Université Mohammed V - Rabat

Projet de Fin d'Année

Système de gestion d'un laboratoire de recherche

Élèves :
El Araby El Mahdi
Ossama El Khalfi

Encadrant :
Khawla Asmi

Jury :
Professeur Soukaina Bouarourou

Année Universitaire 2024-2025

Remerciements

Nous remercions chaleureusement l'équipe pédagogique de l'Université Mohammed V pour son accompagnement durant ce projet. Nos remerciements s'adressent également à notre encadrant pour ses conseils avisés et à nos proches pour leur soutien.

Résumé

Ce rapport présente le développement d'un système centralisé destiné à la gestion d'un laboratoire de recherche, en réponse aux limites des méthodes traditionnelles. L'application permet d'automatiser les processus administratifs, de faciliter la communication entre chercheurs et de renforcer la sécurité des données. Ce travail s'inscrit dans le cadre du projet de fin d'année à l'Université Mohammed V.

Liste des Abréviations

— **API** : Application Programming Interface
— **UI** : User Interface
— **DB** : Database
— **CRUD** : Create, Read, Update, Delete
— **ORM** : Object-Relational Mapping
— **Rust** : Un langage de programmation système
— **Node.js** : Un environnement d'exécution JavaScript côté serveur
— **TypeScript** : Un superset typé de JavaScript
— **WebSockets** : Une technologie de communication en temps réel
— **Next.js** : Un framework React pour la création d'applications web
— **PostgreSQL** : Un système de gestion de base de données relationnelle
— **Prisma** : Un ORM pour les bases de données SQL
— **Docker** : Un outil de conteneurisation
— **Python** : Un langage de programmation
— **FastAPI** : Un framework web modern et rapide pour Python
— **NLP** : Natural Language Processing (Traitement du langage naturel)
— **JWT** : JSON Web Token (jeton web JSON)
— **RBAC** : Role-Based Access Control (contrôle d'accès basé sur les rôles)
— **ACID** : Atomicité, Cohérence, Isolation, Durabilité (propriétés des transactions)
— **SSR** : Server-Side Rendering (rendu côté serveur)
— **SSG** : Static Site Generation (génération de site statique)
— **DBeaver** : Un outil de gestion de base de données
— **Kubernetes** : Un système d'orchestration de conteneurs
— **GitHub** : Un service d'hébergement de répertoires Git en ligne

Introduction générale

La gestion d'un laboratoire de recherche moderne représente un défi multidimensionnel, nécessitant une coordination rigoureuse des activités scientifiques, administratives et collaboratives. Les enjeux majeurs incluent l'optimisation des processus de publication, le suivi des participations aux conférences, la facilitation des échanges et la sécurisation des données de recherche.

Actuellement, de nombreux laboratoires recourent à des méthodes de gestion traditionnelles, reposant sur des outils bureautiques non spécialisés (tableurs, messageries électroniques, systèmes de stockage génériques). Bien que ces solutions soient largement répandues, elles présentent des lacunes significatives : redondance des tâches, dispersion des informations, risques d'erreurs et absence de centralisation. Ces limitations entravent la productivité scientifique et compliquent la prise de décision stratégique.

Dans ce contexte, le développement d'une application web dédiée à la gestion des laboratoires de recherche s'impose comme une solution incontournable. Une telle plateforme permettrait d'automatiser les processus répétitifs, de structurer la collaboration et d'assurer une meilleure traçabilité des données, tout en renforçant la sécurité et l'accessibilité de l'information.

Table des matières

Table des figures

Chapitre 1

Généralités sur le projet

1.1 Problématique

La gestion efficace d'un laboratoire de recherche repose sur la coordination fluide des activités scientifiques, administratives et humaines. Pourtant, de nombreux laboratoires continuent d'utiliser des approches archaïques, souvent basées sur des outils génériques, peu adaptés à la complexité croissante de la recherche contemporaine. Cette situation engendre de nombreuses limites que nous analysons ci-dessous.

1.1.1 Une Gestion Manuelle Fragmentée et Source d'Incohérences

Les laboratoires s'appuient encore trop souvent sur des solutions non spécialisées comme les tableurs ou les échanges de courriels pour suivre les publications, projets, conférences et collaborations.

- **Suivi imprécis et chronophage** : La saisie manuelle des données entraîne des erreurs fréquentes, une mise à jour fastidieuse et une perte de temps considérable, nuisant à la fiabilité du suivi scientifique.
- **Données dispersées et non structurées** : Les informations sont stockées dans des fichiers multiples et non synchronisés, rendant difficile l'obtention d'une vue d'ensemble cohérente et à jour.
- **Vulnérabilité accrue des données** : L'absence de protocoles robustes de sauvegarde et de sécurité expose les fichiers à des risques de perte, de corruption ou d'accès non autorisé.

1.1.2 Communication Désorganisée et Collaboration Sous-Exploitées

La réussite d'un laboratoire dépend de la qualité des échanges entre ses membres. Cependant, les outils de communication utilisés ne sont ni centralisés ni adaptés à un environnement scientifique.

- **Canaux de communication éparpillés** : Entre messageries personnelles et échanges informels, la traçabilité et la conservation des discussions deviennent difficiles.
- **Faible intégration des outils collaboratifs** : L'absence de solutions intégrées et spécialisées freine la coordination et crée des silos informationnels.

1.1.3 Défauts de Sécurité et Faible Capacité d'Évolution

Les systèmes traditionnels, souvent construits de manière ad hoc, présentent des failles critiques.

- **Contrôle d'accès rudimentaire** : La gestion des rôles et des permissions est inexistante ou limitée, ce qui compromet la confidentialité des données sensibles.
- **Infrastructure hétérogène et difficile à maintenir** : La coexistence d'outils isolés rend la maintenance complexe, coûteuse et sujette aux pannes.
- **Absence de scalabilité** : Ces systèmes ne sont pas conçus pour évoluer avec l'augmentation du nombre d'utilisateurs, de projets ou du volume de données, limitant leur durabilité.

1.2 L'objectif à réaliser

1.2.1 Objectif Principal

Développer une application web centralisée, sécurisée, performante et ergonomique pour optimiser la gestion des laboratoires de recherche. Elle repose sur une architecture à microservices, une base de données PostgreSQL, et un outil d'intelligence artificielle en Python pour résumer automatiquement les publications scientifiques.

1.2.2 Fonctionnalités Principales

- **Suivi des publications et conférences** : Vision synthétique et historique complet, avec génération automatique de résumés via l'IA.
- **Messagerie instantanée** : Communication en temps réel entre les membres.
- **Gestion des droits d'accès** : Permissions avancées selon le rôle (administrateur, chercheur, responsable).

1.2.3 Objectifs Secondaires

- Automatiser les tâches pour libérer du temps.
- Centraliser les échanges et favoriser la collaboration.
- Assurer la pérennité, la sécurité et la traçabilité des données.
- Créer une solution évolutive et maintenable sur le long terme.

Chapitre 2

Spécification des besoins

L'architecture backend de cette application repose sur une conception modulaire et distribuée, mettant l'accent sur la séparation claire des responsabilités. Cette méthode vise à maximiser la facilité de maintenance, la possibilité d'évolution indépendante des services, ainsi que la robustesse globale du système. Chaque service, dédié à une fonction métier spécifique, est développé avec la technologie la plus appropriée à ses besoins, tout en s'intégrant harmonieusement dans l'ensemble de l'écosystème applicatif.

2.1 Service de gestion des publications et conférences : `Rust`

Ce service est développé en Rust, en réponse à des exigences strictes de performance, fiabilité et sécurité. Il gère l'ensemble du cycle de vie des publications et conférences à travers des API performantes, incluant un service dédié au téléversement de fichiers ainsi qu'un point de terminaison pour l'observation de métriques internes.

- **Performance et gestion efficace des ressources :** Rust compile en code natif performant, proche du C/C++, sans ramasse-miettes, ce qui est essentiel pour traiter de gros volumes de données bibliographiques et de fichiers.
- **Sécurité mémoire et fiabilité :** Le système d'emprunt de Rust élimine les erreurs mémoire à la compilation, garantissant l'intégrité des données même dans des scénarios à forte charge.
- **Service de téléversement :** Un composant dédié en Rust prend en charge le téléversement sécurisé des fichiers de publication (PDF, documents complémentaires), avec vérification, stockage et association automatique aux métadonnées.
- **Point de terminaison de métriques :** Le service expose un point de terminaison `/metrics` personnalisé, fournissant des statistiques internes telles que le nombre de téléversements, les temps de réponse, les erreurs rencontrées ou encore l'état des composants, facilitant la supervision et le diagnostic.
- **Gestion avancée de la concurrence :** Grâce au modèle de possession de Rust, le traitement simultané des requêtes est effectué de manière sûre, sans accès concurrent dangereux à la mémoire.

Rust s'avère donc particulièrement adapté aux systèmes critiques nécessitant robustesse, performance et observabilité. Il bénéficie également d'un écosystème web moderne

(`Axum`, `SQLx`, etc.) facilitant la construction d'API web fiables et maintenables.

Rust est donc particulièrement adapté aux systèmes critiques. Il bénéficie par ailleurs d'un écosystème web moderne (`Axum`, `SQLx`, etc.) permettant la création d'API web performantes et robustes.

Rust est donc particulièrement adapté aux systèmes critiques. Il bénéficie par ailleurs d'un écosystème web moderne (`Axum`, `SQLx`, etc.) permettant la création d'API web performantes et robustes.

Rust est donc particulièrement adapté aux systèmes critiques et bénéficie d'un écosystème web solide pour des API robustes.

2.2 Service d'authentification et gestion des identités : `Node.js` avec `TypeScript`

Ce service gère l'authentification avec Node.js et TypeScript, pour combiner rapidité de développement et sécurité.

- **Modèle asynchrone et non bloquant :** Idéal pour les vérifications I/O comme la validation de tokens.
- **Robustesse et maintenabilité :** TypeScript améliore la détection d'erreurs et la lisibilité du code.
- **Écosystème sécurisé :** Utilisation de bibliothèques reconnues telles que `Passport.js`, `bcrypt.js`, et `jsonwebtoken`.
- **Limites et solutions :** Pour des charges cryptographiques spécifiques, des modules natifs peuvent être intégrés.

2.3 Serveur de communication en temps réel (chat) : `Node.js` avec WebSockets

Ce service permet des communications bidirectionnelles en temps réel.

- **WebSockets :** Connexions TCP persistantes et full-duplex pour des échanges instantanés.
- **Scalabilité :** L'event loop de Node.js gère efficacement un grand nombre de connexions. Redis Pub/Sub peut assurer la scalabilité horizontale.

2.4 Interface utilisateur (frontend) : `Next.js` avec `TypeScript`

Développé avec Next.js (React) pour des performances optimales et une bonne maintenabilité.

- **Performance et SEO :** Rendu côté serveur (SSR) et génération statique (SSG).
- **Cohérence des types :** Partage de définitions TypeScript entre frontend et backend.
- **UX :** Interface responsive et gestion structurée de l'état applicatif.

2.5 Persistance des données : PostgreSQL

PostgreSQL est choisi pour sa robustesse, ses performances élevées et son respect strict des standards relationnels.

- **Intégrité des données :** Garantit les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité) pour assurer la fiabilité des transactions.
- **Fonctionnalités avancées :** Offre un support natif pour les données semi-structurées via `JSONB`, ainsi que des capacités performantes de recherche en texte intégral.
- **Scalabilité et haute disponibilité :** Intègre des mécanismes de réplication, de partitionnement et de gestion des charges pour répondre aux besoins croissants.
- **Comparaison avec d'autres SGBD :** Préféré à des bases NoSQL comme MongoDB lorsqu'il s'agit de gérer des contraintes relationnelles complexes et des opérations transactionnelles strictes.

2.6 ORM pour base de données : Prisma avec TypeScript

Utilisé dans le backend d'authentification pour faciliter l'accès à la base de données PostgreSQL.

- **Simplicité :** Schéma déclaratif, génération automatique de clients typés.
- **Type-safety :** Autocomplétion, détection d'erreurs.
- **Sécurité :** Prévention des injections SQL via requêtes typées.
- **Transactions :** Support robuste avec gestion concurrente optimisée.

2.7 Conteneurisation et orchestration : Docker

Tous les services, y compris la base de données et les données mock, sont conteneurisés grâce à Docker et orchestrés avec Docker Compose.

- **Isolation complète :** Chaque service s'exécute dans un environnement indépendant, garantissant une isolation des processus et des dépendances.
- **Portabilité et cohérence :** Les conteneurs peuvent être déployés sans modification sur des serveurs locaux, des environnements cloud, ou des clusters.
- **Réseau flexible :** Gestion des adresses IP internes pour la communication inter-services, ainsi que des IP externes pour l'accès public contrôlé.
- **Données mock intégrées :** Les données de test sont préchargées via des conteneurs dédiés pour faciliter le développement et les tests.
- **Préparation à l'orchestration avancée :** La configuration est conçue pour une migration future vers Kubernetes, facilitant l'évolution et la montée en charge.

2.8 Intégration d'une intelligence artificielle : `Python`

Python est utilisé pour le développement des modules d'intelligence artificielle grâce à son écosystème riche et ses bibliothèques avancées de traitement du langage naturel.

- **API web performante :** Utilisation de FastAPI pour exposer des API RESTful et gérer des connexions WebSocket, assurant une communication réactive et scalable.
- **Traitement avancé des documents :** Extraction et analyse de contenu PDF pour alimenter les modèles d'IA.
- **Modèles de NLP :** Implémentation de modèles pré-entraînés de génération et résumé automatique de texte, appuyée par des techniques de tokenisation et prétraitement linguistique.
- **Interopérabilité et maintenabilité :** Architecture modulaire facilitant l'intégration avec les autres composants du système et simplifiant la maintenance.

2.9 Comparaison des Architectures : Monolithique vs Microservices

L'architecture générale de l'application adopte un modèle hybride. Les fonctionnalités principales (gestion des publications, authentification, etc.) reposent sur une architecture microservices, tandis que le serveur de messagerie instantanée est développé selon une structure monolithique. Cette section présente une comparaison critique entre ces deux paradigmes, dans le contexte spécifique de notre système.

2.9.1 Tableau comparatif des caractéristiques

Critère	**Microservices (Cœur de l'application)**	**Monolithique (Serveur de chat)**
Organisation du code	Services indépendants (Rust, Node.js, Python)	Code unifié dans un seul bloc (Node.js + WebSockets)
Scalabilité	Évolutivité horizontale par service	Évolutivité verticale via une instance unique
Déploiement	Déploiement autonome de chaque service (Docker)	Déploiement unique en un seul paquet
Complexité	Complexité accrue (réseau, orchestration)	Complexité initiale réduite
Performance	Optimisation spécifique à chaque service	Faible latence pour les traitements en temps réel
Cohérence des données	Cohérence éventuelle (transactions distribuées complexes)	Cohérence forte (base de données unique)
Liberté technologique	Technologies variées selon le besoin	Restreinte à l'écosystème Node.js
Isolation des pannes	Défaillances circonscrites à un seul service	Risque de panne globale en cas de crash
Vitesse de développement	Autonomie par équipe/service	Rapidité dans un environnement centralisé

2.9.2 Choix des microservices pour le cœur applicatif

1. **Adaptation technologique par domaine** : Chaque composant est développé avec le langage ou l'environnement le plus adapté à ses exigences spécifiques.
   Exemple : Rust pour les opérations performantes sur fichiers ; Node.js pour la gestion asynchrone des sessions.
2. **Scalabilité ciblée** : Les composants fortement sollicités, tels que l'API Gateway, peuvent être mis à l'échelle indépendamment, sans impacter les autres services.
3. **Facilité de maintenance** : Les séparations logiques entre les services (par domaine fonctionnel) facilitent la gestion, le débogage et les évolutions futures.
   Exemple : Aucun modèle de données n'est partagé entre les modules d'authentification et de publication.
4. **Résilience accrue** : En cas de panne dans un service (ex. : service des publications), les autres continuent de fonctionner normalement (ex. : authentification).

2.9.3 Justification du choix monolithique pour le serveur de chat

1. **Contraintes de temps réel** : WebSocket repose sur une communication bidirectionnelle à faible latence, difficile à reproduire entre services séparés par un réseau.
2. **Gestion d'état centralisée** : Le suivi des connexions actives, des messages et des statuts de présence est simplifié dans une architecture monolithique.
3. **Rapidité de développement** : Les fonctionnalités interdépendantes (ex. : statut + messagerie) sont plus rapides à développer dans un code unique et intégré.
4. **Optimisation des performances** : L'absence de latence réseau entre composants internes améliore la réactivité des échanges instantanés.

2.9.4 Enjeux identifiés et stratégies d'atténuation

Problème identifié	**Solution côté microservices**	**Solution côté monolithique**
Communication interservices	API Gateway, Pub/Sub Redis, protocoles REST	Sans objet (communication en mémoire)
Cohérence des données	Patterns Saga, Event Sourcing	Transactions ACID classiques
Complexité d'exploitation	Outils d'orchestration (Docker Compose, Swarm), documentation unifiée	Déploiement unique, scripts simples
Limite de scalabilité	Externalisation possible de l'état (Redis)	Non applicable

2.9.5 Axes d'évolution prévus

— **Scalabilité du serveur de chat** : Migration vers une architecture hybride, avec externalisation de l'état (messages, connexions) via Redis pour permettre la répartition de charge.
— **Renforcement de l'infrastructure microservices** : Intégration d'un API Gateway (Kong, Traefik) et d'un service mesh (Linkerd) pour améliorer la communication interservice et l'observabilité.
— **Supervision centralisée** : Mise en place d'une solution de monitoring comme Prometheus et Grafana afin de visualiser en temps réel les performances de l'ensemble du système.

Conclusion

Nous avons opté pour une architecture modulaire fondée sur des microservices spécialisés, afin de répondre à des exigences élevées en termes de performance, de fiabilité et d'évolutivité. Chaque composant étant isolé, cela facilite la maintenance, les mises à jour, et renforce la résilience du système. Ce choix permet également une intégration continue de nouvelles technologies, garantissant ainsi la pérennité de la plateforme.

Plus précisément, nous avons adopté une approche hybride qui combine les forces de deux modèles : les microservices, pour leur modularité, leur indépendance technologique et leur capacité à monter en charge ; et le monolithe, pour la gestion de la messagerie instantanée, un domaine où la performance et la cohérence de l'état sont essentielles. Cette combinaison nous permet de tirer le meilleur de chaque approche, tout en assurant un équilibre entre flexibilité et efficacité.

Chapitre 3

Conception du site web

3.1 Introduction

Ce chapitre décrit la conception architecturale de l'écosystème de notre application. Le système repose sur trois services principaux, chacun répondant à des besoins spécifiques :

1. **Un système de gestion des publications académiques** : pour centraliser et organiser les travaux de recherche.
2. **Un service d'authentification et de profils utilisateurs** : afin de garantir un accès sécurisé et personnalisé.
3. **Une plateforme de messagerie instantanée** : permettant aux chercheurs de communiquer en direct.

3.2 Vue d'ensemble de l'architecture du système

3.2.1 Architecture multi-services

L'application adopte une approche distribuée, où chaque service fonctionne de manière autonome tout en interagissant avec les autres. Cette modularité permet d'optimiser les performances, la maintenabilité et l'évolutivité du système. Les trois services clés s'appuient sur des modèles architecturaux distincts, choisis pour leur adéquation avec leurs cas d'usage respectifs.

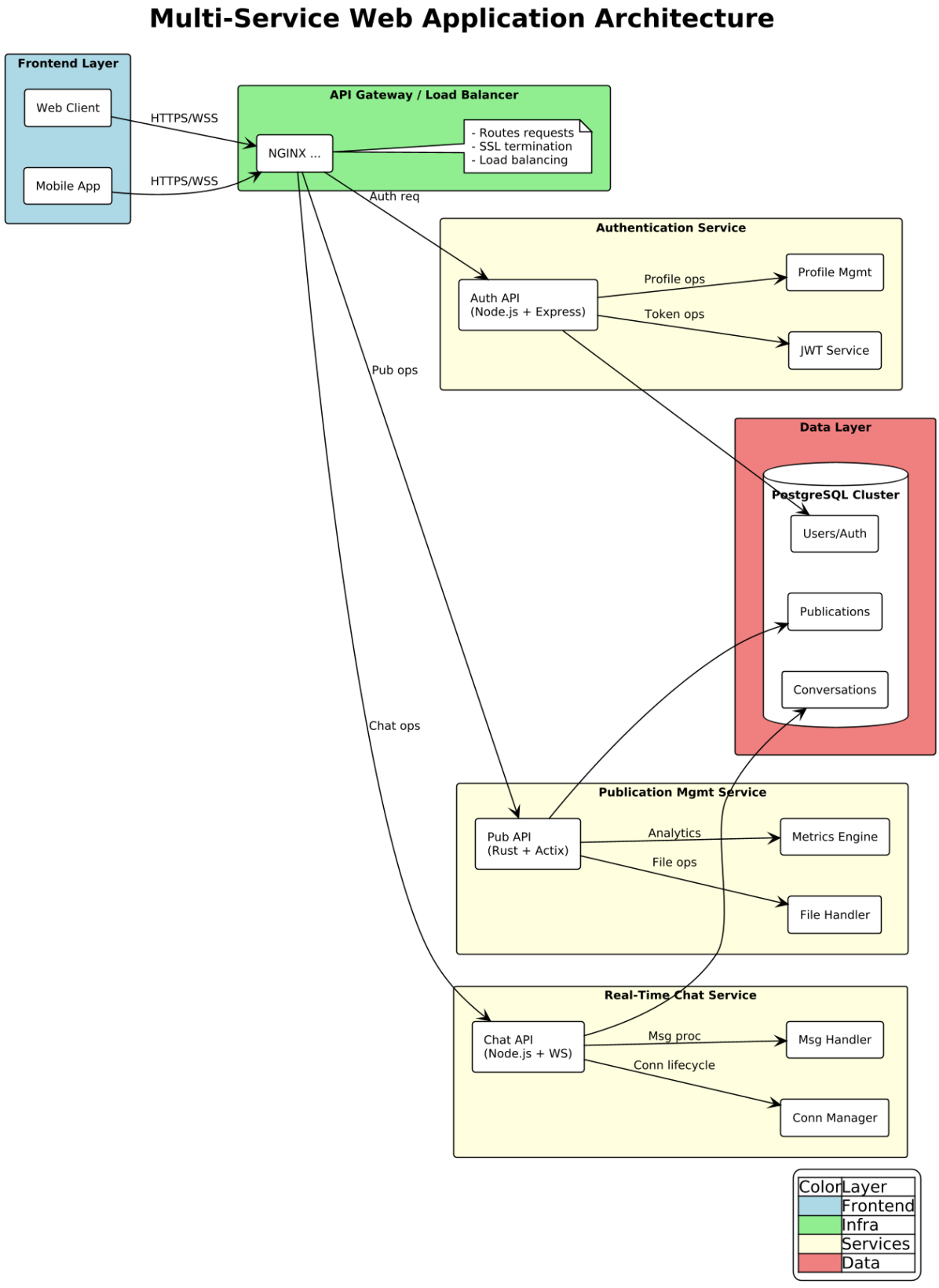

FIGURE 3.1 – Diagramme général du système

3.2.2 Modèle d'interaction de service

Ce schéma détaille les séquences d'interaction entre les services. Par exemple :

1. Le client se connecte via `/auth/login` (HTTP).
2. Le service d'authentification renvoie un JWT.
3. Le client utilise ce token pour accéder aux publications (GET `/publications`).
4. Le chat établit une connexion WebSocket persistante.

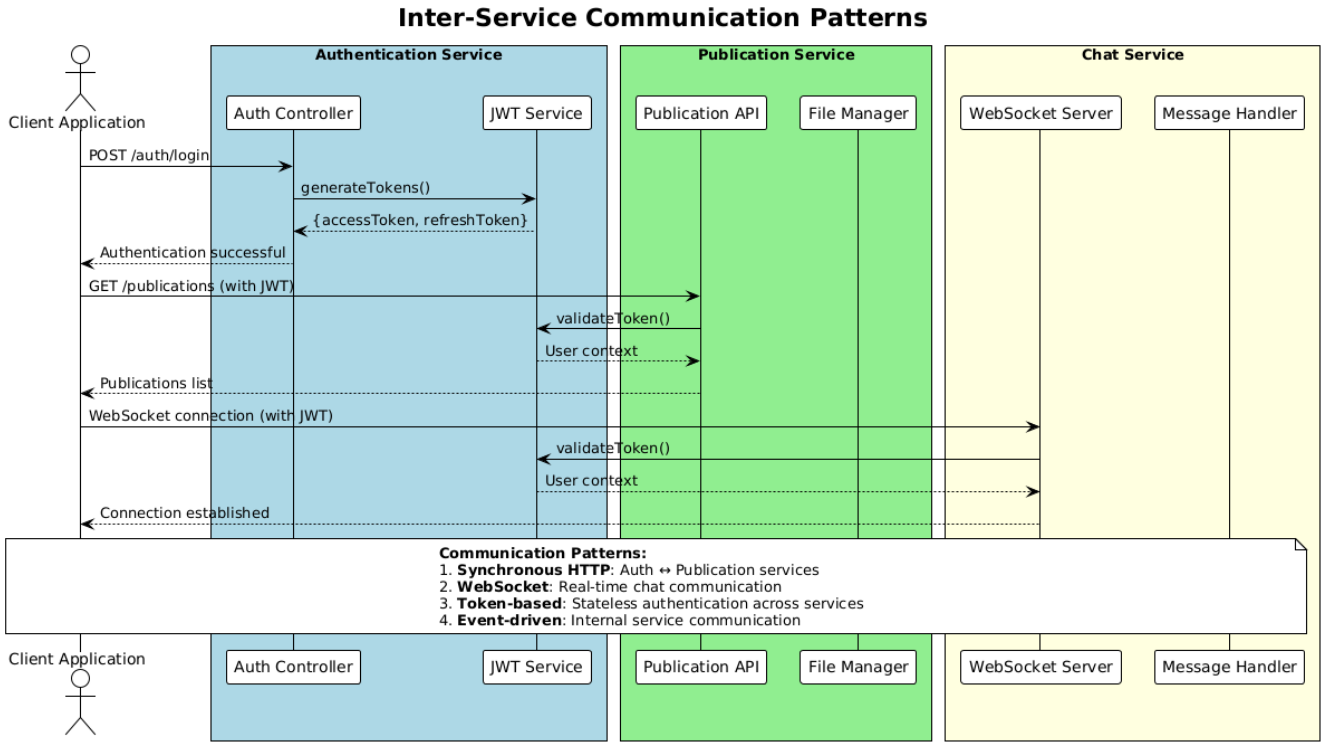

FIGURE 3.2 – Modèle d'Interaction de Service

3.3 Modèles architecturaux spécifiques aux services

3.3.1 Service d'authentification : architecture en couches

L'architecture en couches sépare clairement les responsabilités :
— La couche présentation valide les requêtes.
— La couche métier implémente la logique (ex. hachage de mots de passe).
— La couche données gère les opérations CRUD.

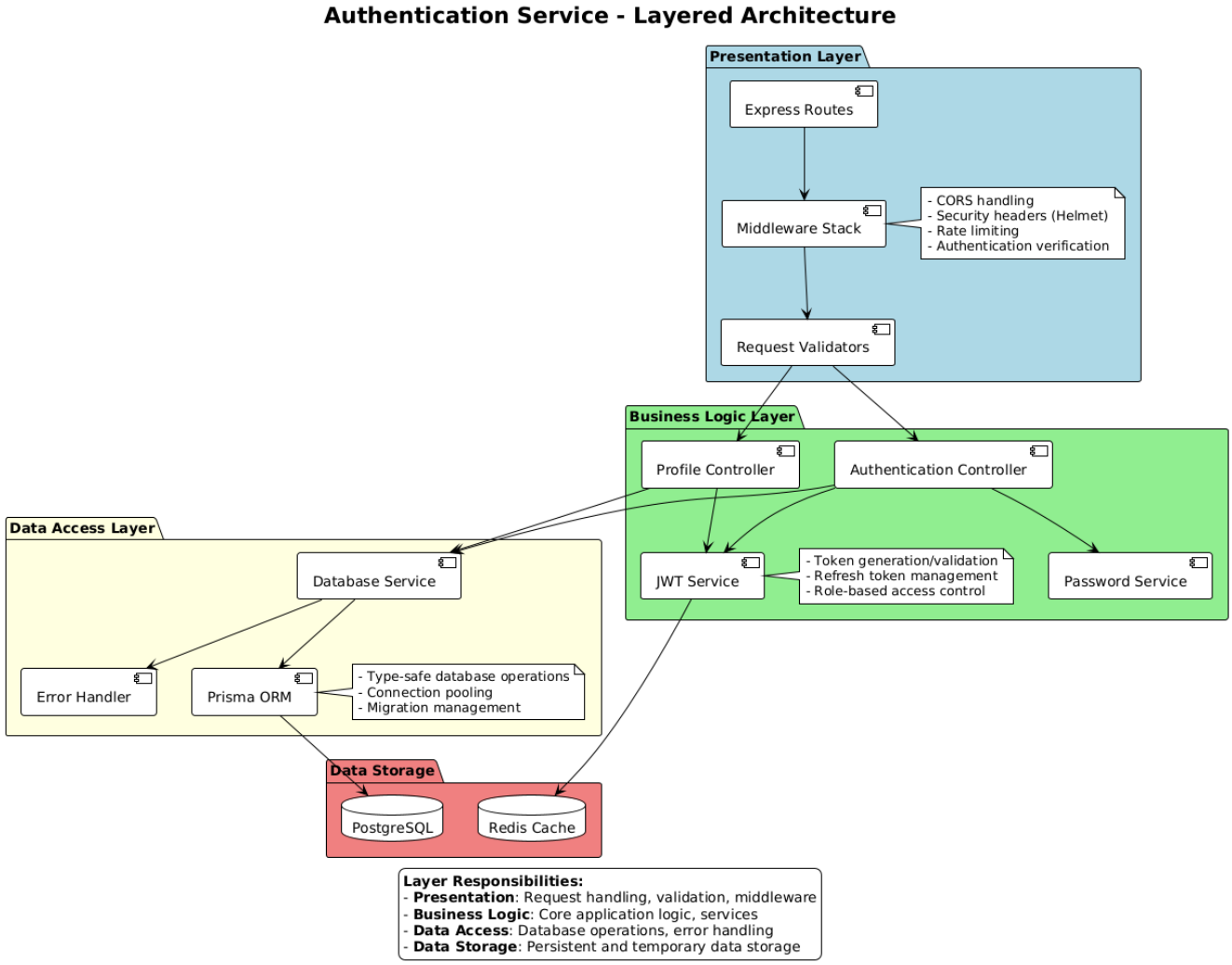

FIGURE 3.3 – Service d'Authentification

3.3.2 Service de publication : conception pilotée par le domaine

Ce diagramme applique les principes du Domain-Driven Design (DDD) :
— Les agrégats (ex. Publication) délimitent les frontières de cohérence.
— Les entités (ex. Conference) ont une identité unique.
— Les services (ex. FileService) gèrent les opérations transverses.

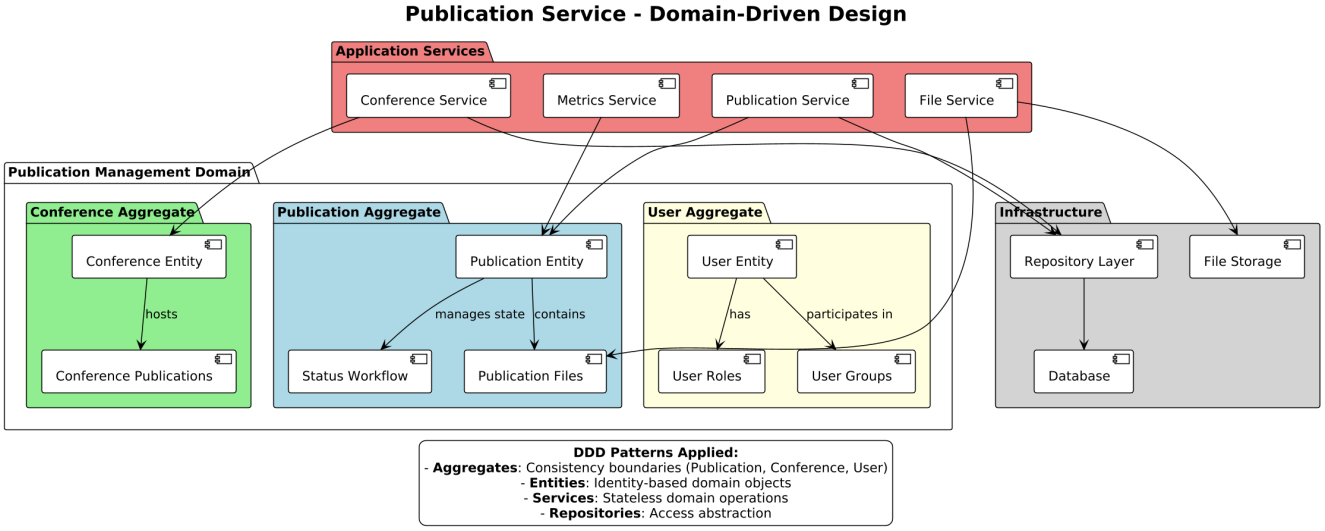

FIGURE 3.4 – Service de Publications

3.3.3 Service de chat : architecture orientée événements

Le diagramme du service de chat illustre une architecture orientée événements conçue pour gérer les interactions entre utilisateurs dans un environnement de chat en temps réel. Cette architecture est pensée pour être évolutive, modulaire, et capable de supporter un grand nombre de connexions simultanées. Elle s'appuie sur l'utilisation de WebSockets pour assurer une communication bidirectionnelle et instantanée ainsi que sur un système d'événements pour coordonner les différents composants du service.

Composants principaux

— **Utilisateurs (User A, User B)**

— **WebSockets (WebSocket A, WebSocket B)**
Interfaces de communication en temps réel entre les utilisateurs et le serveur.
— **Gestionnaire de connexions (Connection Manager)**
Composant chargé de gérer les connexions WebSocket, en enregistrant les connexions et déconnexions des utilisateurs, et en diffusant les mises à jour concernant la présence des utilisateurs.
— **Gestionnaire de messages (Message Handler)**
Responsable de la réception, du traitement et de la diffusion des messages entre les participants au chat.

— **Bus d'événements (Event Bus)**
Système de messagerie interne facilitant la communication entre les différents composants du service en transmettant les événements générés.
— **Base de données (Database)**
Stockage persistant des messages et des informations de connexion, permettant la récupération, l'archivage et la gestion durable des données.

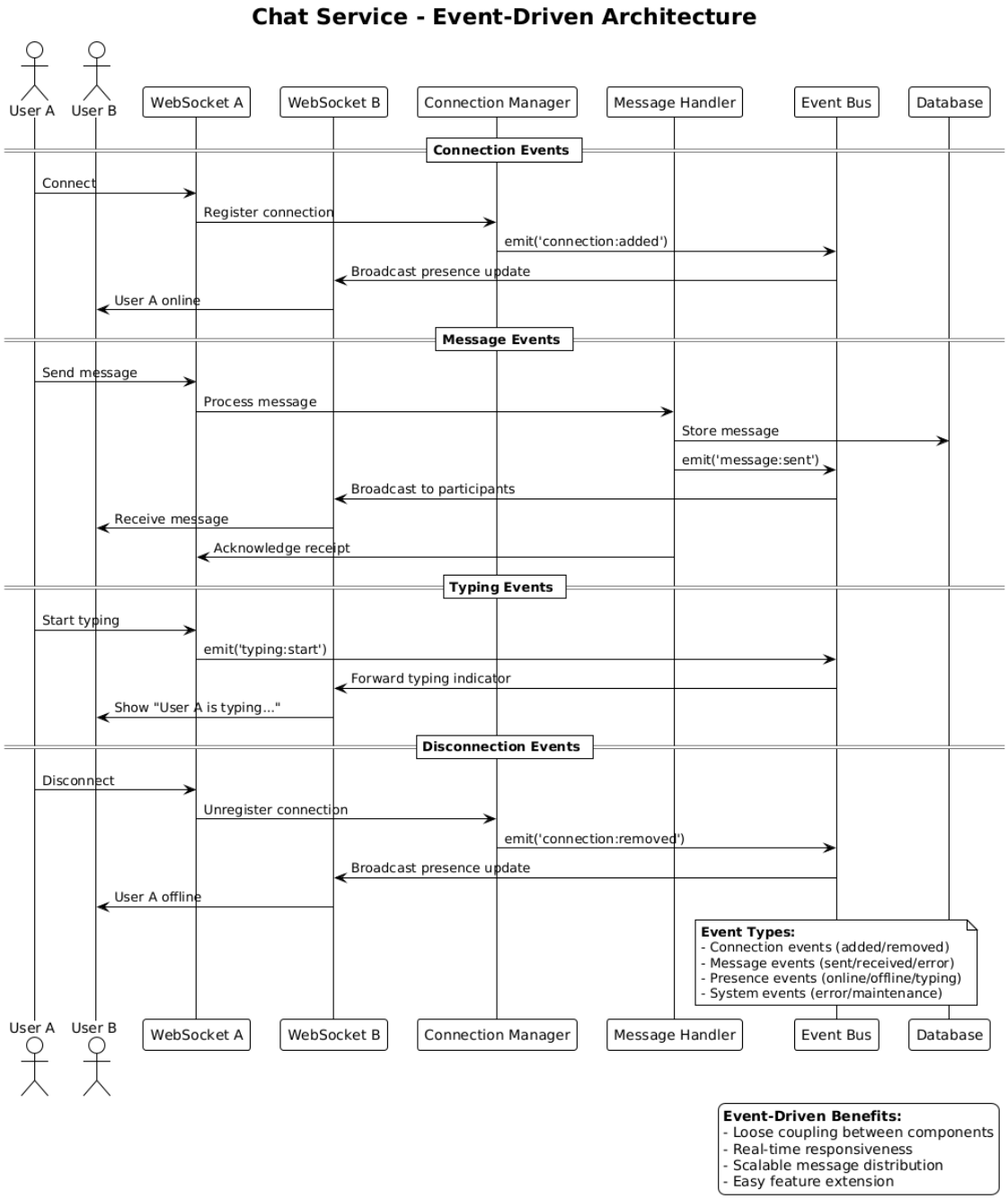

FIGURE 3.5 – Service de Chat

3.4 Architecture des données

3.4.1 Schéma de base de données unifié

Ce schéma UML montre les tables PostgreSQL et leurs relations :
— `users` → `publications` (relation un-à-plusieurs).
— `conferences` → `participants` (relation plusieurs-à-plusieurs via une table de jointure).
— La table `jwt_tokens` gère les sessions actives.

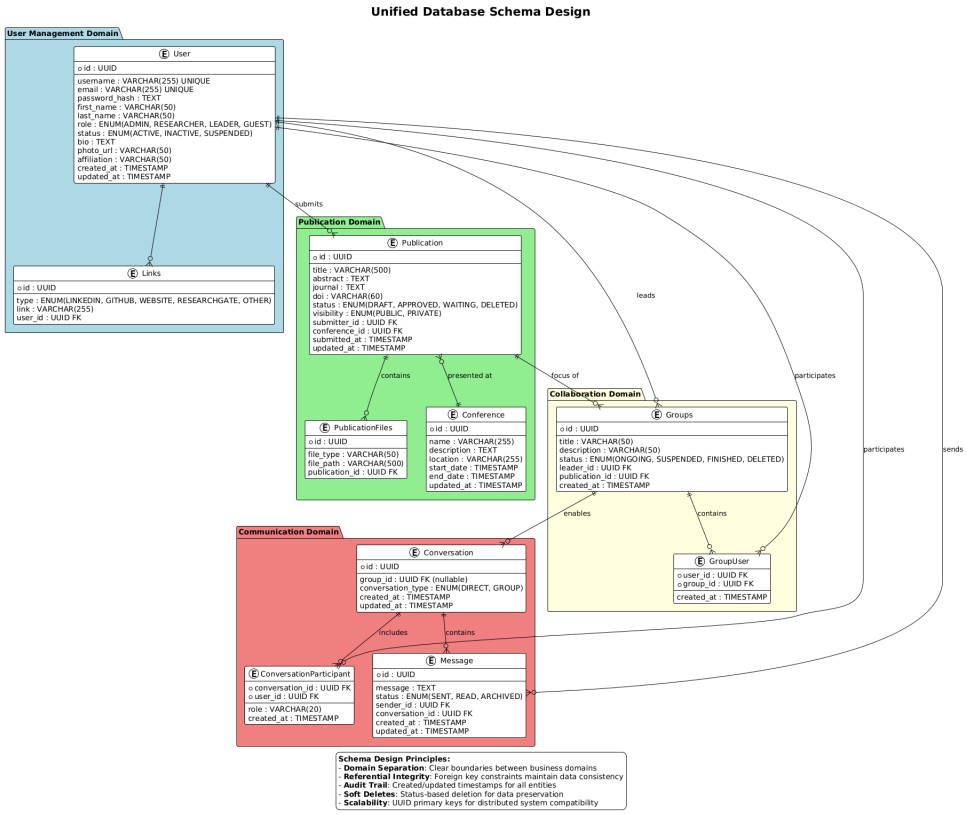

FIGURE 3.6 – Schéma de Base de Données

3.5 Architecture de sécurité

3.5.1 Modèle de sécurité multi-couches

Sécurité Frontend

- **Application de HTTPS** : Assure que les communications entre le client et le serveur se font via HTTPS, chiffrant les données en transit et protégeant contre l'interception.
- **Politique de sécurité des contenus (Content Security Policy)** : Stratégie réduisant les vulnérabilités aux attaques par injection de scripts en spécifiant les sources valides pour le chargement des ressources.
- **Protection contre les attaques XSS** : Prévient les attaques Cross-Site Scripting en neutralisant les scripts malveillants exécutés dans le contexte d'un autre site.
- **Jetons CSRF** : Utilisés pour prévenir les attaques Cross-Site Request Forgery en s'assurant que les requêtes proviennent de la source attendue.

Sécurité Réseau

- **Équilibreur de charge (Load Balancer)** : Répartit le trafic entre plusieurs serveurs pour améliorer disponibilité et résilience.
- **Protection contre les attaques DDoS** : Détecte et bloque le trafic malveillant visant à saturer le système.
- **Limitation du débit (Rate Limiting)** : Restreint le nombre de requêtes par utilisateur pour prévenir abus et attaques.
- **Liste blanche d'IP (IP Whitelisting)** : Autorise uniquement les connexions depuis des adresses IP spécifiques.

Sécurité d'Authentification

- **Validation des jetons JWT** : Vérification de la validité des JSON Web Tokens.
- **Rotation des jetons de rafraîchissement** : Renouvellement sécurisé des jetons d'authentification.

Sécurité d'Autorisation

- **Contrôle d'accès basé sur les rôles (RBAC)** : Attribution des permissions selon les rôles utilisateurs.
- **Autorisations des ressources** : Contrôle précis des actions sur les ressources.
- **Protection des API** : Mesures pour sécuriser les API contre accès non autorisés.
- **Filtrage des données** : Prévention des injections SQL et autres attaques liées aux données.

Sécurité des Données

- **Hachage des mots de passe** : Stockage sécurisé des mots de passe sous forme de hash.

— **Validation des fichiers** : Vérification des fichiers uploadés pour éliminer les codes malveillants.
— **Nettoyage des entrées (Input Sanitization)** : Protection contre XSS, injections SQL, et autres vulnérabilités.

Ces éléments travaillent ensemble pour créer une architecture de sécurité robuste, couvrant les différents aspects de la protection des données et systèmes.

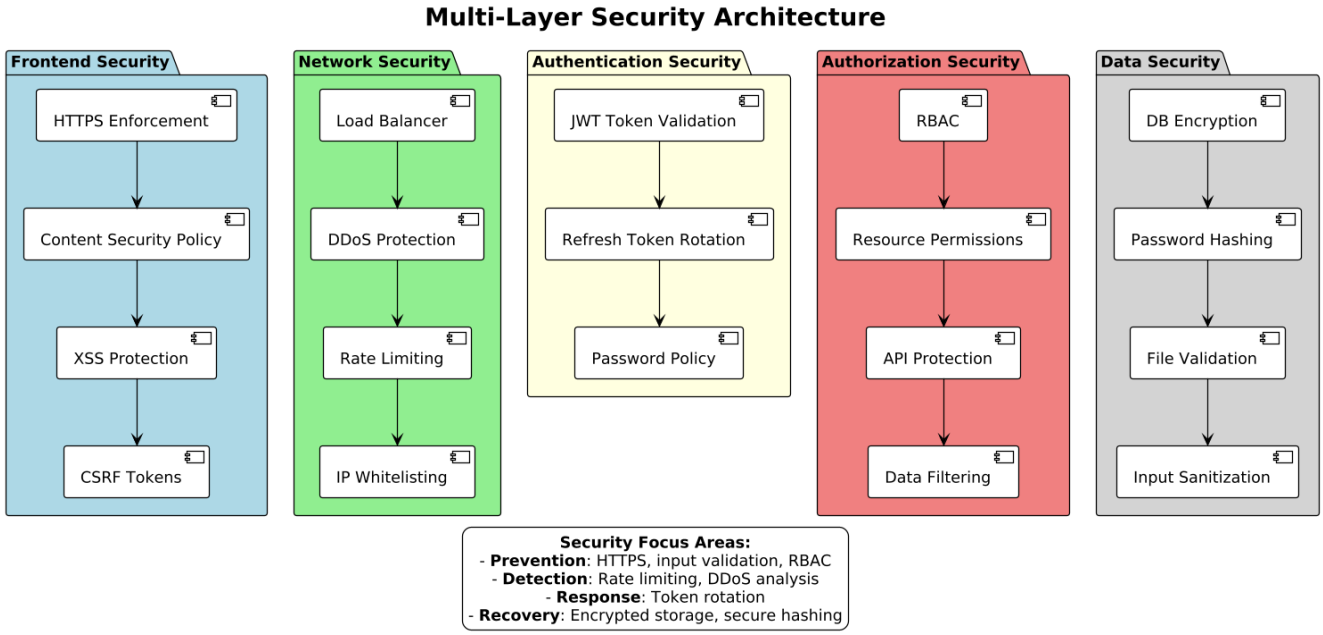

FIGURE 3.7 – Modèle de Sécurité Multi-Couches

Chapitre 4

Mise en œuvre du site web

4.1 Gestion du code source et collaboration : GitHub

Le choix de GitHub comme plateforme d'hébergement du code source et de collaboration a grandement contribué à industrialiser notre processus de développement, en alignant nos pratiques avec les standards professionnels actuels.

4.1.1 Fonctionnalités stratégiques utilisées

— **Gestion des versions :** Chaque modification est tracée via des commits, assurant une traçabilité complète des évolutions du projet.
— **Travail collaboratif :** L'utilisation de branches dédiées et de pull requests permet à chaque développeur de travailler indépendamment tout en maintenant la stabilité de la branche principale.
— **Automatisation des validations :** L'intégration avec des outils externes permet l'exécution automatique de tests et de compilations avant toute fusion, garantissant ainsi la qualité du code.

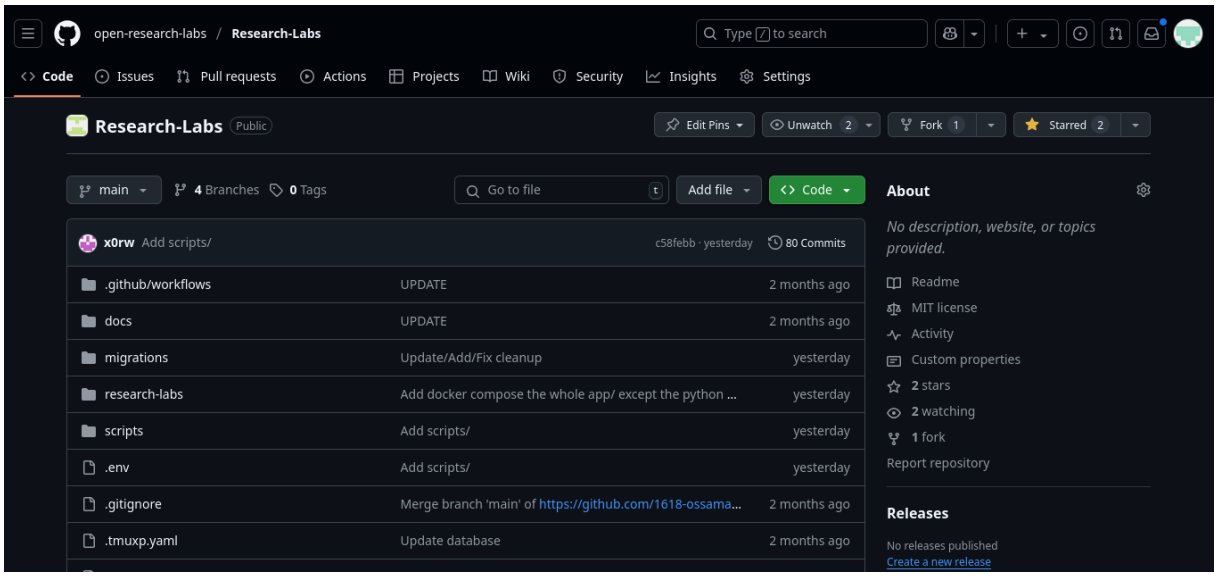

FIGURE 4.1 – GitHub

4.1.2 Organisation du dépôt et gestion des branches

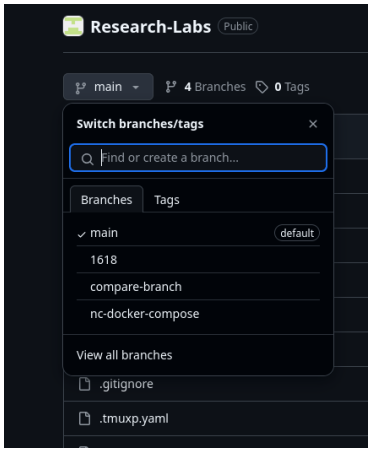

FIGURE 4.2 – Branches Git

La branche `main` conserve la version stable du projet. Pour chaque nouvelle fonctionnalité ou correction, une branche spécifique est créée. Cette organisation minimise les conflits et favorise un développement parallèle efficace.

4.1.3 Demandes de fusion et revue de code

Les pull requests assurent une revue rigoureuse du code par les membres de l'équipe, permettant la détection précoce d'erreurs, l'amélioration de la qualité du code, ainsi que le partage des connaissances.

4.1.4 Suivi des tâches avec GitHub Issues

GitHub Issues a été employé pour documenter et prioriser les tâches. Chaque problème ou nouvelle fonctionnalité fait l'objet d'une issue, facilitant la planification et le suivi de l'avancement.

4.1.5 Impact global de GitHub

Cette méthodologie a permis de centraliser le développement, faciliter le travail asynchrone, maintenir un historique détaillé des modifications, et renforcer la rigueur grâce aux revues systématiques.

4.2 Tests des API : Postman

Postman a été l'outil principal pour tester, documenter et valider les API RESTful développées. Ses fonctionnalités clés incluent :

- **Environnements configurables** pour gérer les différentes phases (développement, production).
- **Collections organisées** permettant de regrouper les endpoints et de faciliter leur réutilisation.
- **Tests automatisés** écrits en JavaScript, garantissant la conformité des réponses (statut, format, performance).

— **Documentation dynamique** générée automatiquement pour faciliter l'intégration frontend/backend.

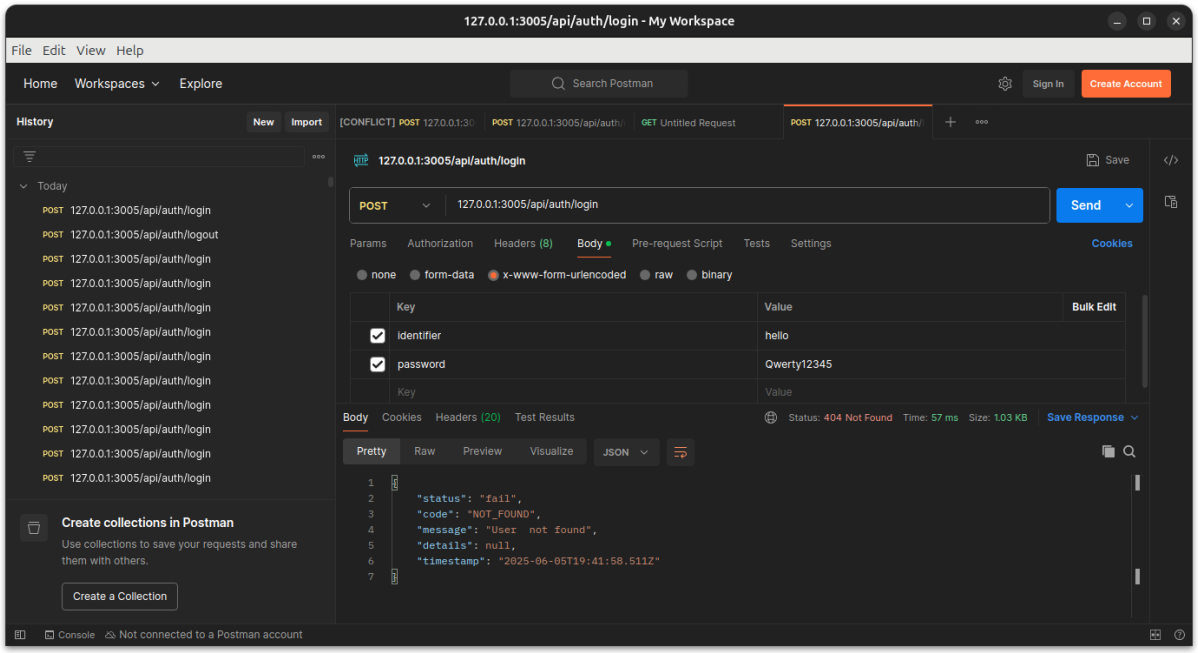

FIGURE 4.3 – Tests d'API

4.3 Tests des WebSockets : wscat

Pour valider les fonctionnalités temps réel telles que les notifications ou le chat, l'outil en ligne de commande `wscat` a été utilisé :

— Connexion directe aux serveurs WebSocket pour envoyer et recevoir des messages en temps réel.
— Tests de stabilité et de latence, permettant un débogage efficace.
— Facilité d'intégration dans des scripts automatisés grâce à sa compatibilité shell.

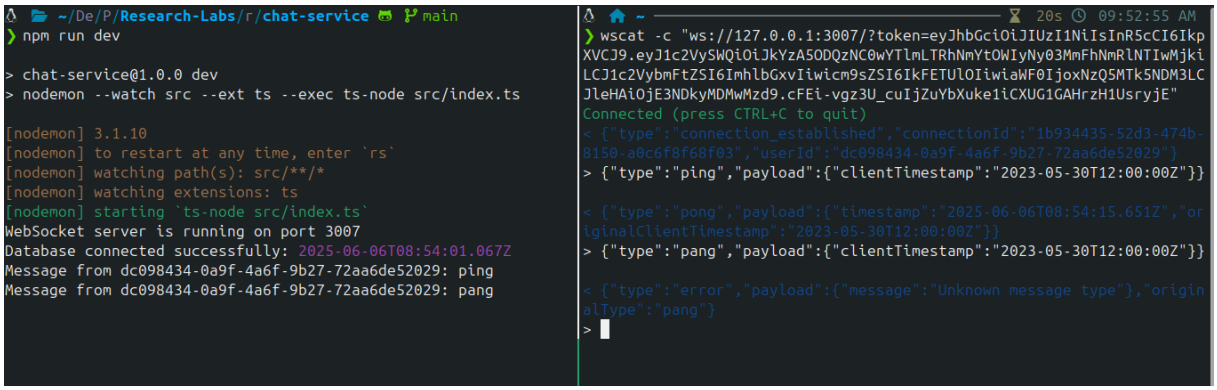

FIGURE 4.4 – Test du serveur chat par wscat

une session de terminal où un serveur de chat est en cours d'exécution et communique avec un client via WebSocket. Le serveur écoute sur le port 3007 et reçoit des messages de

type "ping" du client, auxquels il répond par des messages de type "pong". Cependant, le client envoie également un message de type inconnu, qui est traité par le serveur comme une erreur, illustrant comment le serveur gère les types de messages attendus et les erreurs.

4.4 Gestion et interrogation des bases de données : DBeaver

DBeaver a permis une gestion centralisée des bases de données du projet, notamment PostgreSQL, grâce à :

- Un éditeur SQL puissant avec coloration syntaxique et auto-complétion.
- Une visualisation claire des schémas, tables, relations et index.
- Des fonctionnalités d'import/export pour manipuler facilement les données.
- La gestion sécurisée des connexions (SSH, SSL) pour protéger l'accès aux données.

4.5 Conteneurisation et orchestration : Docker

Docker a été un pilier de notre mise en œuvre, assurant la cohérence des environnements de développement et de tests.

4.5.1 Mise en œuvre technique

- Utilisation de `Dockerfiles` pour définir des images reproductibles des services backend, frontend et base de données.
- Orchestration avec `Docker Compose` pour gérer le lancement simultané et les dépendances entre conteneurs.
- Volumes persistants configurés pour assurer la durabilité des données malgré la recréation des conteneurs.

4.5.2 Bénéfices

- Isolation stricte des environnements pour éviter les conflits de dépendances.
- Portabilité permettant une exécution identique en local, en staging et en production.
- Préparation facilitée à la scalabilité et à l'intégration future avec Kubernetes.
- Intégration possible avec GitHub Actions pour automatiser les builds et tests continus.

4.6 Synthèse

Cette phase de mise en œuvre a permis d'établir une base solide pour le développement du site web, en s'appuyant sur des outils et méthodes professionnels garantissant qualité, traçabilité et collaboration efficace. Bien que le déploiement en production ne soit pas encore réalisé, toutes les préparations nécessaires ont été intégrées dès cette étape afin de faciliter cette future étape.

Conclusion Générale

Ce projet s'inscrit dans une volonté d'améliorer la gestion des activités de recherche à travers une plateforme centralisée. Grâce à une architecture moderne basée sur des microservices, une base de données PostgreSQL, une interface intuitive et une sécurité renforcée, l'application répond aux besoins croissants des laboratoires universitaires. Les perspectives d'évolution incluent l'intégration d'outils analytiques avancés et l'extension vers d'autres structures de recherche.

Bibliographie