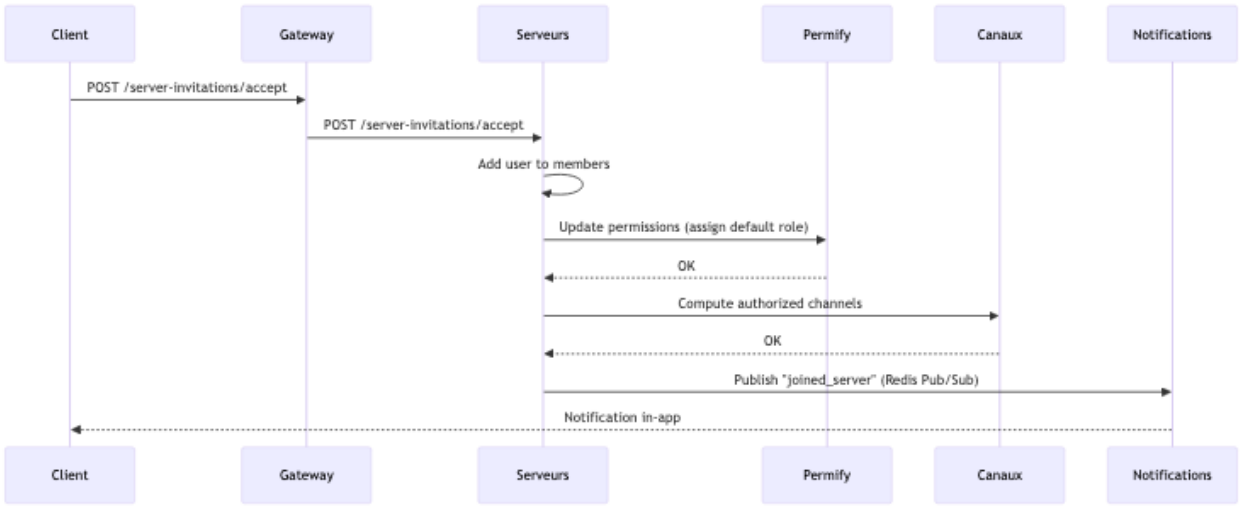
3.2.5.1. Contexte

Lorsqu'un utilisateur accepte une invitation, il devient membre d'un serveur.

3.2.5.2. Services impliqués

- API Gateway
- Serveurs communautaires
- Authorization Service (Permify)
- Canaux
- Notifications

3.2.5.3. Diagramme



3.2.6. Flux 4 : Signalement d'un message

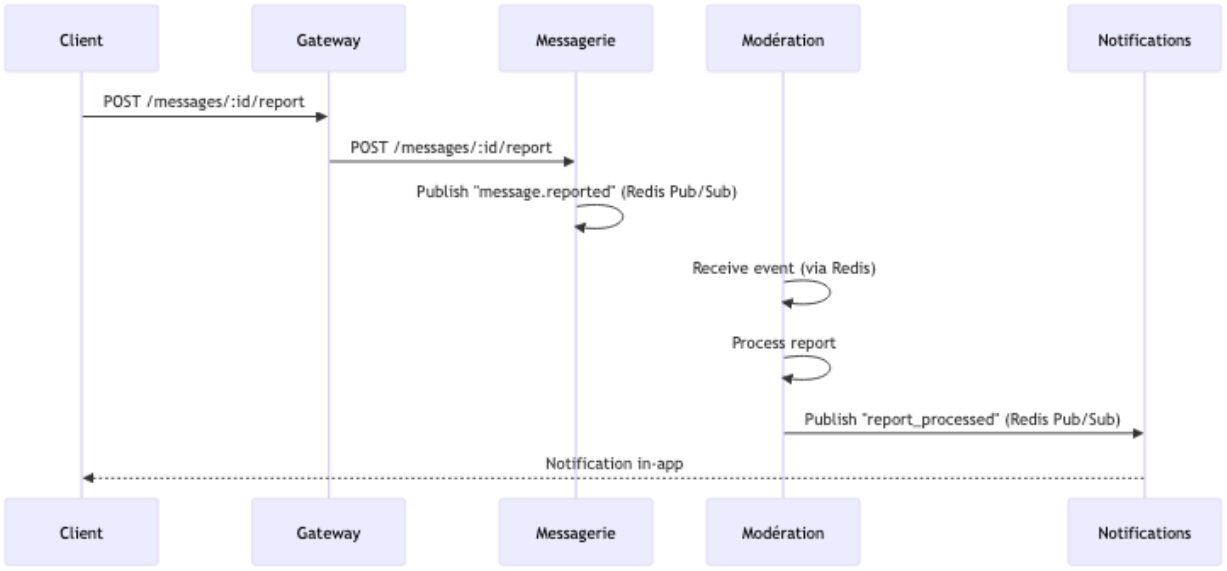
3.2.6.1. Contexte

Un utilisateur signale un message inapproprié.

3.2.6.2. Services impliqués

- API Gateway
- Messagerie
- Modération
- Notifications

3.2.6.3. Diagramme



3.2.7. Évolutions prévues

- Décrire les flux asynchrones supplémentaires (ex: archivage automatique des serveurs inactifs)
- Détail des flux liés à la gestion des rôles et permissions évoluées (multi-niveaux)
- Spécification des flux liés à la recherche avancée

3.3. Interactions techniques entre services

3.3.1. Objectif

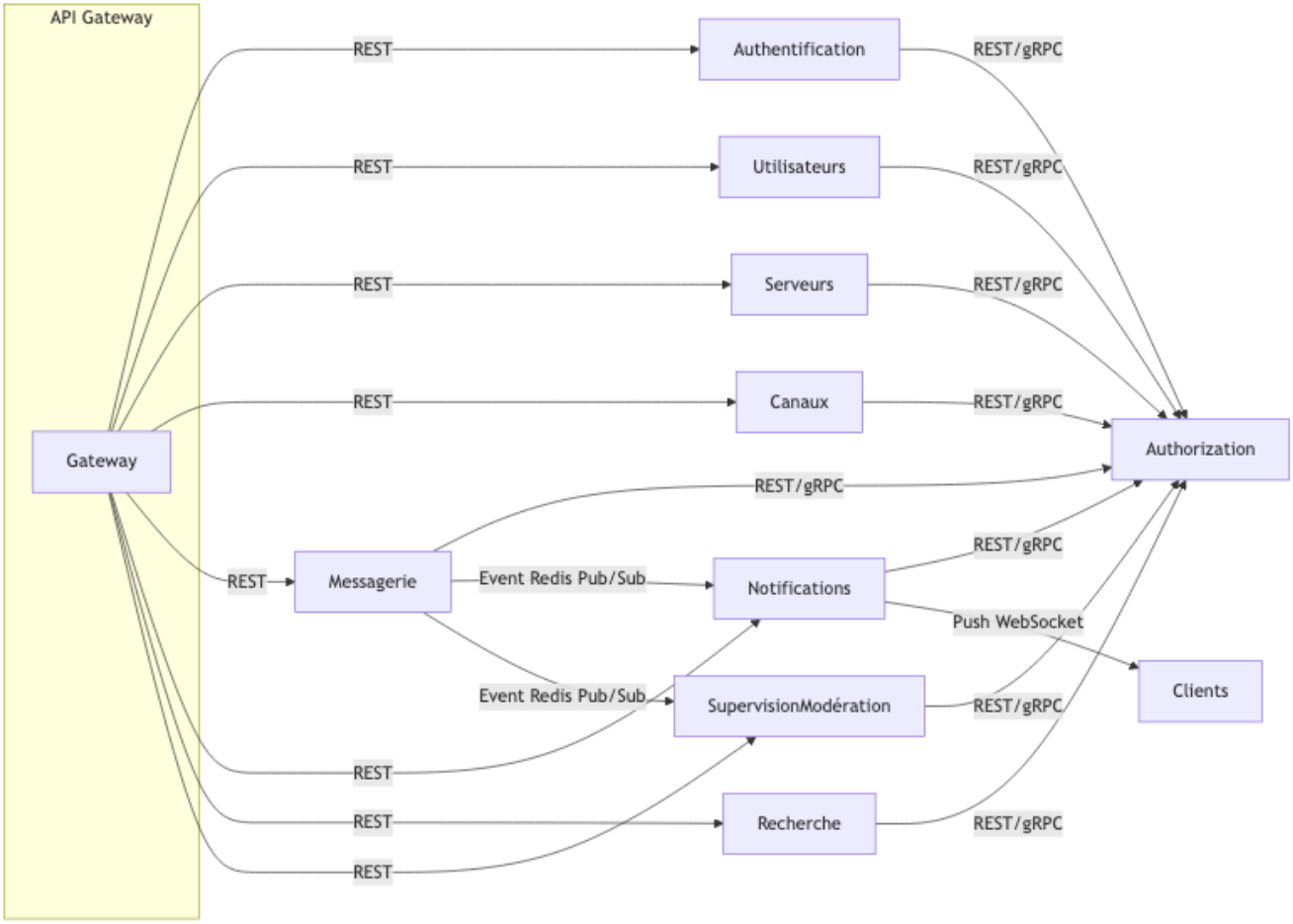
Cette section complète la vue logique en décrivant les interactions techniques précises entre les services de l'application Beep. Elle permet d'anticiper les contraintes d'intégration, de performance et de résilience en explicitant les mécanismes d'appel, les formats échangés et les points de couplage.

3.3.2. Principes de communication

Mode	Description
Appels REST synchrones (via API Gateway)	Utilisés pour les interactions classiques (authentification, gestion des utilisateurs, serveurs, canaux, messages, recherche, supervision & modération). Transitent par l'API Gateway sécurisée (Istio Ingress).
Événements pub/sub (Redis Pub/Sub)	Utilisés pour les notifications, la modération, et certains traitements asynchrones (ex : archivage automatique, supervision). En conformité avec les contraintes du projet (pas de broker centralisé type Kafka).
Partage de cache ou session (Redis)	Utilisé pour la gestion des sessions utilisateurs, la présence en ligne, et les informations temporaires à forte volatilité.
Service transverse d'autorisation (Permify)	Tous les services vérifient les droits d'accès métiers via des appels REST ou gRPC vers le service `Authorization (Permify)`.

Mode	Description
Base de données partagée (évité)	Chaque service possède sa propre base de données. Les accès croisés sont évités sauf nécessité métier stricte (ex: agrégation transverse dans la supervision).

3.3.3. Carte des interactions principales



3.3.4. Formats d’échange

3.3.4.1. REST – Exemple JSON : Création de message

```
POST /channels/:id/messages

{
  "authorId": "user-123",
  "content": "Salut tout le monde !",
  "attachments": []
}
```

Réponse :

```
{
  "id": "msg-456",
```

```
  "timestamp": "2025-05-27T15:00:00Z",
  "status": "created"
}
```

3.3.4.2. REST – Exemple JSON : Authentification utilisateur

```
POST /auth/login

{
  "email": "test@example.com",
  "password": "super-secret"
}
```

Réponse :

```
{
  "accessToken": "eyJhbGciOiJIUzI1...",
  "refreshToken": "def456...",
  "expiresIn": 3600
}
```

3.3.4.3. Événement – Nouveau message

```
event: "message.created"

{
  "channelId": "chan-789",
  "messageId": "msg-456",
  "authorId": "user-123",
  "type": "text"
}
```

3.3.4.4. Événement – Acceptation d'une demande d'ami

```
event: "friend_request.accepted"

{
  "userId": "user-123",
  "friendId": "user-456",
  "timestamp": "2025-05-27T16:00:00Z"
}
```

3.3.5. Sécurité des interactions

- Toutes les APIs REST sont protégées par JWT, transmis via l'API Gateway.

- Les appels inter-services utilisent mTLS (via Istio).
- Les droits d'accès sont vérifiés de manière transverse via `Authorization (Permify)`.
- Les événements pub/sub sont sécurisés par les contrôles d'abonnement et de consommation côté services.

3.3.6. Évolutions prévues

- Introduction de gRPC pour certaines interactions haute-fréquence ou critiques en performance (ex: Permission checks intensifs vers Permify, supervision en temps réel).
- Complétude de la spécification OpenAPI pour tous les endpoints publics et internes.
- Ajout éventuel de patterns de sagas / workflows distribués via des orchestrateurs légers compatibles (ex: Temporal.io, dans les limites des contraintes projet).

Chapter 4. Vue technique

Cette section détaille les **aspects techniques** de l'architecture de Beep : infrastructure, déploiement, sécurité, observabilité et intégration.

Elle traduit les choix logiques en solutions concrètes, en tenant compte des contraintes opérationnelles et des exigences de qualité de service.

4.1. Architecture technique cible

4.1.1. Objectif

Cette section présente l'architecture technique cible de la plateforme **Beep**, dans sa version microservices, en détaillant le découpage fonctionnel, les principes d'intégration, l'architecture de déploiement, ainsi que les modes de communication inter-services.

L'approche adoptée vise à garantir une architecture modulaire, sécurisée, observable et évolutive, cohérente avec les contraintes métier et les exigences de production de Beep.

4.1.2. Principes directeurs

Le passage à une architecture microservices est motivé par plusieurs objectifs structurants :

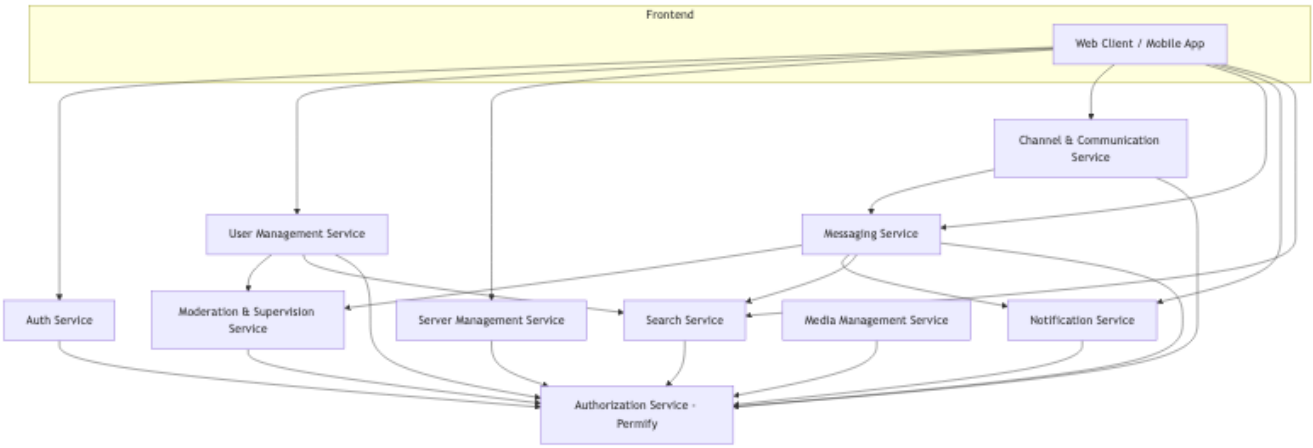
- **Modularité** : permettre l'évolution indépendante des différents domaines fonctionnels, en facilitant l'ajout ou l'évolution de fonctionnalités.
- **Scalabilité ciblée** : offrir la capacité d'adapter dynamiquement le dimensionnement de chaque service en fonction des charges réelles (messagerie, recherche, notifications…).
- **Résilience** : limiter les impacts d'une défaillance en cloisonnant les responsabilités fonctionnelles et les flux.
- **Observabilité** : fournir une visibilité fine et bout-en-bout sur les flux applicatifs et les performances de la plateforme.
- **Sécurité** : garantir un cloisonnement strict, un chiffrement systématique des flux, et une gestion centralisée des autorisations métier.

Pour répondre à ces exigences, l'architecture technique cible s'appuie notamment sur les apports suivants :

- Un service mesh **Istio** pour le contrôle et la sécurisation des communications inter-services.
- Un service transverse **Permify** pour la gestion fine des autorisations (ABAC/RBAC).
- Un découpage en microservices autonomes, chacun responsable de son propre périmètre fonctionnel.

4.1.3. Découpage en microservices

Le découpage est fondé sur les **quartiers fonctionnels** identifiés dans la vue logique :

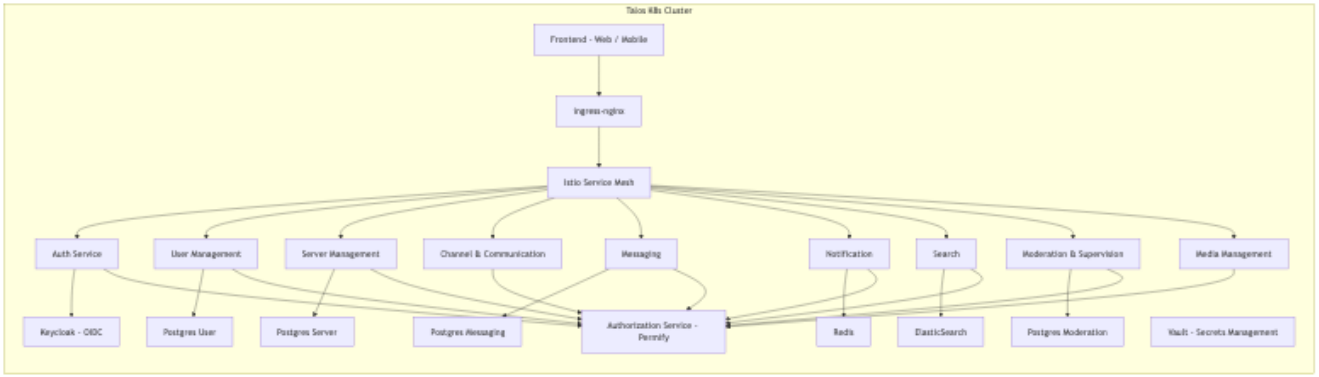


Les principaux services sont :

Service	**Rôle principal**
Auth Service	Authentification OIDC (Keycloak), gestion des tokens
User Management	Gestion des profils utilisateurs, relations
Server Management	Gestion des serveurs communautaires
Channel & Communication	Gestion des canaux, permissions
Messaging	Gestion des messages (temps réel, stockage)
Notification	Notification en temps réel (WebSocket / SSE)
Search	Indexation et recherche full-text
Moderation & Supervision	Modération des contenus, supervision globale
Media Management	Stockage et gestion des fichiers
Authorization (Permify)	Service transverse d’autorisation (ABAC/RBAC)

4.1.4. Architecture de déploiement cible

La plateforme est déployée sur un cluster Kubernetes **Talos**, orchestré sur une infrastructure **Proxmox** baremetal / virtualisée. Le déploiement s’appuie sur une architecture d’observabilité avancée et un maillage de sécurité renforcé.



Les composants transverses clés sont :

- **Istio** : sécurisation mTLS des flux internes, politique réseau fine, observabilité transverse.
- **Permify** : service d'autorisation utilisé par l'ensemble des microservices pour évaluer les droits d'accès.
- **Vault** : gestion sécurisée des secrets, avec rotation automatique.
- **ElasticSearch** : moteur de recherche full-text.
- **Redis** : stockage temporaire (sessions, présence).
- **PostgreSQL** : base de données dédiée par domaine métier.

4.1.5. Modes de communication inter-services

Les interactions entre microservices sont structurées selon le principe suivant :

- **Appels REST synchrones** (via Istio, sécurisé en mTLS) pour les interactions courantes entre services (CRUD, recherche, vérification de droits…).
- **Utilisation transverse de Permify** en REST ou en gRPC (selon besoins de performance) pour les vérifications d'autorisations.
- **WebSocket / SSE** pour les notifications temps réel.
- **Pas de message broker** : les flux asynchrones sont réalisés via Redis Pub/Sub ou mécanismes légers, en conformité avec les contraintes de l'exercice.

Grâce à Istio, chaque appel entre microservices bénéficie d'une sécurisation mTLS automatique, d'une authentification forte, et d'une traçabilité complète (spans propagés, métriques).

4.1.6. Conclusion

L'architecture technique cible de Beep vise à garantir :

- Une séparation claire des responsabilités métier.
- Une sécurité transverse forte (mTLS, contrôle d'accès via Permify, secrets centralisés).
- Une scalabilité fine, avec une capacité d'évolution indépendante des services.
- Une excellente observabilité, grâce à l'intégration native avec le service mesh.
- Une conformité totale avec les contraintes du projet (absence de broker, CQRS, ES).

Les sections suivantes détailleront plus en profondeur la gestion de l'authentification, la supervision, la sécurité, l'intégration UI et les stratégies de gestion de production.

4.2. Gestion de l'authentification et du contrôle d'accès

4.2.1. Objectif

Cette section décrit le système d'authentification et d'autorisation de Beep, dans le contexte de l'architecture microservices.

Elle précise :

- L'intégration d'un serveur OIDC
- Les flux d'authentification
- La gestion des rôles et permissions
- L'utilisation du service transverse **Authorization (Permify)**
- Les implications sur les interactions inter-services

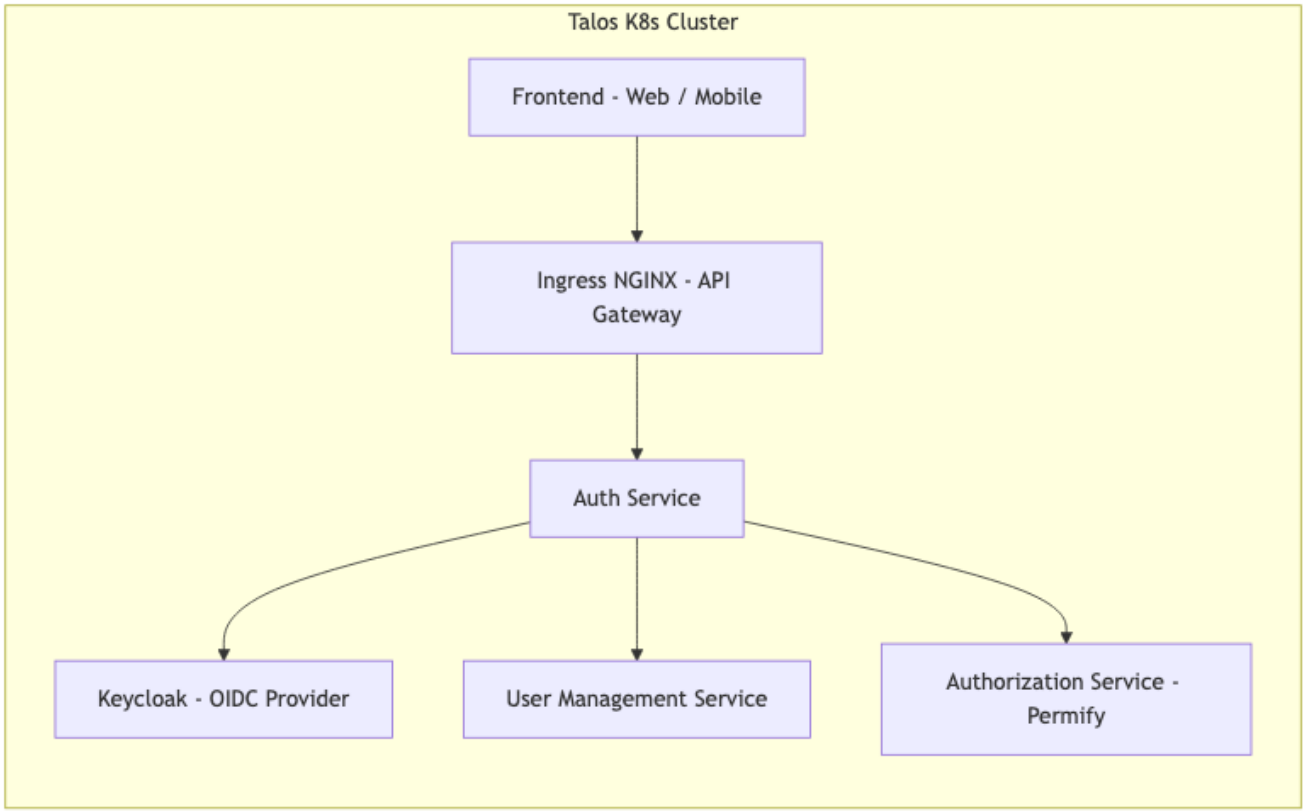
4.2.2. Architecture de l'authentification

Le service d'authentification de Beep repose sur deux briques complémentaires :

- **Auth Service** : brique applicative qui expose les endpoints REST d'authentification de Beep (signup, login, refresh), orchestre les échanges avec Keycloak, et déclenche les flux applicatifs nécessaires (création de profil utilisateur, audit, etc.).
- **Keycloak** : Identity Provider OIDC, utilisé par le Auth Service pour la gestion des comptes et des sessions, et par le Frontend pour les flux OAuth interactifs (Google, Polytech).

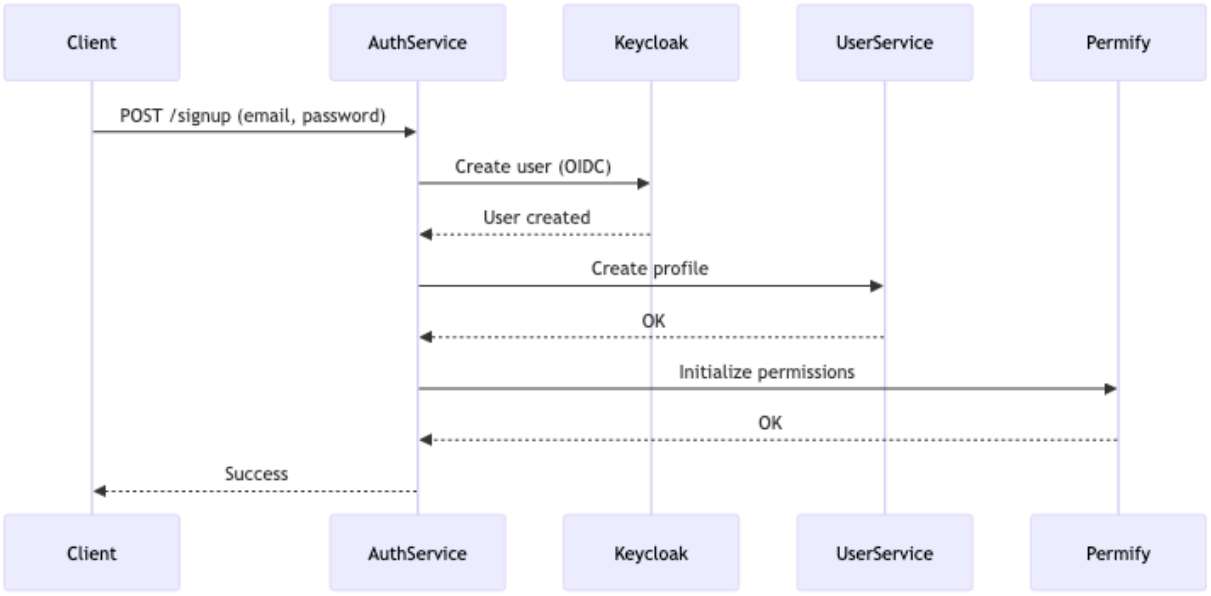
Dans tous les diagrammes, le `Auth Service` est la façade authentification pour le Frontend.

4.2.3. Architecture de déploiement

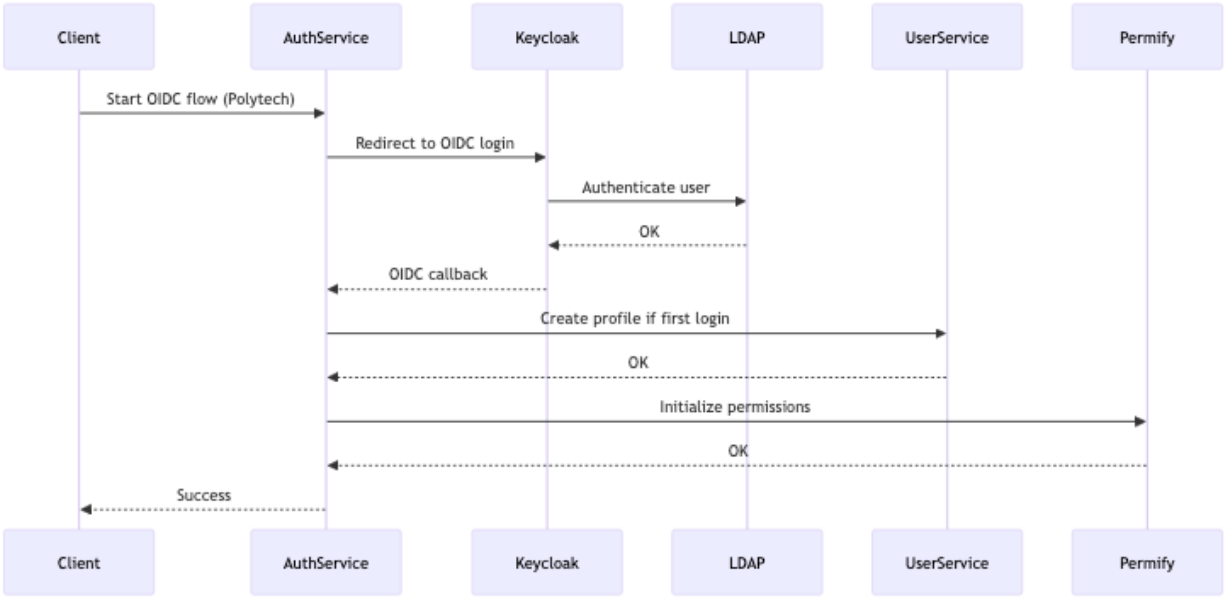


4.2.4. Flux d'authentification principaux

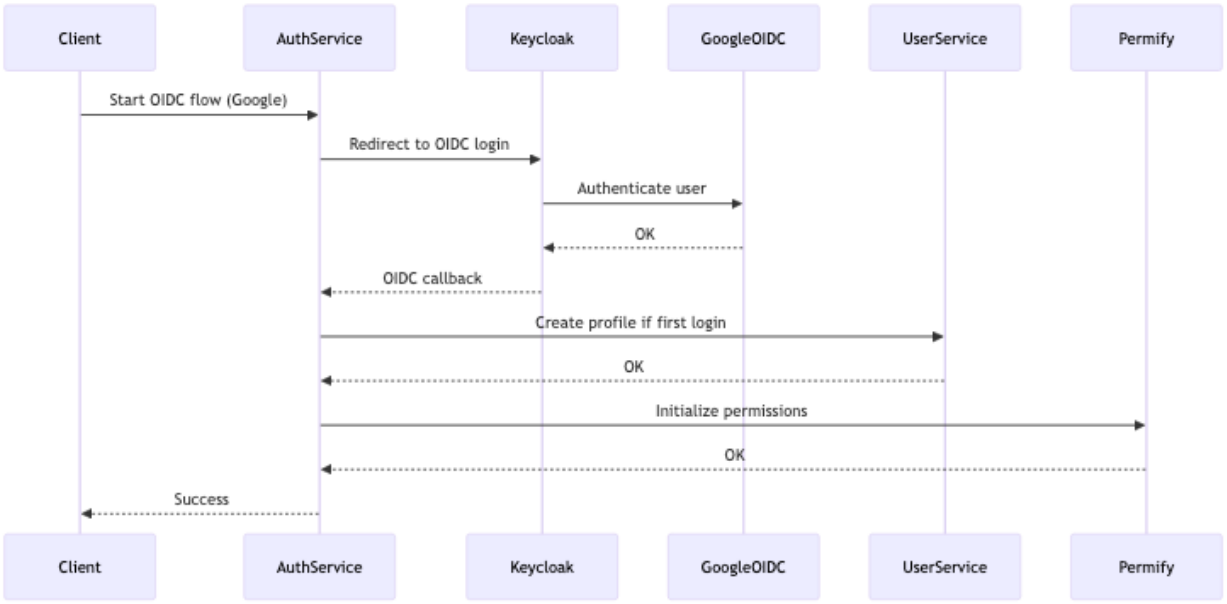
4.2.4.1. Création d'un compte Beep (email / mot de passe)



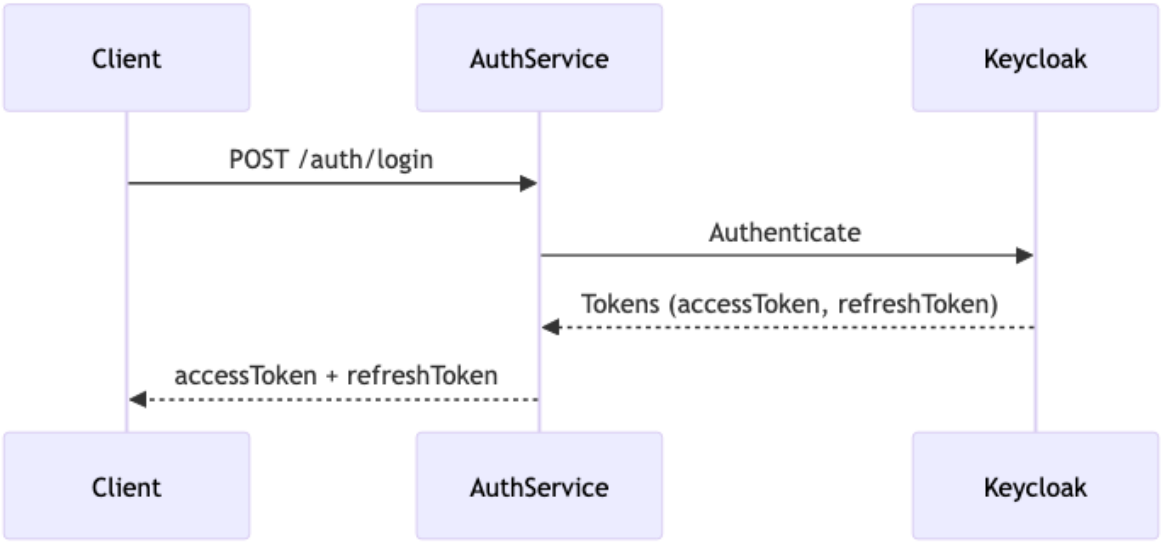
4.2.4.2. Création d'un compte Beep via Polytech (LDAP via OIDC)



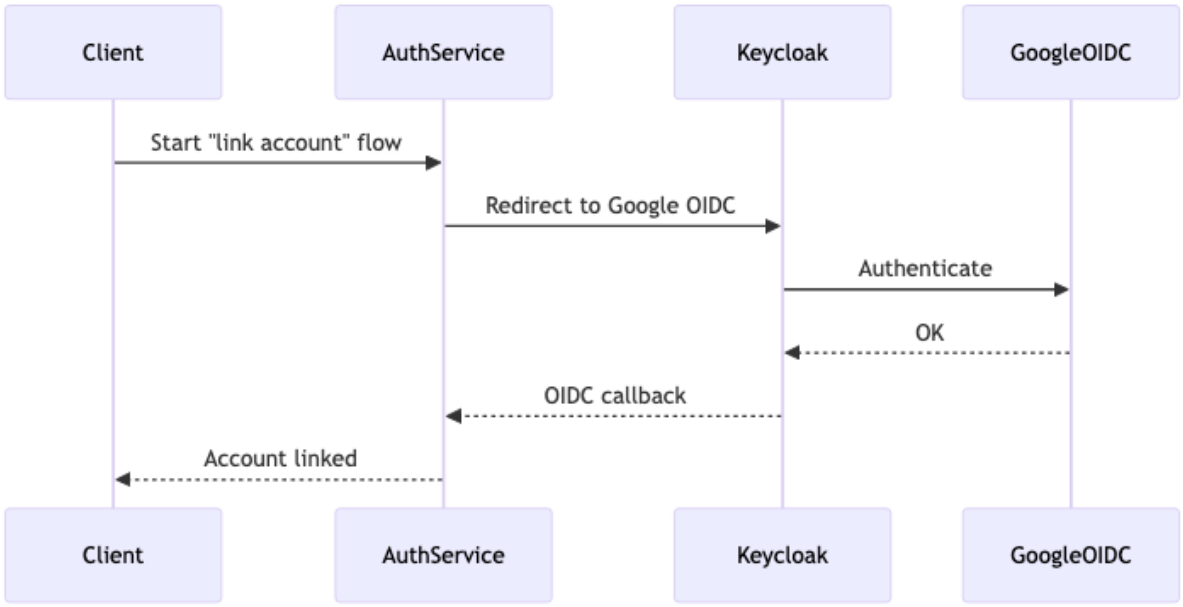
4.2.4.3. Création d'un compte Beep via Google



4.2.4.4. Connexion d’un utilisateur



4.2.4.5. Association d’un compte Google à un compte Beep existant



4.2.5. Gestion des rôles et permissions

La plateforme distingue :

- **Rôles globaux** (gérés dans Keycloak via OIDC claims)
  - `user`
  - `admin`
- **Rôles locaux** (par serveur), gérés au niveau du service `Server Management` et du service `Channel & Communication`
  - `owner`, `default`, + rôles personnalisés
  - Ces rôles sont stockés dans la politique d'autorisation centralisée (via `Authorization Service - Permify`).

4.2.6. Contrôle d'accès

- Tous les appels REST des microservices sont protégés par **JWT (accessToken)**.
- Les services vérifient systématiquement le JWT en amont via l'API Gateway.
- Les autorisations locales (serveur, canal) sont évaluées via des appels REST/gRPC vers le service transverse **Authorization (Permify)**.

Exemple :

- Lorsqu'un utilisateur tente de poster un message dans un canal, le service `Messaging` interroge **Permify** pour valider que l'utilisateur dispose de la permission `channel:write` sur ce canal.

Ce modèle garantit :

- Une cohérence des politiques d'autorisation
- Une capacité d'audit transverse
- Une flexibilité pour faire évoluer les modèles de permission sans modifier chaque microservice

4.2.7. Conclusion

Ce système garantit :

- Une fédération simple des identités (Google, Polytech)
- Une séparation claire entre **authentification** (Keycloak) et **autorisations métier** (via `Authorization Service - Permify`)
- Une extensibilité pour de futurs providers d'identité (ex: SAML entreprise)
- Une architecture d'autorisation évolutive et auditable

4.3. Observabilité et supervision

4.3.1. Objectifs

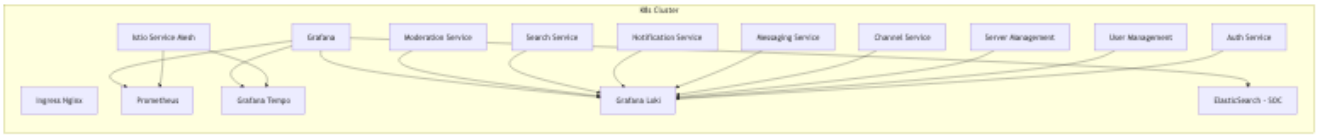
Garantir la capacité à superviser l'ensemble des composants de la plateforme Beep, à diagnostiquer rapidement les anomalies et à comprendre les comportements utilisateurs, est un prérequis essentiel pour assurer une exploitation de qualité et une expérience utilisateur optimale.

L'observabilité du système repose sur plusieurs dimensions complémentaires :

- la journalisation des événements et des flux techniques (logs),
- la traçabilité distribuée des requêtes (traces),
- la collecte et l'analyse de métriques métier et techniques (metrics),
- l'intégration avec les outils de supervision existants (SOC/SIEM).

4.3.2. Architecture d'observabilité

Le dispositif d'observabilité de Beep s'appuie sur une stack open-source cohérente et éprouvée, déployée sur le cluster Kubernetes de production.



Les composants clés sont :

- **Prometheus** pour la collecte de métriques (techniques et métier)
- **Grafana Tempo** pour le tracing distribué (basé sur OpenTelemetry)
- **Grafana Loki** pour la journalisation centralisée des logs
- **Grafana** comme point d'entrée pour l'observabilité unifiée
- **ElasticSearch** pour l'archivage long terme des logs et leur exposition vers le SIEM.

Le maillage réseau assuré par **Istio** (mutual TLS activé) permet une instrumentation automatique et homogène de la majorité des flux inter-services, facilitant la collecte des traces et métriques.

4.3.3. Journalisation centralisée (Logs)

Chaque microservice produit des logs structurés au format JSON, enrichis avec les métadonnées nécessaires :

- identifiants de trace (trace_id, span_id)
- identifiants utilisateur
- identifiants métier (serveur, canal, message, etc.)
- contexte d'exécution (environnement, version de service)

Les logs sont agrégés via **Loki** pour une analyse en temps réel, avec rétention courte (15 jours) pour l'exploitation courante, et archivage plus long (ElasticSearch) pour les besoins de conformité et de forensique.

Les événements de sécurité critiques (authentification, tentatives d'attaque, modification de rôles, suppressions sensibles…) sont systématiquement exportés vers le **SIEM**.

4.3.4. Traces distribuées

Les flux utilisateur impliquent souvent plusieurs microservices en cascade. Afin de comprendre et d'optimiser ces parcours, tous les appels inter-services sont tracés via **OpenTelemetry**, avec Tempo en backend.

Chaque requête reçoit un identifiant de trace global propagé tout au long de son cycle de traitement, permettant :

- d'identifier les goulots d'étranglement,
- de mesurer la latence de bout en bout,
- de diagnostiquer rapidement les anomalies de performance.

Les traces sont visualisables via Grafana, et corrélées avec les logs et métriques pour une analyse complète.

4.3.5. Supervision technique et métier (Metrics)

Chaque service expose des métriques Prometheus en natif (via Istio sidecar ou instrumentation spécifique).

Les métriques collectées couvrent plusieurs aspects :

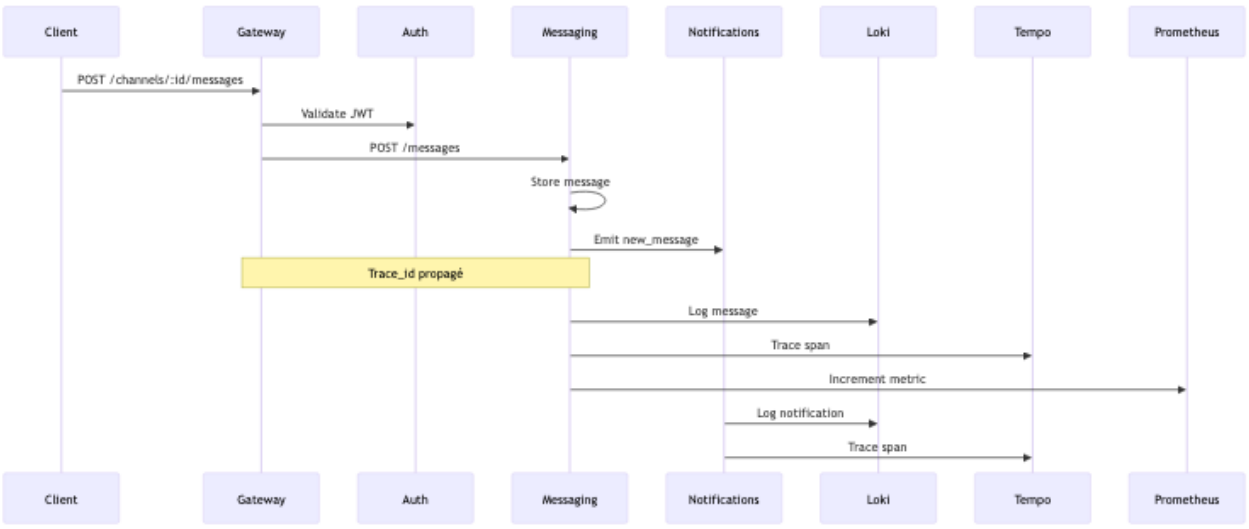
- **Métriques système** : CPU, mémoire, usage réseau
- **Métriques applicatives** : nombre de requêtes, latence par endpoint, taux d'erreurs
- **Métriques métier** : nombre de serveurs créés, nombre de messages envoyés, taux de rétention des utilisateurs, volume de notifications traitées…

Des tableaux de bord Grafana permettent aux équipes :

- de surveiller la santé des services,

- de suivre les indicateurs métier clés,
- d'anticiper les besoins de scaling.

4.3.6. Exemple de flux observable : Envoi de message



4.3.7. Intégration SOC et sécurité des logs

Le dispositif d'observabilité est directement relié au SOC de l'entreprise :

- Les logs de sécurité sont forwardés en temps réel vers **ElasticSearch** / SIEM.
- Des alertes automatiques sont configurées pour les événements sensibles :
- tentatives de connexion suspectes,
- anomalies de comportement,
- patterns de scan ou d'attaque,
- élévation de privilèges anormale.

4.3.8. Conclusion

L'architecture d'observabilité de Beep repose sur des standards ouverts et des solutions éprouvées, garantissant :

- une **visibilité complète** du système, à tous les niveaux,
- une **traçabilité complète** pour les besoins de conformité,
- une capacité à diagnostiquer et résoudre rapidement les incidents,
- un alignement avec les pratiques de supervision de l'entreprise.

Elle constitue un levier clé pour garantir la qualité de service, la sécurité, et l'évolutivité de la plateforme.

4.4. Architecture de sécurité

4.4.1. Objectifs

La sécurité est une composante structurante de l'architecture de Beep.

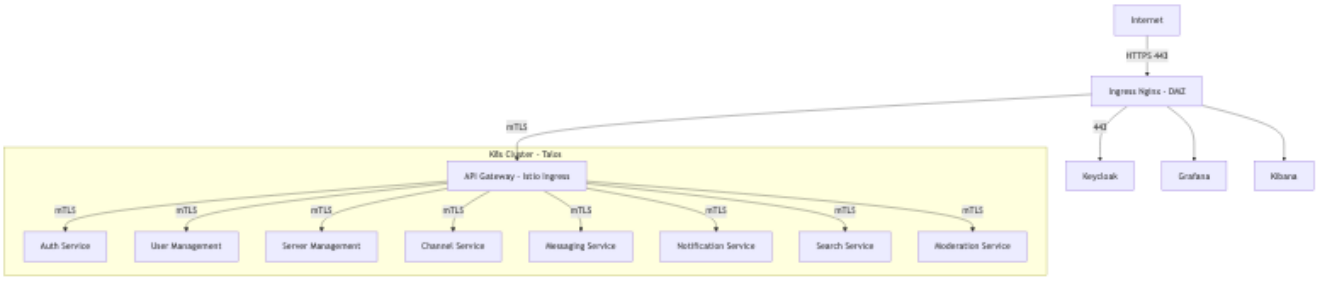
Dans un contexte **multi-tenant**, avec des flux temps réel, une exposition publique et des fonctionnalités de communication interpersonnelle, il est indispensable de garantir :

- la **confidentialité des échanges**,
- l'**intégrité des données**,
- l'**authentification forte** des utilisateurs,
- le **contrôle strict des autorisations**,
- la **résilience face aux menaces**.

Cette section décrit les dispositifs mis en œuvre pour répondre à ces objectifs dans le cadre de l'architecture microservices.

4.4.2. Architecture réseau sécurisée

L'architecture réseau repose sur le principe de **séparation stricte des zones**, avec une **DMZ Kubernetes** en frontal.



Les communications sont sécurisées à plusieurs niveaux :

- **TLS 1.3** en entrée, sur tous les flux externes,
- **mTLS interne** (mutual TLS) pour les communications entre l'API Gateway et les services backend (via le Service Mesh Istio),
- **Network Policies Kubernetes** limitant les communications inter-pods au strict nécessaire.

4.4.3. Authentification et gestion des identités

Le système d'authentification repose sur **Keycloak** en mode OIDC provider.

Les utilisateurs peuvent s'authentifier :

- via un compte **Beep natif** (email / mot de passe),
- via leur compte **Google**,
- via leur compte **Polytech** (OIDC sur LDAP).

Les tokens JWT signés par Keycloak sont utilisés comme **vecteur d'identité** et de preuve d'authentification sur l'ensemble des API.

La validation des tokens est réalisée :

- par l'API Gateway en frontal,
- par les microservices lors des contrôles d'accès locaux.

4.4.4. Gestion fine des autorisations

Le contrôle d'accès métier ne se limite pas aux rôles globaux (`user`, `admin`).

Un service transverse d'autorisations, basé sur **Permify** (moteur d'autorisations relationnelles), est intégré pour gérer :

- les permissions au niveau des **serveurs communautaires**,
- les permissions par **canal**,
- les permissions par **action métier**.

Permify offre un modèle déclaratif flexible et auditable, permettant de :

- décrire les autorisations sous forme de politiques explicites,
- évaluer les droits en temps réel,
- garantir la cohérence des règles même en cas d'évolution du modèle métier.

Exemple : seul un utilisateur disposant de la permission `channel:write` sur un canal donné pourra y poster un message.

4.4.5. Chiffrement des flux et des données

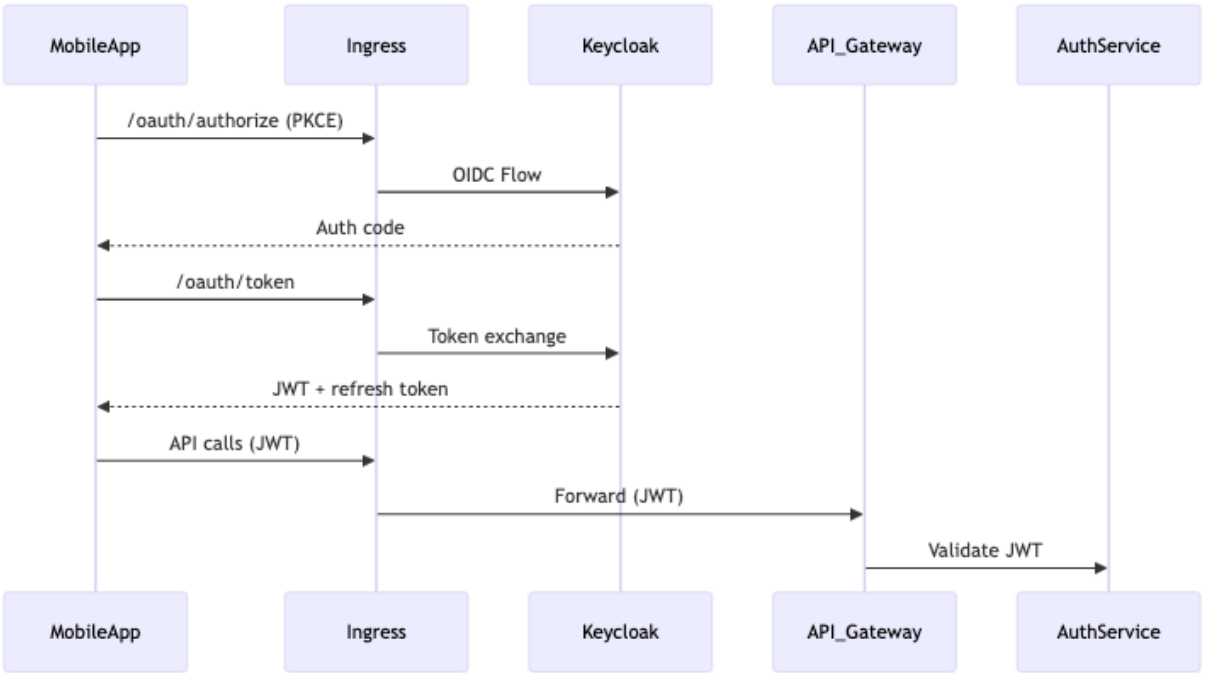
Le chiffrement est systématisé :

- **TLS 1.3** en entrée et en interne (via Istio),
- **JWT signés** (RS256),
- Données au repos :
- **PostgreSQL** : encryption at rest (chiffrement natif PG + stockage chiffré Talos / Ceph),
- **ElasticSearch** : index chiffrés, transport chiffré,
- **Redis** : TLS activé, chiffrement mémoire optionnel.
- Secrets :
- rotation automatique via **Vault** pour les secrets dynamiques,
- **SealedSecrets** pour les secrets statiques en GitOps.

4.4.6. Cas spécifique : application mobile

Le flux d'authentification pour l'application mobile repose sur **OIDC avec PKCE**, garantissant une

protection forte contre les attaques de type **man-in-the-middle** ou interception de token.



4.4.7. Sécurisation des communications inter-services

Les échanges inter-microservices sont strictement contrôlés :

- chaque service dispose de son **identité mTLS** (SPIFFE ID via Istio),
- les politiques de communication sont définies dans Istio (**Authorization Policies**),
- les contrôles d'accès métier (via JWT + Permify) viennent en complément du filtrage réseau.

Ce modèle permet d'appliquer un **principe de moindre privilège**, en réduisant le périmètre d'attaque potentiel.

4.4.8. Monitoring de la sécurité

Le dispositif de supervision (cf. section **Observabilité**) est enrichi d'un monitoring sécurité :

- les logs de sécurité (authentifications, anomalies) sont forwardés vers le **SIEM**,
- des alertes automatiques sont configurées (**Grafana**, **Alertmanager**),
- les certificats mTLS sont surveillés (expiration, renouvellement),
- des tableaux de bord sécurité spécifiques sont maintenus.

4.4.9. Conclusion

L'architecture de sécurité de Beep repose sur des principes robustes et alignés avec les meilleures pratiques actuelles :

- **Zero Trust** au niveau réseau,
- **authentification forte** et centralisée,

- **contrôle d'accès délégué** et auditable,
- **chiffrement systématique**,
- **supervision active**.

Elle offre un niveau de garantie adapté aux besoins de production, tout en constituant un socle évolutif pour les futures exigences de conformité et de responsabilité numérique.

4.5. Moteur de recherche – Intégration fonctionnelle et technique

4.5.1. Objectifs

Dans une messagerie temps réel comme Beep, la capacité de recherche constitue un levier clé de l'expérience utilisateur.

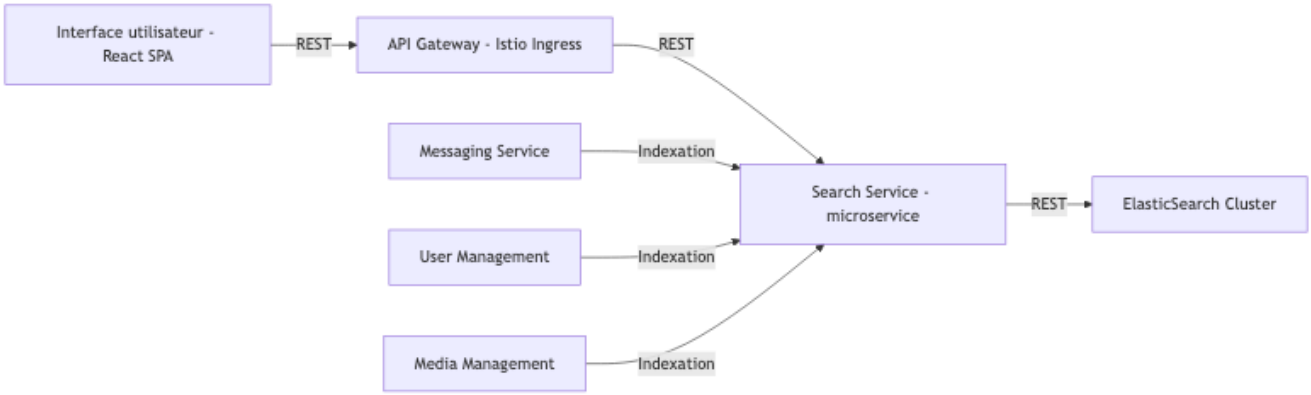
Le moteur de recherche vise à :

- permettre une **recherche textuelle rapide et précise** sur les messages, fichiers, utilisateurs et serveurs,
- fournir un **filtrage contextuel** (par serveur, canal, type de contenu),
- garantir une **latence minimale** même en cas de montée en charge,
- respecter les **permissions d'accès** des utilisateurs.

Cette section présente l'architecture retenue pour cette brique fonctionnelle.

4.5.2. Architecture technique

Le moteur de recherche est conçu comme un **service transverse dédié** au sein de l'architecture microservices.



Le service `Search` encapsule l'accès à ElasticSearch et centralise :

- la logique d'indexation,
- la gestion des droits,
- l'interface de recherche pour les clients frontaux.

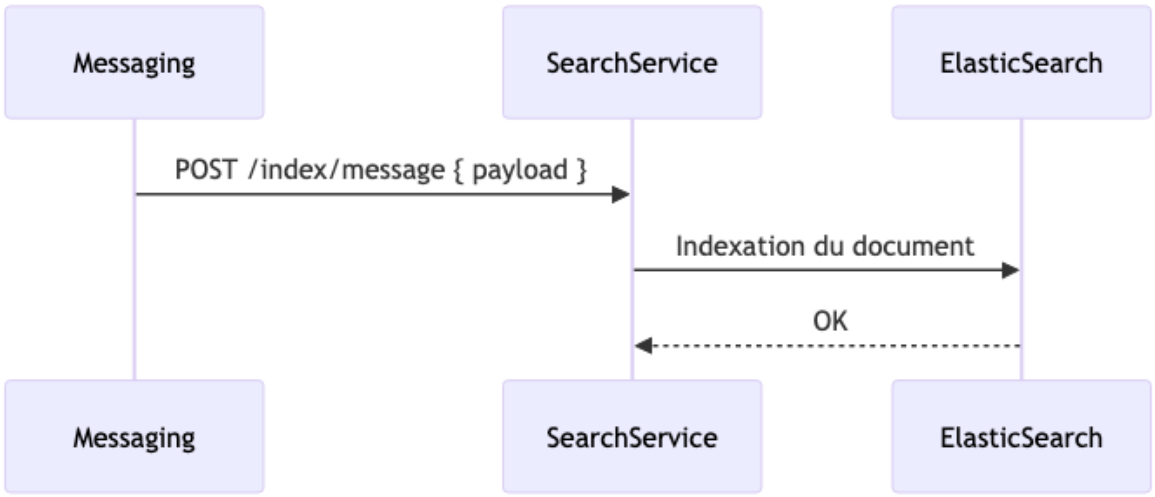
Cela garantit une indépendance vis-à-vis du moteur sous-jacent et facilite son évolution.

4.5.3. Stack technique retenue

Composant	Technologie	Justification
Moteur de recherche	ElasticSearch OSS	Maturité, performance, richesse fonctionnelle
Service API	Search Service (Go / NestJS)	API dédiée, contrôle d'accès centralisé
Frontend	React SPA	Intégration fluide à l'expérience utilisateur Beep
Communication	REST	Simplicité, compatibilité multi-clients

4.5.4. Processus d'indexation

Les services métier (messagerie, utilisateurs, fichiers) publient les événements pertinents vers le service `Search`, via des appels REST ou des hooks.



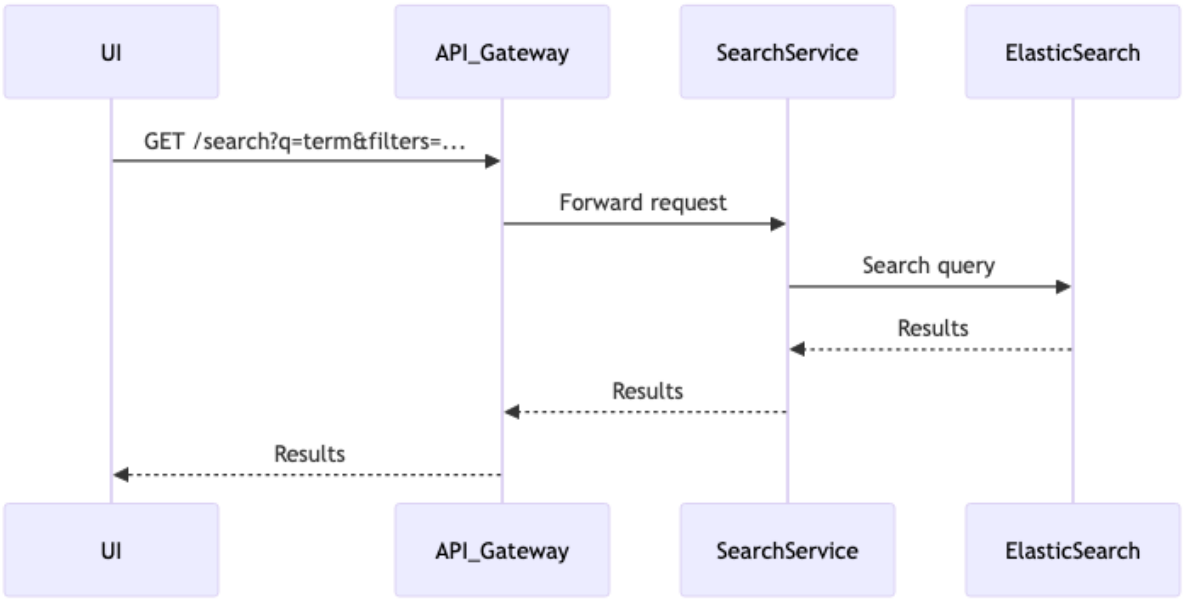
Chaque type d'objet dispose de son propre index :

- **Messages** : contenu, métadonnées (auteur, canal, serveur, timestamp)
- **Fichiers** : nom, type, uploader, contexte de partage
- **Utilisateurs** : username, displayName, email partiel (si autorisé)
- **Serveurs** : nom, description

L'indexation est **quasi temps réel**, garantissant une fraîcheur optimale des résultats.

4.5.5. Processus de recherche

Le client frontal interagit avec le moteur de recherche via l'API Gateway.



Le service `Search` :

- enrichit les requêtes avec le **contexte utilisateur** (extraction des claims du JWT),
- applique des **filtres d'autorisation** dynamiques (scope des serveurs et canaux accessibles),
- effectue si besoin des vérifications d'autorisation spécifiques via des appels à **Permify**,
- formate les résultats pour une présentation unifiée.

4.5.6. UX et intégration UI

La recherche est intégrée en **composant transverse** dans l'interface Beep :

- champ de recherche unifié,
- suggestions temps réel (typeahead),
- affichage des résultats segmenté par type (messages, fichiers, utilisateurs, serveurs),
- filtres avancés (serveur, canal, date, type de contenu).

Cela garantit une UX cohérente et fluide, sans exposition des détails techniques de l'architecture.

4.5.7. Contrôle d'accès

Le moteur de recherche respecte strictement les **permissions d'accès** :

- Seuls les contenus visibles par l'utilisateur connecté sont recherchables.
- Les droits sont validés par le service `Search`, en exploitant :
- les **claims JWT**,
- des requêtes vers **Permify** pour les vérifications complexes ou contextuelles (ex : permissions granulaires par canal).

Ainsi, un utilisateur ne pourra jamais accéder à des messages ou documents en dehors de son périmètre autorisé.

4.5.8. Scalabilité et performances

ElasticSearch permet une montée en charge horizontale :

- sharding des index par type de contenu,
- possibilité de partitionnement par serveur (si besoin).

Le service `Search` peut être scalé indépendamment des autres services en fonction de la volumétrie et du trafic.

Les index sont optimisés pour des **temps de réponse faibles** (< 100 ms sur les requêtes courantes).

4.5.9. Conclusion

L'intégration d'un moteur de recherche transverse dédié renforce considérablement l'expérience utilisateur de Beep :

- navigation fluide dans les contenus,
- recherche contextuelle précise,
- respect strict des permissions,
- montée en charge maîtrisée.

Le découplage en microservice garantit également une **évolutivité** et une **maintenabilité** optimales pour cette brique.

4.6. Intégration des applications UI

4.6.1. Objectifs

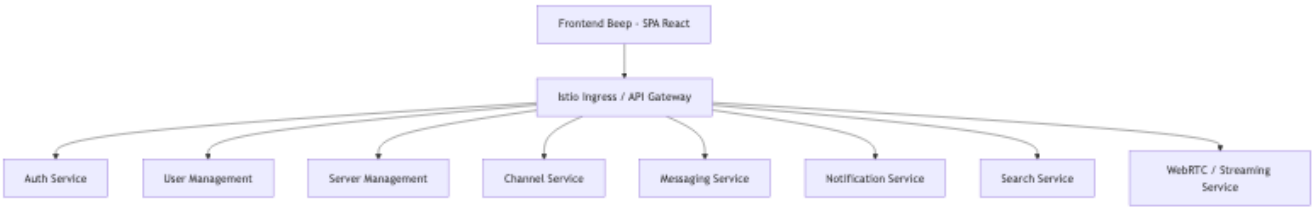
L'architecture microservices de Beep doit rester **totalement transparente pour l'utilisateur final**.

La plateforme doit proposer une **expérience utilisateur fluide, cohérente et unifiée**, quels que soient les services sollicités en back-end.

Cette section décrit :

- les principes d'intégration frontend,
- les patterns d'interaction avec les services,
- le cas spécifique des flux temps réel (WebSocket, WebRTC).

4.6.2. Architecture générale



Le **point d'entrée unique** pour le client est l'API Gateway, protégée par mTLS et JWT.

4.6.3. Approche d'intégration

4.6.3.1. API Gateway unifiée

Toutes les requêtes UI passent par l'API Gateway :

- homogénéisation des points d'entrée,
- gestion des erreurs,
- contrôle d'accès,
- traçabilité (observabilité).

Cela isole le client des détails d'implémentation des microservices.

4.6.3.2. SPA React centralisée

Le client Beep est développé sous forme de **Single Page Application (SPA)** :

- framework : React + Next.js (statique optimisé / hydratation côté client),
- état applicatif : React Query ou Redux Toolkit.

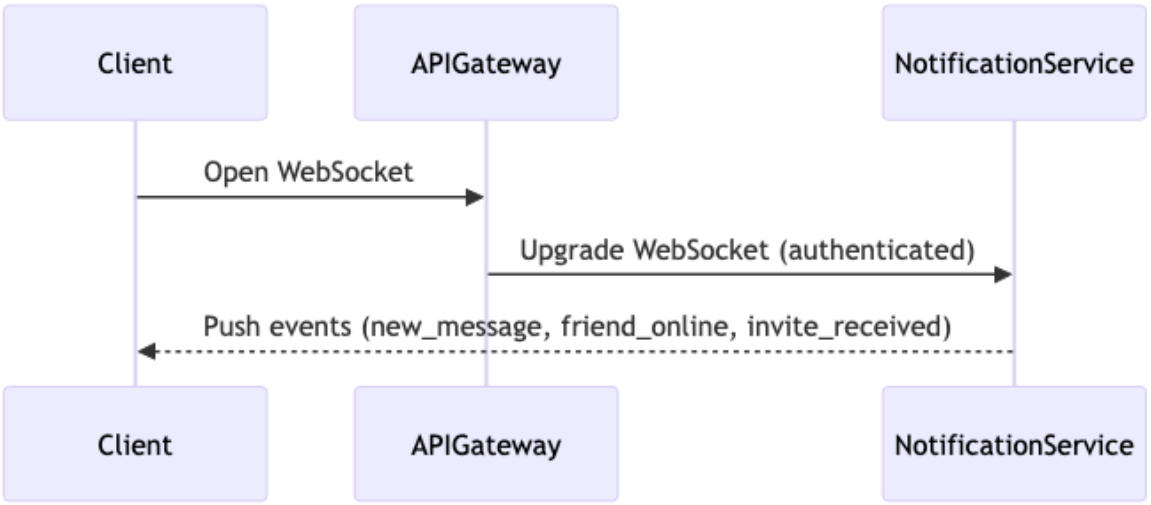
Les interactions se font en :

- REST synchrones (appels métier),
- WebSocket (temps réel : messages, notifications, présence).

4.6.3.3. Gestion des flux temps réel

Le canal principal temps réel est géré via **WebSocket sécurisé** :

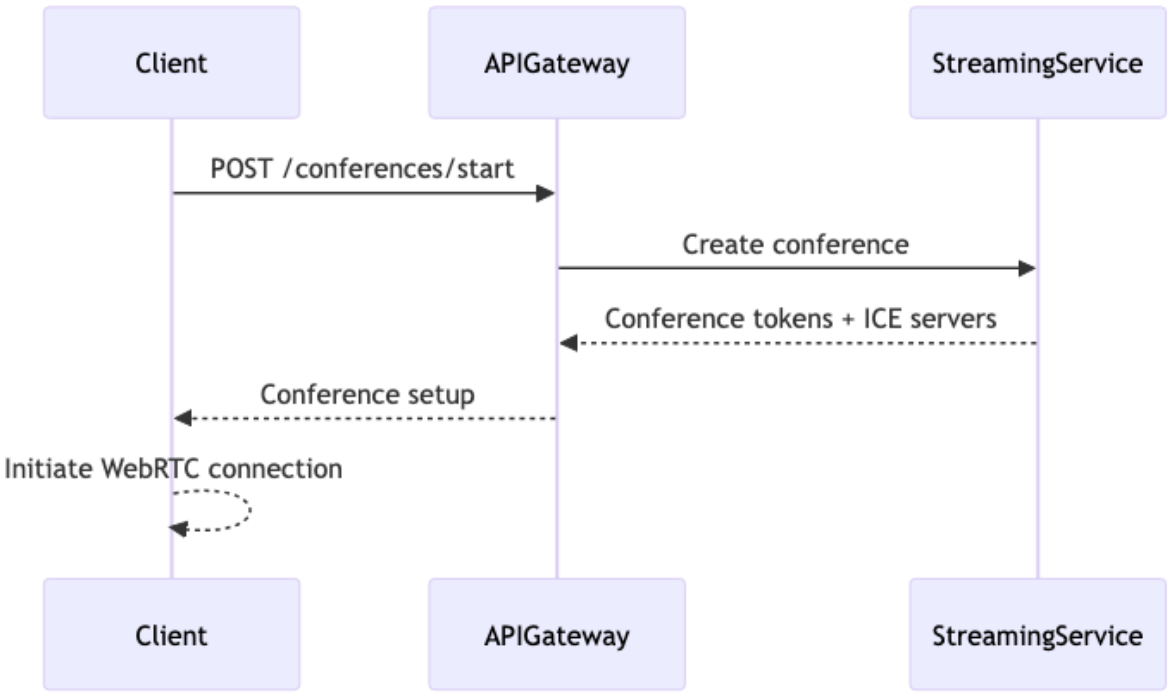
- Notifications : push (nouveaux messages, statut en ligne, invitations),
- Messagerie : nouveaux messages en direct.



4.6.3.4. Cas spécifique : WebRTC (visioconférences)

La brique conférence (vidéo/voix) utilise un pattern hybride :

- contrôle de session : REST,
- signalisation : WebSocket,
- transport média : **WebRTC** (P2P ou via TURN).



Les flux médias ne transitent pas par le cluster applicatif, ce qui garantit **scalabilité et faible latence**.

4.6.4. Cohérence UX

L'application est conçue pour offrir une UX :

- homogène sur l'ensemble des fonctionnalités,
- réactive (optimisation du temps de réponse perçu),
- résiliente face aux indisponibilités partielles (dégradation progressive).

Techniques employées :

- gestion centralisée des erreurs et de la session,
- feedback utilisateur instantané,
- synchronisation en temps réel de l'état applicatif.

4.6.5. Conclusion

Cette approche garantit que **l'architecture distribuée reste invisible pour l'utilisateur**.

Le découplage fort entre frontend et microservices permet de :

- faire évoluer les briques backend en toute transparence,
- optimiser indépendamment les flux UI et les flux métier,
- maintenir une UX optimale, même à grande échelle.

4.7. Gestion de la production et exploitation

4.7.1. Objectifs

Assurer une mise en production :

- fiable,
- reproductible,
- sécurisée,
- monitorée.

Garantir un **pilotage opérationnel efficace** de la plateforme Beep :

- gestion des incidents,
- supervision en continu,
- stratégie de continuité et de réversibilité.

4.7.2. Pipeline CI/CD

Le pipeline CI/CD est conçu autour de :

- **GitLab CI** pour l'automatisation des processus,
- **GitOps** (ArgoCD) pour la cohérence des déploiements.

Principes :

- Build d'images OCI standardisées (Docker/OCI),
- tests automatisés :
- unitaires,
- tests d'intégration,
- scans de sécurité (SAST, Dependency Scanning),
- publication dans registre sécurisé (Harbor ou GitLab Registry),
- promotion d'environnement en environnement (Dev → Staging → Prod).

4.7.3. Stratégie de déploiement

Modes de déploiement :

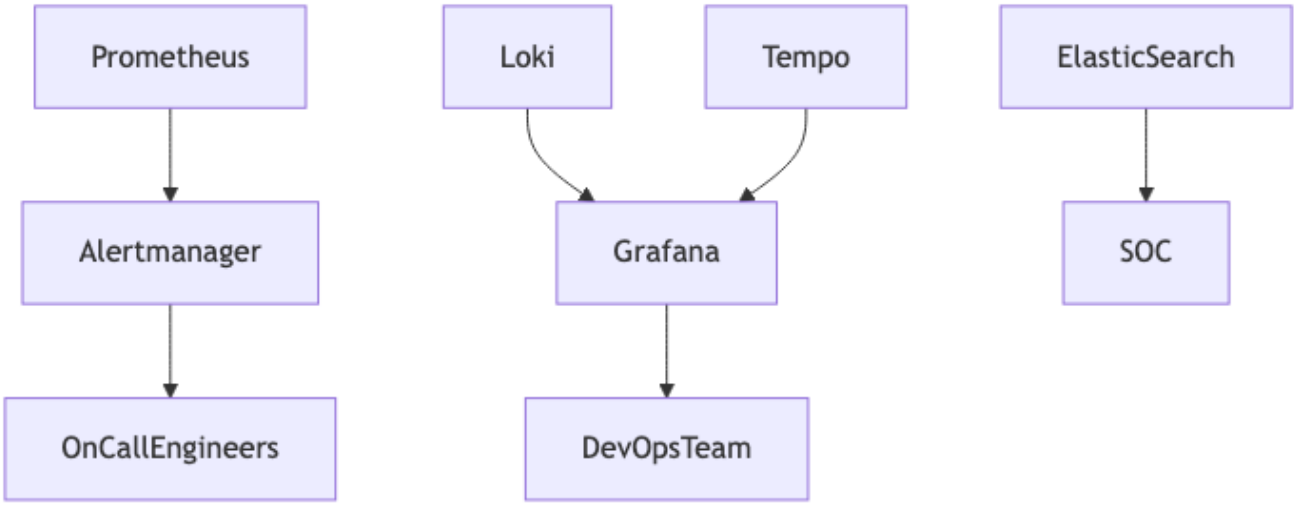
- **Canary Release** : validation progressive sur un sous-ensemble de trafic,
- **Blue/Green Deployment** (optionnel) pour mise à jour sans coupure,
- **Progressive Delivery** avec surveillance automatique des indicateurs clés.

Rollback :

- automatique en cas d'échec détecté,
- manuel possible à toute version validée (stratégie de tags et artefacts versionnés).

4.7.4. Supervision post-production

Architecture de supervision :



Composants :

- **Prometheus** : métriques système + métier,
- **Alertmanager** : gestion des alertes,
- **Grafana** : visualisation (dashboards temps réel),
- **Loki** : centralisation des logs applicatifs,
- **Tempo** : traces distribuées,
- **ElasticSearch** : logs de sécurité, audit,
- intégration avec le **SOC entreprise**.

Tableaux de bord :

- Santé des services,
- Expérience utilisateur (latences perçues),
- Flux critiques (messagerie, recherche, authentification).

4.7.5. Gestion des incidents

Processus :

- Détection proactive via seuils et alerting,
- On-call rotation (astreinte) organisée,
- Playbooks d'intervention documentés,
- Analyse post-incident systématique (post-mortem),
- Publication d'un **rapport de disponibilité mensuel**.

Indicateurs :

- SLO (Service Level Objectives),
- SLI (Service Level Indicators),
- MTTR (Mean Time to Recovery).

4.7.6. Sauvegarde et réversibilité

Politique de sauvegarde :

- Sauvegarde régulière et chiffrée des bases (PostgreSQL, ElasticSearch),
- Rotation et rétention configurée,
- Externalisation vers un stockage sécurisé (type S3 ou équivalent chiffré),
- Tests de restauration périodiques (scénarios réalistes validés).

Réversibilité :

- Stratégie de versioning stricte des artefacts et des manifests GitOps,
- Possibilité de rollback contrôlé jusqu'à **N versions antérieures**,
- Documentation précise des dépendances par version.

4.7.7. Plan d'évolution continue

Démarche DevOps :

- revues régulières de l'architecture et des processus,
- tests de montée en charge planifiés,
- analyse des **coûts d'exploitation**,
- recherche continue de simplification technique,
- intégration des retours opérationnels des équipes et des utilisateurs.

Veille technologique :

- suivi des nouvelles versions des outils clés (Talos, Istio, Keycloak, Elasticsearch, etc.),
- étude des solutions complémentaires (service mesh avancé, outils de chaos engineering),
- intégration progressive de bonnes pratiques du Cloud Native Landscape.

4.7.8. Conclusion

Le dispositif de gestion de production proposé permet à la plateforme Beep de :

- garantir un haut niveau de qualité de service,
- détecter et résoudre rapidement les incidents,
- sécuriser les données,
- assurer une capacité d’évolution contrôlée.

Ce cadre opérationnel soutient la **maturité de la plateforme** en environnement de production exigeant.