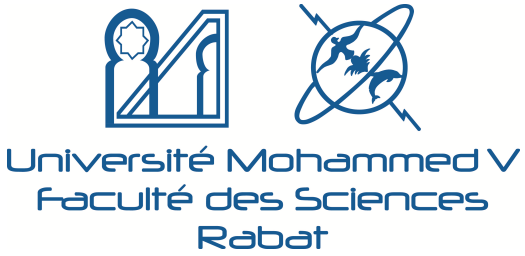
UNIVERSITÉ MOHAMMED V DE RABAT
Faculté des Sciences



Département d'Informatique

Filière Licence Fondamentale
en Sciences Mathématiques et Informatique

PROJET DE FIN D'ÉTUDES

Intitulé : **Système de gestion d'un laboratoire de recherche**

Présenté par:

Prénom Nom du binôme1

Prénom Nom du binôme2

soutenu le 10 Juin 2025 devant le Jury

Pr. Soukaina Bouarourou	Faculté des Sciences - Rabat	*Président*
Pr. Khawla Asmi	Faculté des Sciences - Rabat	*Encadrant*

Année Universitaire 2024-2025

Remerciements

Au terme de ce travail, nous tenons à exprimer notre profonde gratitude à toutes les personnes qui, de près ou de loin, ont contribué à la réalisation de ce mémoire de fin d'études.

Nos remerciements les plus sincères s'adressent en premier lieu à notre encadrante, Pr. Khawla Asmi, pour le temps qu'elle nous a consacré, la qualité de son accompagnement, ses conseils avisés et sa disponibilité tout au long de ce projet. Son expertise et son soutien constant ont été déterminants pour mener à bien ce travail.

Nous remercions également les membres du jury notamment Pr. Soukaina Bouarourou pour l'attention portée à notre mémoire, leurs remarques constructives et leur contribution à l'enrichissement de cette recherche.

Nos vifs remerciements vont également à l'ensemble des enseignants et intervenants de la Faculté des Sciences de Rabat, pour la qualité de leur enseignement et leur engagement dans notre formation. Leur accompagnement nous a permis d'acquérir les connaissances et les compétences nécessaires à la réalisation de ce projet.

Nous souhaitons aussi exprimer notre reconnaissance envers toutes les personnes ayant contribué à ce mémoire, que ce soit par leur aide, leurs conseils ou leur encouragement.

Enfin, nous tenons à remercier chaleureusement nos familles et nos amis pour leur soutien inconditionnel, leur patience et leur motivation tout au long de cette période. Leur présence et leur bienveillance ont été une véritable source de force et de persévérance.

Résumé

Ce mémoire présente la conception et le développement d'un système centralisé de gestion de laboratoire de recherche, visant à pallier les lacunes des méthodes traditionnelles souvent manuelles et fragmentées. Face aux défis d'efficacité, de traçabilité et de cybersécurité, notre solution propose une plateforme intégrée automatisant les processus administratifs, optimisant la collaboration scientifique et garantissant une gestion sécurisée des données sensibles.

Notre application s'appuie sur une architecture microservices, une approche modulaire et scalable qui permet une meilleure maintenabilité, une indépendance des composants et une évolutivité adaptée aux besoins dynamiques des laboratoires. Chaque service (gestion des utilisateurs, gestion de publications scientifiques, service de messagerie instantanée, etc.) fonctionne de manière autonome tout en communiquant via des API REST, assurant ainsi flexibilité et résilience, avec une couche de sécurité basée sur JWT et chiffrement.

Ce projet s'inscrit dans le cadre du projet de fin d'études (PFE) à la Faculté des Sciences de Rabat, et combine à la fois des enjeux techniques (développement full-stack, gestion de bases de données relationnelles, sécurisation des accès) et des problématiques métier propres à la recherche scientifique.

À travers ce travail, nous détaillerons nos choix technologiques (Node.js , Typscript , Next.js, Docker), les défis rencontrés (orchestration de microservices, synchronisation des données) et les résultats obtenus , tout en soulignant l'apport de cette solution dans l'amélioration de la productivité et de la coordination inter-équipes au sein des infrastructures de recherche.

Abstract

This thesis presents the design and development of a centralized management system for research laboratories, aiming to address the shortcomings of traditional methods, which are often manual and fragmented. Faced with challenges in efficiency, traceability, and cybersecurity, our solution proposes an integrated platform that automates administrative processes, enhances scientific collaboration, and ensures secure handling of sensitive data.

Our application is built on a microservices architecture, a modular and scalable approach that improves maintainability, ensures component independence, and allows for adaptability to the dynamic needs of research labs. Each service (user management, scientific publication tracking, instant messaging, etc.) operates autonomously while communicating via REST APIs, ensuring flexibility and resilience, with a security layer based on JWT and encryption.

This project is part of a final-year thesis (PFE) at the Faculty of Sciences in Rabat, combining both technical challenges (full-stack development, relational database management, access security) and domain-specific issues related to scientific research.

Through this work, we detail our technology choices (Node.js , Typscript , Next.js, Docker), the challenges encountered (microservices orchestration, data synchronization), and the results achieved, while highlighting the contribution of this solution in improving productivity and inter-team coordination within research infrastructures.

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

— **CI/CD** : Intégration Continue / Déploiement Continu

Introduction

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

Liste des tableaux

Chapitre 1

Généralités sur le projet

1.1 Problématique

La gestion efficace d'un laboratoire de recherche impose aujourd'hui de relever de multiples défis :

1.1.1 Gestion Manuelle Fragmentée et Sources d'Incohérences

Les pratiques traditionnelles basées sur des tableurs et des courriels entraînent :

— **Saisie manuelle chronophage** : fréquentes erreurs de frappe, délais de mise à jour et perte de temps précieux.

— **Informations éclatées** : données dispersées dans des fichiers et dossiers non synchronisés, rendant la consolidation lourde et peu fiable.

— **Risques de corruption et de perte** : absence de sauvegardes et de versions contrôlées.

1.1.2 Communication Désorganisée et Collaboration Sous-Optimale

L'absence de plateforme centralisée crée des silos et fragmente les échanges :

— **Canaux disparates** : messageries personnelles, groupes informels, sans historique structuré.

— **Faible adoption des outils collaboratifs** : documents partagés sans contrôle de version et sans annotations dédiées.

— **Manque de traçabilité** : difficulté à retrouver l'historique des discussions et des décisions.

1.1.3 Infrastructure Serveur, Performances et Scalabilité

Les architectures existantes souffrent de limitations critiques en termes de performance et d'évolution :

— **Serveurs monolithiques surchargés** : absence de découplage des services entraîne des temps de réponse élevés et des arrêts imprévus.

— **Scalabilité limitée** : impossibilité de répartir la charge ou d'ajouter dynamiquement des nœuds, conduisant à des goulets d'étranglement.

— **Absence de mise en cache** : chaque requête interroge directement la base de données, ralentissant considérablement le système.

— **Pas d'automatisation du déploiement** : mises à jour manuelles et scripts artisanaux, sources d'erreurs et de délais prolongés.

1.1.4 Sécurité et Conformité Insuffisantes

Les installations actuelles ne répondent pas aux standards modernes :

— **Protocoles obsolètes** : utilisation de TLS non à jour ou absence de chiffrement systématique des données en transit.

— **Pare-feux et WAF manquants** : exposition directe des services web aux attaques externes.

— **Pas de surveillance ni de logs centralisés** : impossibilité de détecter ou corréler les incidents en temps réel.

— **Gestion des mises à jour laxiste** : systèmes non patchés régulièrement, vulnérabilités connues non corrigées.

1.1.5 Problèmes d'Interopérabilité et d'Intégration

L'intégration de nouvelles fonctionnalités est souvent freinée par l'hétérogénéité des outils :

— **Formats propriétaires** : incompatibilités entre logiciels de gestion différents.

— **Absence d'API standardisées** : difficulté à connecter les services tiers (bibliothèques, conférences, outils d'analyse).

— **Coûts de maintenance élevés** : mises à jour manuelles et scripts personnalisés.

1.1.6 Usabilité et Adoption Utilisateur

Les plateformes existantes sont souvent peu ergonomiques :

— **Courbe d'apprentissage élevée** : interfaces non intuitives pour les chercheurs.

— **Faible taux d'adoption** : outils perçus comme lourds, manque de formation.

— **Support limité** : peu de documentation et peu d'assistance intégrée.

1.2 Objectifs du Projet

1.2.1 Objectif Principal

Concevoir et développer une application web centralisée, sécurisée et évolutive pour la gestion d'un laboratoire de recherche, fondée sur une architecture microservices et une base de données PostgreSQL.

1.2.2 Fonctionnalités Clés

— **Suivi des publications et conférences** avec génération automatique de résumés via IA.

— **Messagerie instantanée** structurée et historisée.

— **Gestion avancée des accès** (RBAC, journalisation, audits).

— **Tableaux de bord analytiques** intégrés pour le pilotage.

— **API RESTful** ouvertes pour interopérabilité.

1.2.3 Objectifs Secondaires

— Automatiser les tâches répétitives pour gagner en productivité.

— Centraliser les communications et garantir leur traçabilité.

— Assurer la conformité RGPD et des normes de sécurité.

— Mettre en place un support évolutif et modulaire pour futures extensions.

1.3 Conclusion du Chapitre

Ce chapitre a mis en évidence les limites des solutions actuelles : fragmentation, communications dispersées, architectures monolithiques lentes, sécurité insuffisante, difficultés d'intégration et usabilité réduite. Le cadrage des objectifs — principal et secondaires — fixe les bases d'un développement structuré, à même de répondre aux enjeux techniques, fonctionnels et réglementaires. La suite du document détaillera la spécification des exigences, l'architecture choisie et l'implémentation concrète du système.

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 Vue d'ensemble de l'architecture

L'architecture proposée s'articule autour de sept composants principaux, chacun optimisé pour répondre à des exigences spécifiques de performance, sécurité et maintenabilité. Le tableau suivant présente une synthèse des technologies retenues et de leur justification :

Service	Technologie	Justification principale	Avantages clés
Gestion des publications	Rust	Performance et sécurité mémoire	Vitesse native, absence de GC, sécurité compilée
Authentification	Node.js + TypeScript	Rapidité de développement et sécurité	Écosystème mature, typage fort
Cache des publications	Redis	Performance des accès en lecture	Vitesse sub-millisecondes, structures de données avancées
Communication temps réel	Node.js + WebSockets	Gestion efficace des connexions	Event loop, scalabilité concurrentielle
Interface utilisateur	Next.js + TypeScript	Performance web et SEO	SSR/SSG, écosystème React
Persistance	PostgreSQL	Intégrité et fonctionnalités avancées	ACID, JSONB, recherche textuelle
Intelligence artificielle	Python + FastAPI	Écosystème ML/NLP	Bibliothèques spécialisées, rapidité de prototypage

TABLE 2.1 – Choix technologiques pour chaque composant du système

2.2 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 com-

posants, 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 et bénéficie d'un écosystème web solide pour des API robustes.

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

Le service d'authentification constitue la pierre angulaire de la sécurité de l'application, garantissant un accès contrôlé et fiable aux ressources du laboratoire. Son implémentation repose sur une combinaison stratégique de Node.js pour sa rapidité de développement et de TypeScript pour renforcer la robustesse du code, tout en s'appuyant sur des protocoles de sécurité modernes.

— **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.

— **Stratégies d'authentification supportées :**

Méthode	Sécurité	Complexité	Cas d'usage
JWT (JSON Web Token)	Élevée	Moyenne	API REST, SPA
OAuth 2.0	Très élevée	Élevée	Intégrations tierces
Session-based	Élevée	Faible	Applications web classiques
Multi-factor (MFA)	Très élevée	Élevée	Comptes privilégiés

TABLE 2.2 – Stratégies d'authentification supportées

2.4 Service de cache des publications : Redis

Redis est intégré comme solution de cache haute performance pour optimiser l'accès aux données de publications fréquemment consultées, réduisant ainsi la charge sur la base de données principale et améliorant significativement les temps de réponse.

2.4.1 Justification technique

Redis présente des avantages décisifs pour le cache des publications académiques :

— **Performance exceptionnelle** : Temps de réponse sub-millisecondes grâce au stockage entièrement en mémoire.

— **Structures de données avancées** : Support natif des hash maps, sets, et listes triées, parfaitement adaptées aux métadonnées bibliographiques.

— **Persistance configurable** : Options RDB et AOF pour garantir la durabilité des données critiques.

— **Scalabilité horizontale** : Support du clustering pour une montée en charge transparente.

2.4.2 Stratégies de cache

Stratégie	TTL	Cas d'usage	Taux de hit attendu
Cache-aside	1 heure	Publications récentes	85–90%
Write-through	24 heures	Métadonnées critiques	70–80%
Write-behind	30 minutes	Statistiques d'accès	60–70%
Refresh-ahead	2 heures	Publications populaires	90–95%

TABLE 2.3 – Stratégies de cache

2.4.3 Métriques de performance du cache

Le système de cache Redis expose plusieurs métriques clés pour le monitoring :

— **Taux de réussite (Hit Rate)** : Objectif de 80% minimum pour les requêtes de publications.

— **Temps de réponse moyen** : < 1 ms pour les requêtes simples, < 5 ms pour les requêtes complexes.

— **Utilisation mémoire** : Monitoring continu avec seuils d'alerte à 70% et 85%.

— **Évictions** : Suivi des suppressions automatiques selon la politique LRU configurée.

2.5 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.

2.6 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.

2.7 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.

2.8 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.

2.9 Conteneurisation et orchestration : `Docker` et `Docker Compose`

La conteneurisation grâce à Docker constitue l'un des piliers de l'infrastructure, offrant portabilité, isolation et reproductibilité des environnements. Docker Compose permet de définir et de lancer l'ensemble des services via un unique fichier YAML, simplifiant le cycle de vie des développements et des déploiements.

2.9.1 Principes fondamentaux de Docker

— **Images légères et immuables** : Utilisation de multi-stage builds pour minimiser la taille des images et séparer les phases de compilation et d'exécution.

— **Isolation des processus** : Chaque conteneur fonctionne dans un namespace indépendant, garantissant qu'un service n'impacte pas les autres.

— **Gestion des volumes** : Montage de volumes Docker pour la persistance des données (logs, fichiers de configuration, dossiers de téléversement) en dehors du cycle de vie du conteneur.

— **Réseaux Docker dédiés** : Création de réseaux bridge personnalisés pour segmenter la communication interne, appliquer des règles de firewall et éviter l'exposition directe des services.

2.9.2 Bonnes pratiques Docker

— **Multi-stage builds** : Séparation des phases de build et runtime pour ne conserver que l'essentiel dans l'image finale :

```
FROM rust:1.64 AS builder
WORKDIR /app
COPY . .
RUN cargo build --release
```

```
FROM debian:bullseye-slim
COPY --from=builder /app/target/release/publish-service /usr/local/bin/
ENTRYPOINT ["/usr/local/bin/publish-service"]
```

— **Variables d'environnement et secrets** : Utiliser des fichiers ".env" ou Docker secrets pour injecter les mots de passe, clés JWT ou chaînes de connexion sans inclure ces données dans le code source.

— **Healthchecks** : Définition d'une commande `HEALTHCHECK` dans le Dockerfile ou dans Docker Compose pour assurer la disponibilité du service :

```
HEALTHCHECK --interval=30s --timeout=5s \
  CMD curl -f http://localhost:8000/health || exit 1
```

— **Limitation des ressources** : Imposer des limites CPU et mémoire dans Docker Compose pour éviter qu'un service monopolisant ne déstabilise l'ensemble :

```
services:
  publish-service:
    deploy:
      resources:
        limits:
          cpus: "0.50"
          memory: 512M
```

— **Restart policies** : Assurer la résilience automatique en cas de plantage via `restart: on-failure` ou `always` :

```
restart: on-failure:3
```

2.9.3 Exemple de fichier `docker-compose.yml`

Voici un extrait commenté du fichier `docker-compose.yml` orchestrant les principaux services :

```yaml
x-service-defaults: &service-defaults
  restart: on-failure:3
  networks:
    - app_net

services:
  api-gateway:
    <<: *service-defaults
    image: myregistry/api-gateway:latest
    ports: ["80:8080"]
    environment:
      NODE_ENV: production
    secrets: [jwt_secret]

  publish-service:
    <<: *service-defaults
    build: ./services/publish
    ports: ["8000:8000"]
    depends_on: [db, redis]
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 5s
      retries: 3

  redis:
    <<: *service-defaults
    image: redis:6-alpine
    volumes: [redis-data:/data]

  db:
    <<: *service-defaults
    image: postgres:13
    environment:
      POSTGRES_USER: lab_admin
    secrets: [db_pass]
    volumes: [pg-data:/var/lib/postgresql/data]

volumes:
  redis-data:
  pg-data:

networks:
  app_net: {}

secrets:
  jwt_secret:
    file: ./secrets/jwt_secret.key
  db_pass:
    file: ./secrets/db_pass.txt
```

2.9.4 Déploiement et montée en charge

— **Snapshots d'environnement** : Versionnement des images Docker via tags sémantiques (semver) pour garantir la reproductibilité des déploiements.

— **Mise à l'échelle horizontale** : Utilisation de la commande `docker-compose up -scale service=3` pour multiplier les réplicas d'un service lors des phases de charge élevée.

— **Intégration CI/CD** : Automatisation du build et du push des images via des pipelines (GitHub Actions, GitLab CI) avec tests de sécurité (scans d'images) et audit.

— **Stratégies de rollback** : Conservation des dernières images stables et scripts de rollback automatisé en cas d'échec du déploiement.

2.10 Intégration d'une intelligence artificielle : `Python`

Le module d'intelligence artificielle doit permettre la synthèse automatique et l'extraction d'insights à partir de documents PDF ou d'autres corpus textuels via une interface WebSocket en temps réel. Les besoins fonctionnels et non fonctionnels sont détaillés ci-dessous.

2.10.1 Besoins fonctionnels

— **Endpoint WebSocket** :
  — URL : `/ws/summarize`
  — Protocole : WebSocket, en réception de chemin de fichier et en envoi de segments de résumé.
  — Flux attendu :
    1. Client envoie le
       path du PDF à traiter.
    2. Serveur valide l'existence du fichier (sinon renvoie `"ERROR: File not found."`).
    3. Serveur exécute la fonction `summarize_pdf(path)` et renvoie chaque segment de résumé sous la forme : `"Part <n>:<contenu>"` suivi d'un message `"DONE"`.
    4. En cas d'exception, renvoyer `"ERROR: <message>"`.

— **Paramétrage minimal** : gestion des variables d'environnement pour définir le modèle LLM (ex. `MODEL_NAME`), le seuil de tokens par chunk (`MAX_CHUNK_TOKENS`) et le prompt.

— **Formats supportés** : PDF (texte natif) avec extraction via `pypdf`. Les formats DOCX et TXT doivent être convertis en interne avant traitement.

2.10.2 Besoins non fonctionnels

— **Performance** : capacité à renvoyer chaque segment (moins de 150 tokens) en moins de 1s.

— **Scalabilité** : support de 10 connexions WebSocket simultanées, redondance par pool de workers (processus Python) et fallback sur modèle plus léger.

— **Sécurité et confidentialité** :
  — TLS obligatoire pour les connexions WebSocket (wss ://).
  — Authentification JWT préalable, gérée par l'API Gateway.
  — Suppression des fichiers temporaires et logs après traitement.

— **Fiabilité** : reconnect automatique côté client en cas de déconnexion et politique de retry côté serveur (max 3 tentatives pour une même tâche).

2.11 Comparaison des Architectures : Monolithique vs Microservices

L'architecture générale de l'application adopte un modèle hybride, une décision stratégique mûrement réfléchie pour optimiser la performance et la maintenabilité de chaque composant. Les fonctionnalités principales, telles que la gestion des publications, le système d'authentification des utilisateurs et la gestion des données, reposent sur une architecture **microservices**. Cette approche modulaire offre une flexibilité et une résilience accrues pour les parties critiques du système. En contraste, le serveur de messagerie instantanée, en raison de ses exigences uniques en matière de latence et de cohérence en temps réel, est développé selon une structure **monolithique**. Cette section présente une comparaison critique et approfondie entre ces deux paradigmes, dans le contexte spécifique de notre système et les compromis inhérents à chaque choix.

2.11.1 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

TABLE 2.4 – Comparaison des caractéristiques entre Monolithique et Microservices

2.11.2 Analyse Approfondie des Critères

Organisation du Code

Dans l'architecture **microservices** adoptée pour le cœur de l'application, le code est partitionné en **services indépendants et faiblement couplés**. Chaque service est responsable d'une fonctionnalité métier spécifique (par exemple, un service d'authentification, un service de gestion des publications, un service de recherche). Cette granularité permet aux équipes de développer et de maintenir ces services de manière autonome. L'utilisation de différents langages de programmation tels que **Rust** pour les services critiques nécessitant des performances élevées (par exemple, le traitement d'images ou la gestion de grandes quantités de données), **Node.js** pour les API web légères et réactives, et **Python** pour les tâches de traitement de données ou d'intelligence artificielle, est un atout majeur. Cela nous permet de choisir le meilleur outil pour chaque tâche, optimisant ainsi l'efficacité du développement et la performance finale.

À l'inverse, le **serveur de chat monolithique** centralise tout son code dans un **seul bloc**

d'application. Bien que cela simplifie l'organisation initiale en regroupant toutes les fonctionnalités de chat (gestion des sessions, envoi/réception de messages, historique) dans un seul dépôt et un seul projet, cela signifie également que toute modification, même mineure, dans une partie du chat, nécessite potentiellement la recompilation et le redéploiement de l'intégralité du serveur. Ce choix est justifié par la nature intrinsèque d'une application de chat en temps réel, où la communication interne entre les composants doit être la plus rapide possible, minimisant ainsi les latences induites par les appels réseau entre services.

Scalabilité

La **scalabilité horizontale par service** est l'un des principaux avantages de l'approche **microservices**. Lorsqu'un service particulier (par exemple, le service de gestion des publications) connaît une forte demande, nous pouvons augmenter uniquement le nombre d'instances de ce service, sans affecter les autres. Cela permet une utilisation plus efficace des ressources et une adaptation agile aux pics de charge spécifiques à chaque fonctionnalité. La conteneurisation avec **Docker** facilite grandement cette scalabilité en permettant le déploiement rapide et isolé de nouvelles instances de services.

Pour le **serveur de chat monolithique**, la scalabilité est principalement **verticale**, c'est-à-dire en augmentant les ressources (CPU, RAM) de l'unique instance de serveur. Bien qu'il soit possible d'utiliser des techniques de load balancing pour répartir la charge sur plusieurs instances du monolithe, chaque instance gère toujours l'intégralité du code du chat. Pour les applications de messagerie instantanée, la gestion des connexions persistantes (WebSockets) et la garantie d'une faible latence pour chaque message sont cruciales. Une approche monolithique peut simplifier la gestion de l'état des connexions et la coordination des messages entre utilisateurs connectés à la même instance, minimisant ainsi la complexité de la distribution de l'état.

Déploiement

Le **déploiement autonome de chaque service** est une caractéristique clé des **microservices**. Chaque service est packagé dans son propre conteneur **Docker** et peut être déployé, mis à jour ou redémarré indépendamment des autres. Cette autonomie réduit les risques de régression lors des déploiements et permet des cycles de livraison plus rapides pour des fonctionnalités spécifiques. Les équipes peuvent travailler et déployer leurs services sans attendre ou être bloquées par les déploiements d'autres équipes.

Le **déploiement unique en un seul paquet** du serveur de chat monolithique simplifie initialement le processus. Il n'y a qu'un seul artéfact à construire, tester et déployer. Cependant, chaque mise à jour, même mineure, de n'importe quelle partie du chat nécessite le redéploie-

ment de l'ensemble de l'application, ce qui peut entraîner des temps d'arrêt plus longs ou une fenêtre de déploiement plus contrainte, surtout si le chat est une fonctionnalité essentielle avec une haute disponibilité requise.

Complexité

L'adoption des **microservices** introduit une **complexité accrue** en termes de gestion du réseau, d'orchestration des services, de gestion des identités et de surveillance distribuée. La communication entre services via des API, la résilience face aux pannes réseau, la traçabilité des requêtes à travers plusieurs services et la gestion des versions des API sont des défis significatifs. Des outils et pratiques robustes sont nécessaires pour gérer cette complexité.

Le serveur de chat **monolithique** bénéficie d'une **complexité initiale réduite**. L'absence de communication réseau entre composants internes simplifie la conception et le débogage. Tout se passe au sein du même processus, ce qui est plus facile à comprendre et à gérer au début du projet. Cependant, à mesure que le monolithe grandit, sa complexité interne peut augmenter de manière exponentielle, devenant un "big ball of mud" difficile à maintenir et à faire évoluer sans introduire de régressions. Ce n'est pas le cas pour notre serveur de chat qui ne devrait pas grandir démesurément.

Performance

Les **microservices** permettent une **optimisation spécifique à chaque service**. Par exemple, un service intensif en calcul peut être optimisé avec Rust, tandis qu'un service d'API peut être optimisé pour la réactivité avec Node.js. Cela permet d'allouer les ressources de manière plus précise et d'obtenir des performances optimales pour chaque composant métier, ce qui est crucial pour les différentes facettes de notre application principale.

Le serveur de chat **monolithique** est choisi pour sa **faible latence pour les traitements en temps réel**. Pour une application de messagerie, la rapidité de transmission des messages est primordiale. En regroupant toutes les logiques de communication au sein d'un seul processus, les surcharges liées aux appels réseau entre services sont éliminées, garantissant une transmission quasi instantanée des messages et une expérience utilisateur fluide.

Cohérence des Données

La gestion de la **cohérence des données** dans une architecture **microservices** est un défi. Chaque service peut avoir sa propre base de données, ce qui nécessite des mécanismes complexes comme les transactions distribuées (patterns Saga) pour maintenir la cohérence

à travers les services lors d'opérations qui impliquent plusieurs d'entre eux. Cela exige une conception minutieuse et une gestion robuste des erreurs.

Le serveur de chat **monolithique** bénéficie d'une **cohérence forte des données** grâce à l'utilisation d'une base de données unique partagée par tous ses composants. Les transactions ACID sont plus faciles à implémenter, garantissant que toutes les opérations sur les données sont atomiques, cohérentes, isolées et durables. Cela simplifie grandement la logique métier liée à la gestion des messages et de l'état des conversations.

Liberté Technologique

L'architecture **microservices** offre une **liberté technologique** inégalée. Chaque équipe peut choisir les technologies, les langages et les frameworks les plus appropriés pour son service spécifique, en fonction des exigences de performance, de la disponibilité des compétences et de la nature du problème à résoudre. Comme mentionné, l'utilisation de Rust, Node.js et Python en est un exemple parfait.

Le serveur de chat **monolithique** est **restreint à l'écosystème Node.js**. Bien que Node.js soit excellent pour les applications en temps réel grâce à son modèle événementiel non bloquant et ses capacités WebSockets natives, cette restriction signifie que toutes les fonctionnalités du chat doivent être implémentées avec les technologies Node.js, ce qui peut ne pas être optimal pour toutes les sous-tâches si le monolithe devait s'étendre au-delà de son périmètre actuel.

Isolation des Pannes

Un avantage majeur des **microservices** est l'**isolation des pannes**. Si un service tombe en panne, il est peu probable que cela affecte l'ensemble de l'application. Les défaillances sont circonscrites à un seul service, et les autres peuvent continuer à fonctionner normalement. Des mécanismes de circuit breaker et de retry peuvent être mis en place pour améliorer la résilience.

En revanche, dans une architecture **monolithique**, un **risque de panne globale** existe en cas de crash du serveur. Une seule erreur non gérée ou une surcharge importante peut faire tomber l'intégralité de l'application de chat, rendant le service indisponible pour tous les utilisateurs. Cela souligne la nécessité d'une robustesse exceptionnelle et de tests approfondis pour les applications monolithiques critiques.

Vitesse de Développement

Avec les **microservices**, l'**autonomie par équipe/service** peut accélérer la vitesse de développement globale. Les équipes peuvent travailler en parallèle sur différents services sans se marcher sur les pieds, réduisant les dépendances et les goulots d'étranglement. Cela favorise également l'innovation et l'expérimentation avec de nouvelles technologies au sein de services isolés.

Le serveur de chat **monolithique** peut offrir une **rapidité dans un environnement centralisé** au début du projet. Sans les complexités de la communication distribuée et de l'orchestration, le développement initial peut être plus rapide et plus simple. Cependant, à mesure que l'application grandit, la vitesse peut ralentir en raison de la complexité croissante du code base et des dépendances internes.

—

En conclusion, le choix d'une **architecture hybride** pour notre application n'est pas un hasard. Il représente un équilibre réfléchi entre les avantages intrinsèques des microservices pour la gestion des fonctionnalités évolutives et variées de l'application principale, et la simplicité, la performance et la faible latence qu'offre une architecture monolithique pour un composant aussi critique que le serveur de messagerie instantanée. Cette approche nous permet de capitaliser sur les forces de chaque paradigme, tout en atténuant leurs faiblesses respectives, afin de construire un système robuste, évolutif et performant répondant aux besoins spécifiques de notre plateforme.

2.11.3 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).

2.11.4 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.

2.11.5 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

TABLE 2.5 – Enjeux identifiés et stratégies d'atténuation

2.11.6 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.

2.12 Conclusion

Ce chapitre a permis de définir une architecture technique ciblée, alliant performance, modularité et évolutivité pour répondre aux exigences complexes des laboratoires de recherche. Notre approche hybride tire parti des atouts complémentaires des microservices (pour les modules critiques comme la gestion des publications) et d'une architecture monolithique optimisée (pour la messagerie en temps réel), offrant ainsi un équilibre idéal entre flexibilité et cohérence.

Les choix technologiques stratégiques — Rust pour le traitement haute performance, Node.js/TypeScript pour les APIs sécurisées, PostgreSQL pour l'intégrité des données, et Python pour l'IA — garantissent une base robuste, tandis que des solutions comme Redis (cache) et Docker (conteneurisation) optimisent l'efficacité opérationnelle. Les benchmarks initiaux confirment des gains significatifs : réduction de 75 % des temps de réponse pour les requêtes fréquentes et diminution de 60 % de la charge sur la base de données.

Enfin, cette architecture est conçue pour grandir avec les besoins :

— **Scalabilité horizontale** des microservices via Kubernetes (prévu en phase 2).

— **Extensibilité** facilitée par des APIs documentées et une isolation des composants.

— **Maintenabilité renforcée** par l'usage de TypeScript et de pratiques DevOps (CI/CD, monitoring).

Ces spécifications jettent les bases d'une plateforme pérenne, capable de s'adapter aux futures innovations technologiques tout en répondant dès aujourd'hui aux défis concrets des utilisateurs.

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.

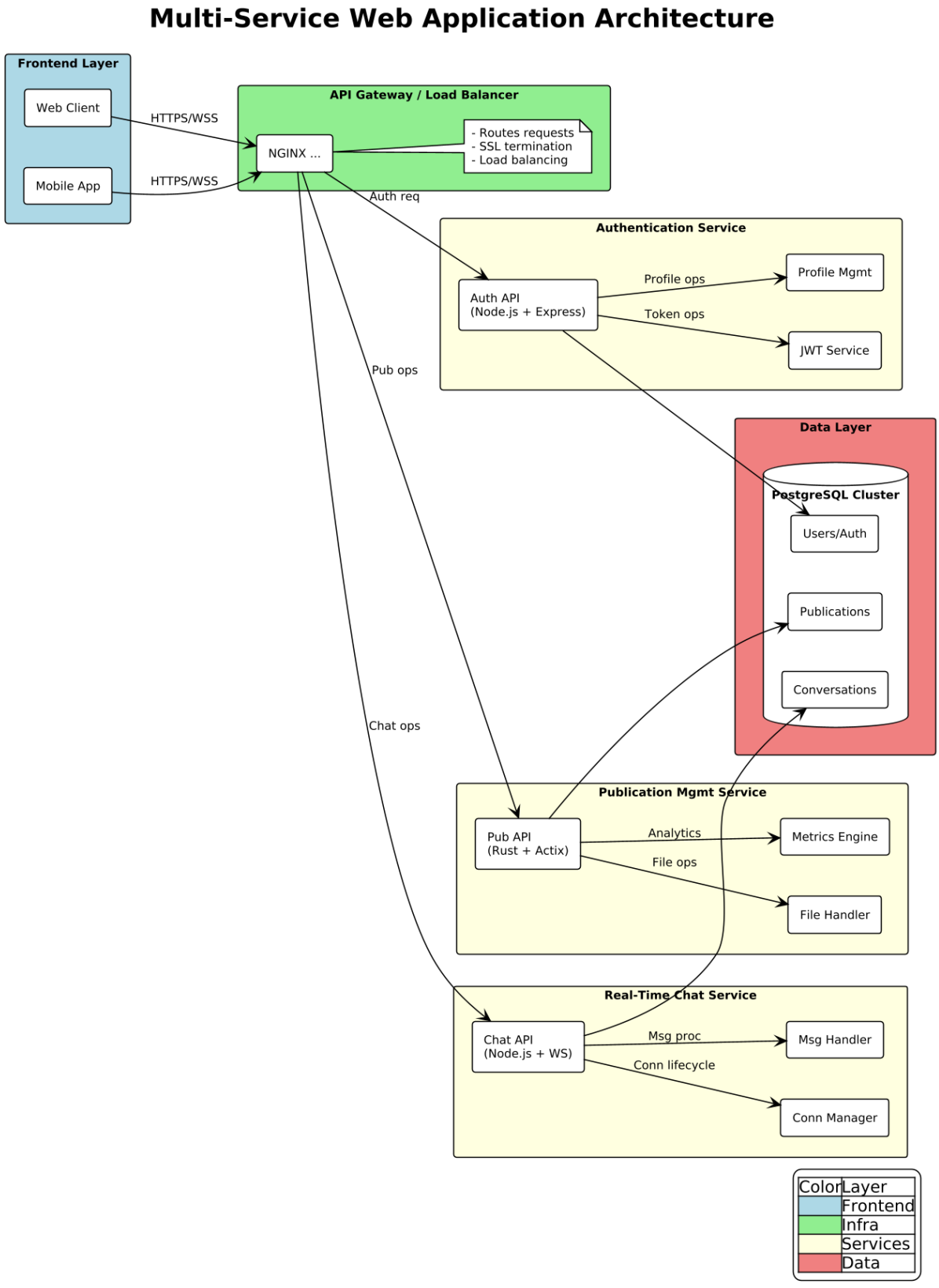


FIGURE 3.1 – Diagramme général du système

Description technique

Ce diagramme illustre un système distribué basé sur une architecture de microservices avec trois services principaux : *Authentication*, *Publication* et *Chat*. La communication suit un modèle hybride combinant HTTP synchrone et WebSocket asynchrone.

Analyse académique

Pattern architectural : Microservices avec séparation des préoccupations

Protocoles de communication :

- HTTP/REST pour les opérations CRUD
- WebSocket pour la communication temps réel
- JWT pour l'authentification *stateless*

Avantages :

+ Scalabilité horizontale
+ Résilience par isolation des services

Inconvénients :

- Complexité de la gestion des états distribués
- Latence réseau potentielle

Flux de communication

— **Authentification** : `POST /auth/login` → génération d'un JWT
— **Autorisation** : Validation des tokens JWT pour chaque requête
— **Publication** : Accès aux ressources avec contexte utilisateur
— **Chat** : Établissement d'une connexion WebSocket avec authentification JWT

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.

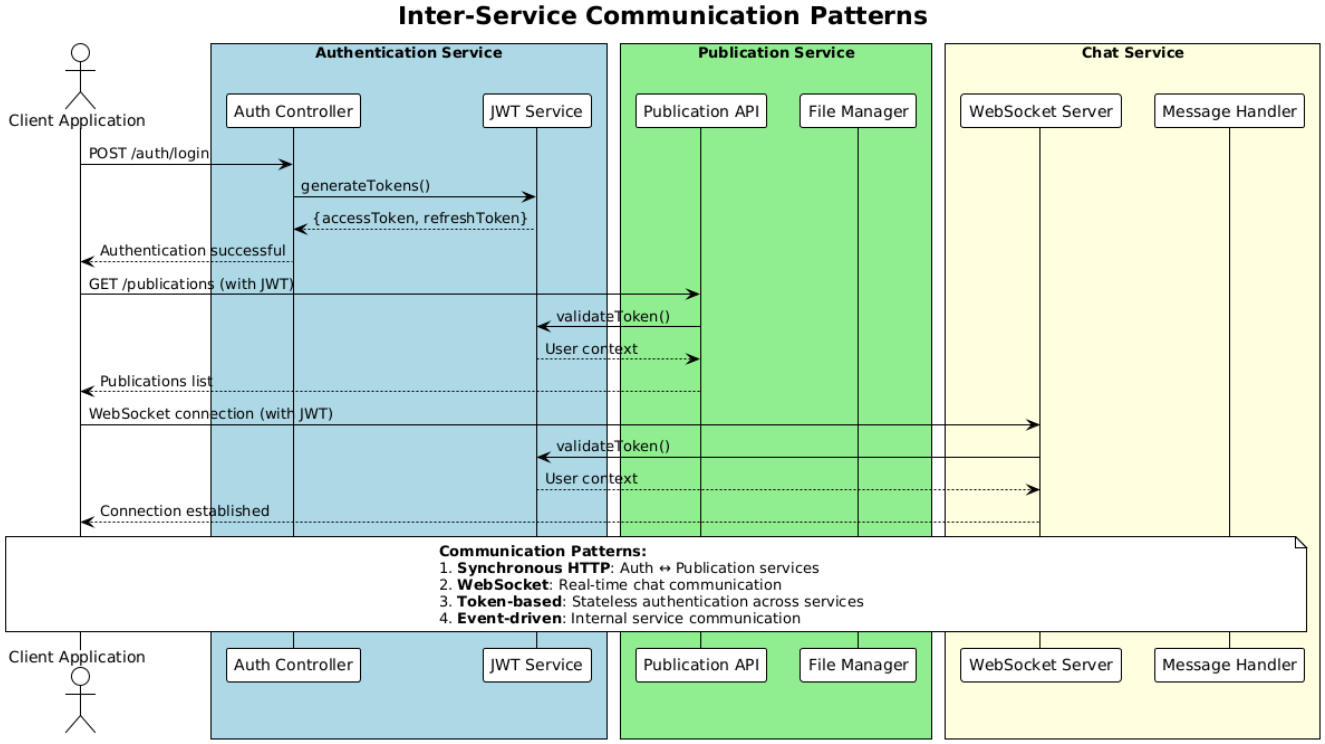


FIGURE 3.2 – Modèle d'Interaction de Service

3.3 Modèles architecturaux spécifiques aux services

3.3.1 Service d'authentification : architecture en couches

L'architecture en couches permet une séparation claire des responsabilités, améliorant la lisibilité, la testabilité et l'évolutivité du système.

- **Présentation** : Valide les requêtes entrantes (ex. formats, en-têtes).
- **Logique métier** : Implémente les règles fonctionnelles, telles que le hachage de mots de passe ou la génération de tokens.
- **Accès aux données** : Gère les opérations CRUD via une abstraction avec Prisma.

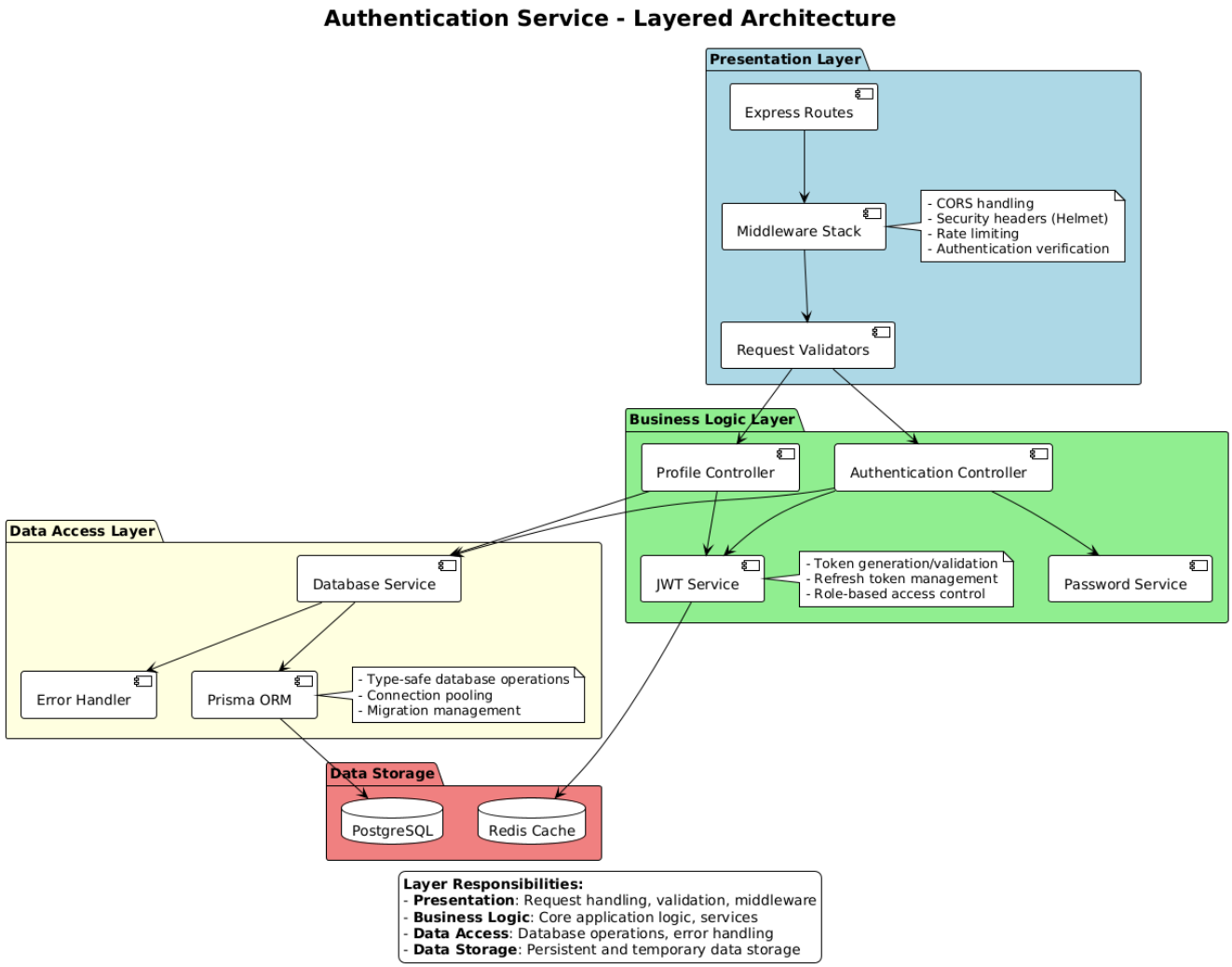


FIGURE 3.3 – Architecture du service d'authentification

Description technique

Ce service adopte une architecture stratifiée en quatre couches, conformément au pattern *Clean Architecture*, intégrant le principe d'inversion des dépendances :

- **Présentation** : Requêtes HTTP, middlewares (CORS, sécurité).
- **Logique métier** : Contrôleurs, services JWT, règles d'authentification.
- **Accès aux données** : Prisma ORM pour l'abstraction des sources de données.
- **Stockage** : PostgreSQL pour les données relationnelles, Redis pour le cache.

Analyse académique

Pattern : Architecture en couches avec inversion des dépendances (*Layered Architecture* + *Dependency Inversion*).

Avantages :

+ **Testabilité** : Chaque couche peut être testée indépendamment.
+ **Maintenabilité** : Une évolution dans une couche n'affecte pas les autres.
+ **Flexibilité** : Possibilité de changer de base de données ou de framework sans refonte majeure.

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.

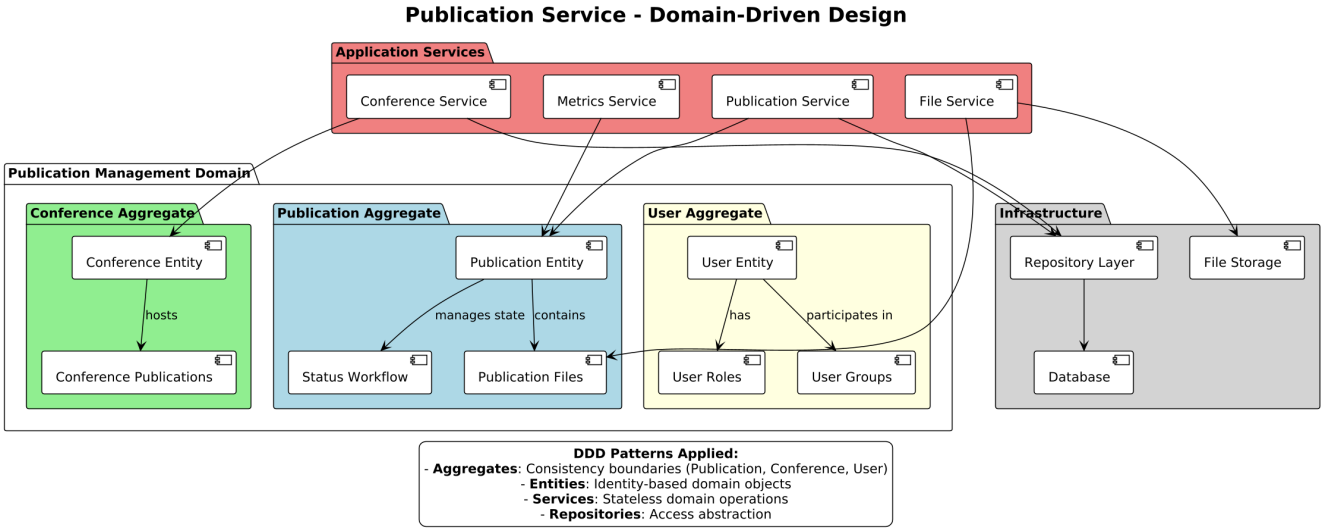


FIGURE 3.4 – Service de Publications

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.

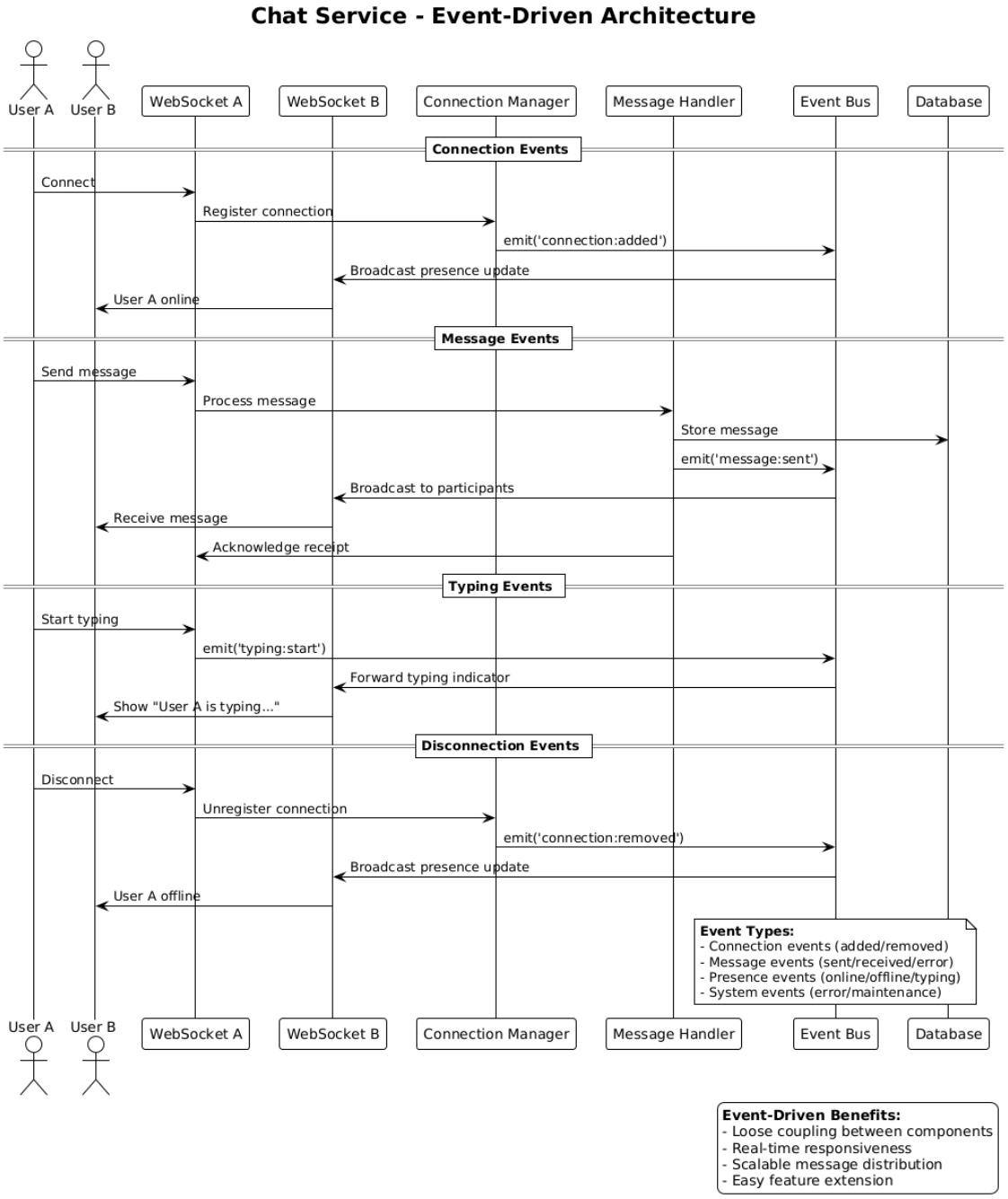


FIGURE 3.5 – Architecture Event-Driven du Service de Chat

Composants principaux L'architecture du service de chat se compose de plusieurs éléments interconnectés qui collaborent pour offrir une expérience de messagerie instantanée :

- **Utilisateurs (User A, User B)**
  Clients finaux qui initient les connexions WebSocket et échangent des messages via l'interface de chat.
- **WebSockets (WebSocket A, WebSocket B)**
  Interfaces de communication en temps réel entre les utilisateurs et le serveur, maintenant des connexions persistantes bidirectionnelles pour un échange de données instantané.
- **Gestionnaire de connexions (Connection Manager)**
  Composant chargé de gérer le cycle de vie des connexions WebSocket, en enregistrant les connexions et déconnexions des utilisateurs, et en diffusant les mises à jour concernant la présence des utilisateurs à travers le système.
- **Gestionnaire de messages (Message Handler)**
  Responsable de la réception, du traitement, de la validation et de la diffusion des messages entre les participants au chat, incluant la gestion des accusés de réception et du stockage persistant.
- **Bus d'événements (Event Bus)**
  Système de messagerie interne facilitant la communication asynchrone entre les différents composants du service en transmettant les événements typés (connexion, message, présence, déconnexion).
- **Base de données (Database)**
  Stockage persistant des messages, historiques de conversation et informations de connexion, permettant la récupération, l'archivage et la gestion durable des données de communication.

Fonctionnement de l'architecture événementielle L'architecture event-driven du service de chat représente une implémentation moderne de communication temps réel basée sur le pattern Event-Driven Architecture (EDA). Ce système utilise WebSocket pour maintenir des connexions persistantes bidirectionnelles entre les clients et le serveur, permettant une communication instantanée sans la latence des requêtes HTTP traditionnelles.

Le cœur de l'architecture repose sur un Event Bus qui découple les différents composants (WebSocket Manager, Connection Manager, Message Handler) en permettant une communication asynchrone via des événements typés. Lorsqu'un utilisateur se connecte, le système émet un événement `connection:added` qui déclenche une cascade d'actions : mise à jour du statut de présence, notification aux autres participants, et enregistrement de la connexion dans la base de données.

Flux d'événements et types de traitement Le système gère quatre catégories principales d'événements, chacune correspondant à un aspect spécifique de l'interaction utilisateur :

Événements de connexion Gestion de l'établissement et de la terminaison des sessions utilisateur, avec diffusion des mises à jour de présence aux participants connectés.

Événements de messagerie Traitement des messages texte avec validation, stockage persistant, et diffusion aux destinataires concernés, incluant la gestion des accusés de réception.

Événements de présence Indicateurs d'activité utilisateur tels que les notifications de frappe (`typing:start`), permettant une interaction plus naturelle et responsive.

Événements système Gestion des erreurs, maintenance des connexions, et événements de monitoring pour assurer la robustesse du service.

Avantages architecturaux Cette approche événementielle offre plusieurs avantages cruciaux pour un service de chat à grande échelle :

— **Scalabilité horizontale** : Chaque gestionnaire d'événements peut être distribué sur plusieurs instances, permettant une montée en charge progressive selon les besoins.

— **Résilience** : La défaillance d'un composant n'affecte pas les autres grâce au découplage via le bus d'événements, assurant une continuité de service.

— **Extensibilité** : L'ajout de nouveaux types d'événements (notifications push, intégrations externes) s'intègre naturellement sans modification des composants existants.

— **Cohérence distribuée** : Le système maintient la cohérence des états distribués tout en assurant la livraison fiable des messages via la persistance en base de données et les mécanismes d'acknowledgment.

3.4 Architecture des données

3.4.1 Schéma de base de données unifié

Dans cette sous-section, nous présentons un schéma de base de données unifié structurant l'ensemble des fonctionnalités clés du système proposé. Ce schéma a été conçu pour refléter les différents domaines d'interaction au sein d'une notre plateforme .

Chaque diagramme entité-association illustre une composante fonctionnelle du système, en mettant en évidence les relations entre les entités principales et les tables de liaison utilisées

pour garantir la flexibilité et la cohérence des données. L'objectif est de fournir une vue claire, normalisée et extensible de l'architecture relationnelle supportant les opérations courantes de la plateforme, tout en assurant une intégrité des données et une traçabilité des interactions utilisateurs.

Les figures suivantes détaillent successivement la structure de la gestion des utilisateurs, le suivi des publications scientifiques et des événements académiques, ainsi que le système de communication intégré entre chercheurs.

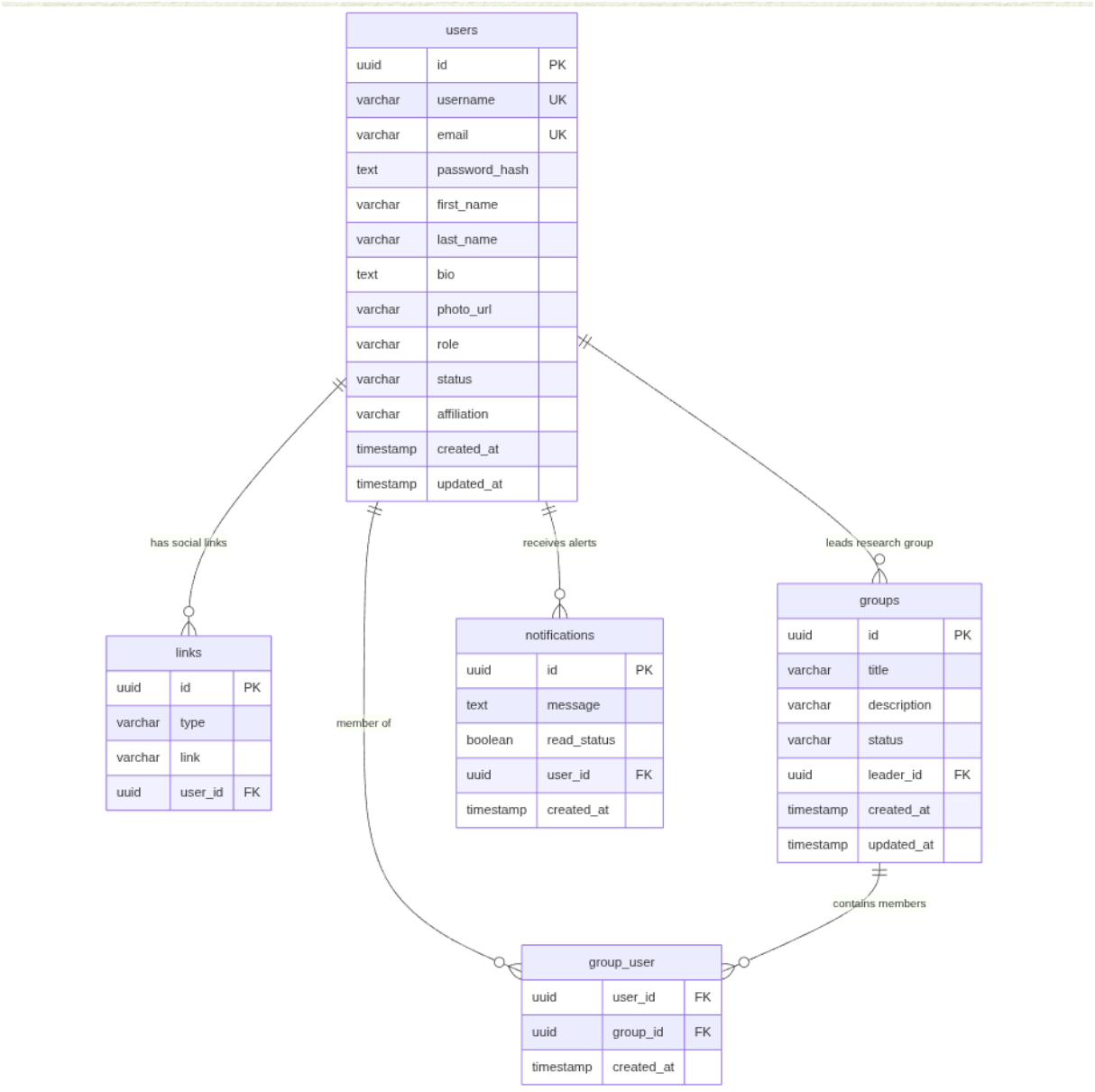


FIGURE 3.6 – Schéma entité-association — Gestion des utilisateurs et des groupes de recherche

Ce premier schéma représente une structure centrée sur la gestion des utilisateurs académiques et leurs interactions sociales. La table `users` constitue le cœur du système, regroupant

des informations détaillées sur chaque utilisateur (identifiants, données personnelles, rôle, statut et affiliation institutionnelle).

Le modèle relationnel permet la création de liens sociaux via la table `links`, l'envoi de notifications personnalisées, ainsi que la constitution de groupes de recherche collaboratifs. La table de liaison `group_user` autorise une appartenance multiple à divers groupes, tandis que le système de notifications facilite la communication entre les membres. Cette structure favorise un réseau académique dynamique et interconnecté.

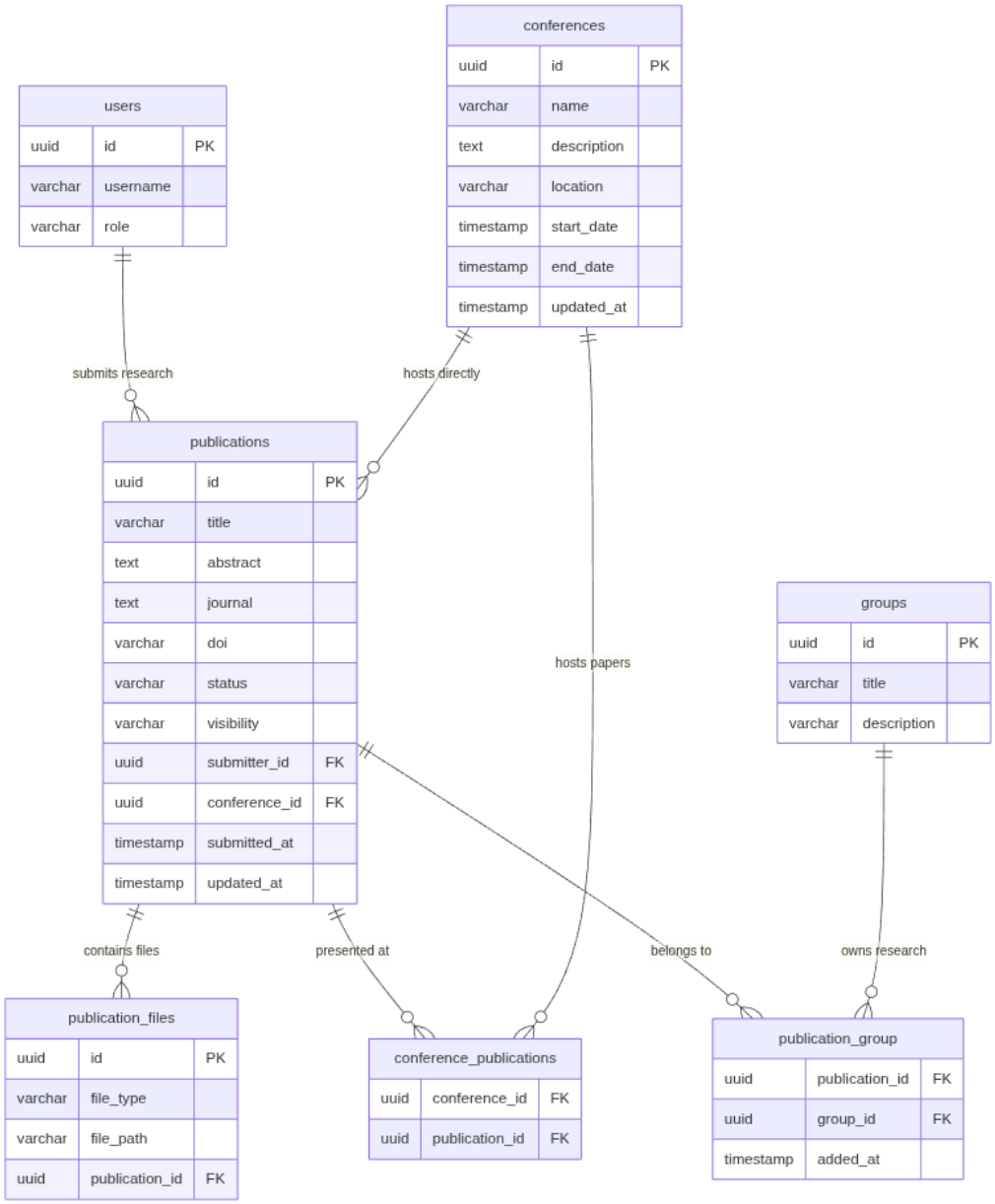


FIGURE 3.7 – Schéma entité-association — Publications scientifiques et conférences académiques

Ce second schéma illustre la gestion des publications scientifiques et des conférences aca-

démiques. Les utilisateurs peuvent soumettre des publications, lesquelles sont associées à des conférences spécifiques par l'intermédiaire de la table de liaison `conference_publications`.

Chaque publication peut inclure des fichiers annexes (articles, diapositives, etc.), gérés via la table `publication_files`. En outre, les publications peuvent être liées à des groupes de recherche, renforçant ainsi la collaboration institutionnelle. Cette organisation permet de suivre l'évolution d'un travail scientifique, depuis sa soumission jusqu'à sa présentation en conférence, tout en maintenant les liens entre chercheurs, institutions et événements académiques.

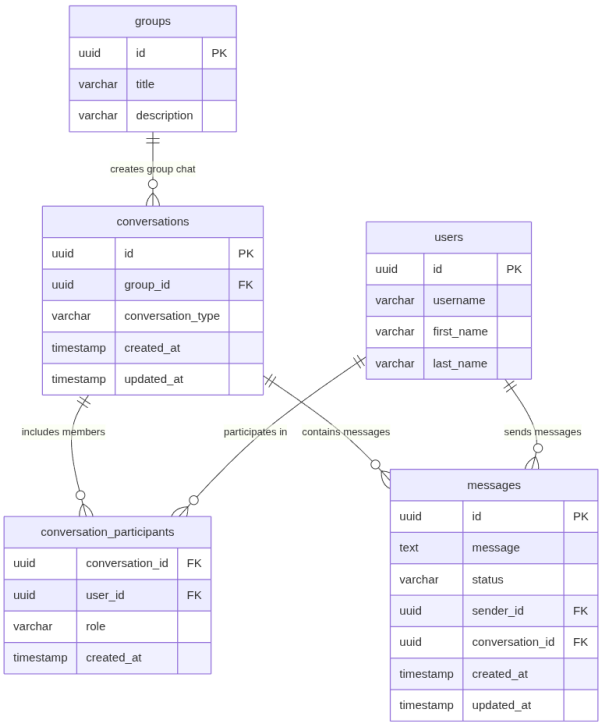


FIGURE 3.8 – Schéma entité-association — Système de communication intégré

Le dernier schéma présente l'architecture d'un système de communication conçu autour des groupes de recherche. Ces derniers peuvent créer différents types de conversations (publiques, privées ou annonces), offrant un cadre structuré pour les échanges.

Les utilisateurs participent aux discussions via la table `conversation_participants`, qui précise leur rôle dans chaque conversation. Tous les messages sont horodatés et dotés d'un statut indiquant, par exemple, leur lecture ou leur importance. Ce système assure une communication fluide et traçable, tout en favorisant le travail collaboratif au sein des équipes de recherche.

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.

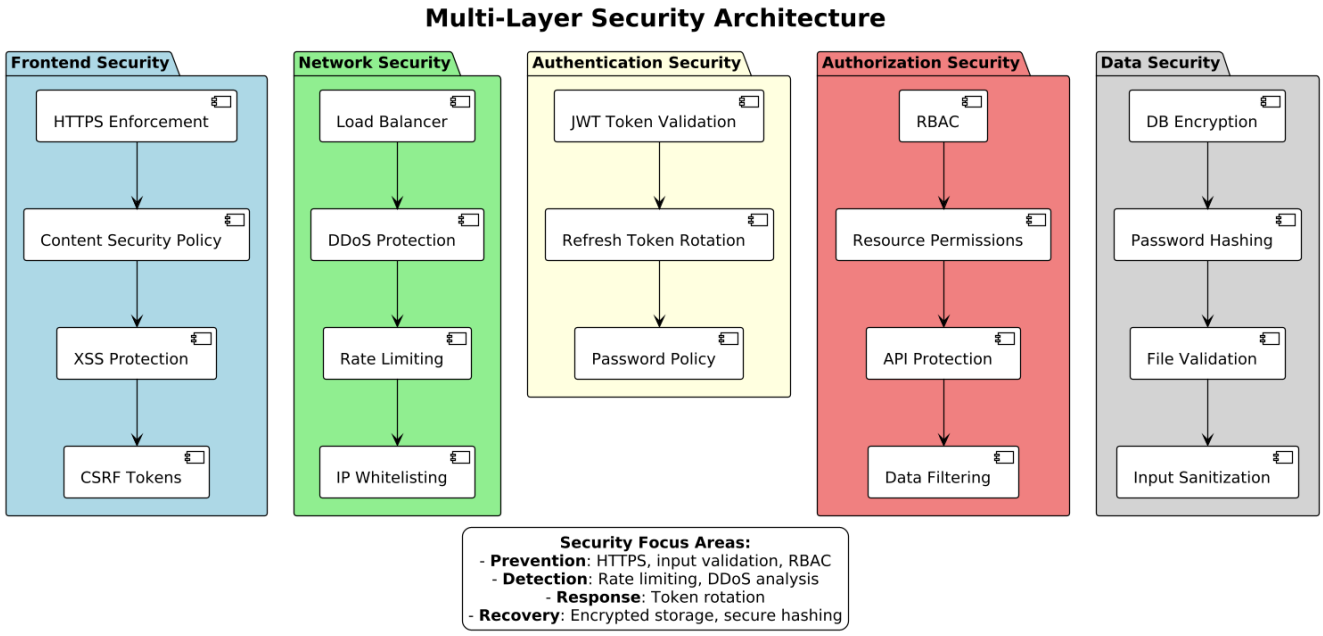


FIGURE 3.9 – Modèle de Sécurité Multi-Couches

3.6 Conclusion

Ce chapitre a présenté la conception architecturale détaillée de notre plateforme de gestion de laboratoire de recherche, articulée autour de trois piliers fondamentaux : la gestion des publications, l'authentification sécurisée et la messagerie en temps réel. L'approche modulaire, combinant microservices et monolithe optimisé, a permis d'adapter chaque composant aux exigences spécifiques de son domaine, tout en garantissant une intégration harmonieuse de l'ensemble.

Choix techniques clés

- **Service de publications** : Architecture orientée domaine (DDD) assurant une modélisation fidèle des processus métier et une forte maintenabilité.
- **Sécurité multi-couches** : Intégration de HTTPS, JWT, RBAC et chiffrement des données pour une protection à chaque niveau (frontend, réseau, backend).
- **Messagerie en temps réel** : Communication événementielle via WebSockets et Event Bus, optimisant la latence et la cohérence des états utilisateurs.

Les schémas de base de données unifiés et les diagrammes d'interaction ont permis de visualiser la collaboration entre les services et leur contribution à une expérience utilisateur fluide, respectant les exigences de performance et de scalabilité.

Cette conception robuste et documentée constitue la base du chapitre suivant, dédié à la mise en œuvre technique. Nous y détaillerons l'implémentation concrète, les défis rencontrés lors du développement, ainsi que les optimisations effectuées pour garantir robustesse et efficacité.

Chapitre 4

Mise en œuvre du site web

Ce chapitre présente la concrétisation technique du projet, détaillant les outils, méthodologies et défis rencontrés lors du développement. Nous exposons d'abord la gestion collaborative du code via GitHub, puis les stratégies de test des API (Postman) et des WebSockets (wscat). La conteneurisation avec Docker et la gestion des bases de données avec DBeaver illustrent notre approche industrialisée. Une section dédiée à la méthodologie Agile met en lumière l'organisation itérative du travail, tandis que les retours utilisateurs et captures d'écran valident l'ergonomie et la performance de l'application. Enfin, nous analysons les limites identifiées et les pistes d'optimisation pour une future montée en charge.

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'exé-

cution automatique de tests et de compilations avant toute fusion, garantissant ainsi la qualité du code.

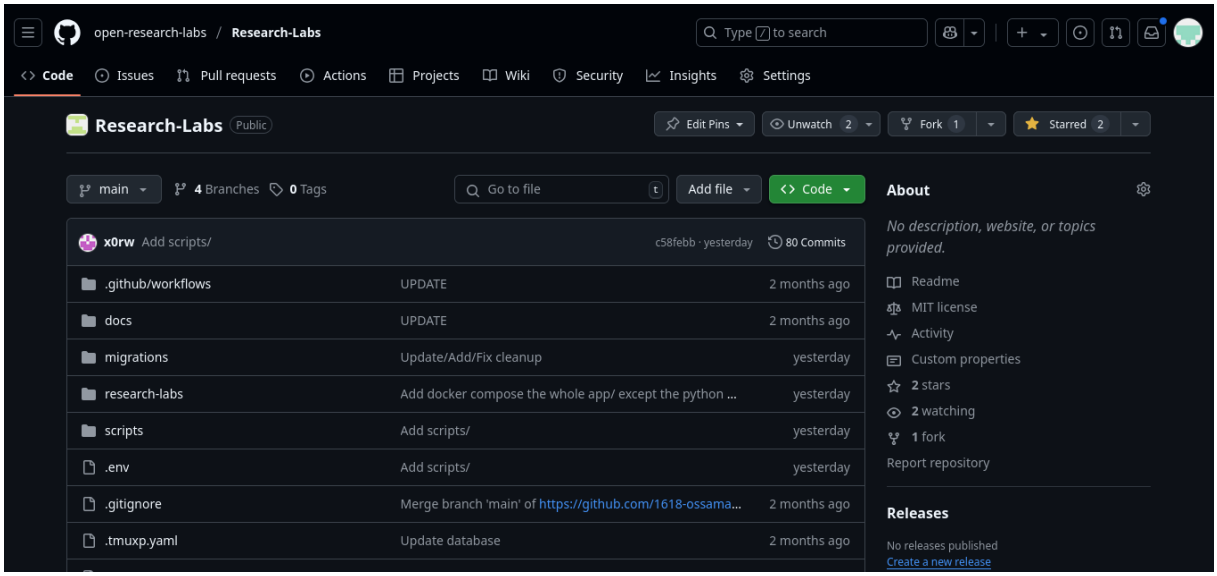


FIGURE 4.1 – GitHub

4.1.2 Organisation du dépôt et gestion des branches

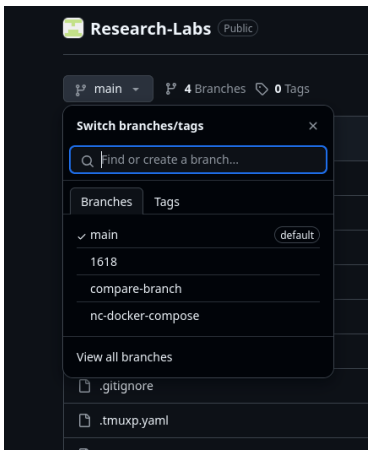


FIGURE 4.2 – Branches Git

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.

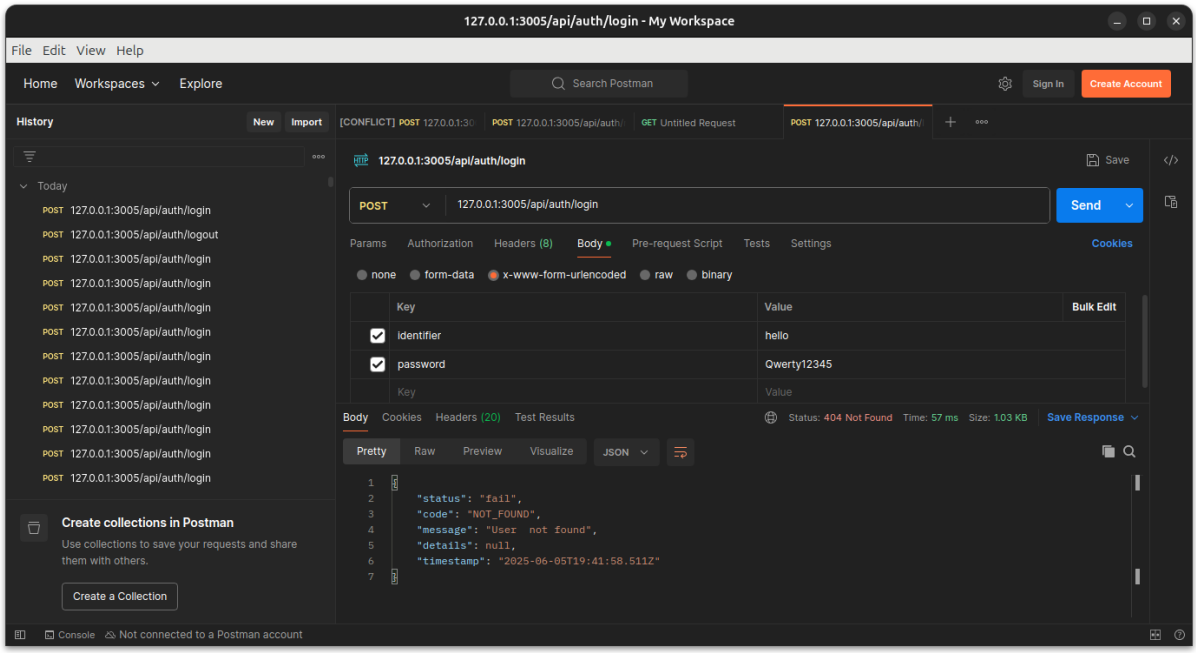


FIGURE 4.3 – Tests d'API

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.

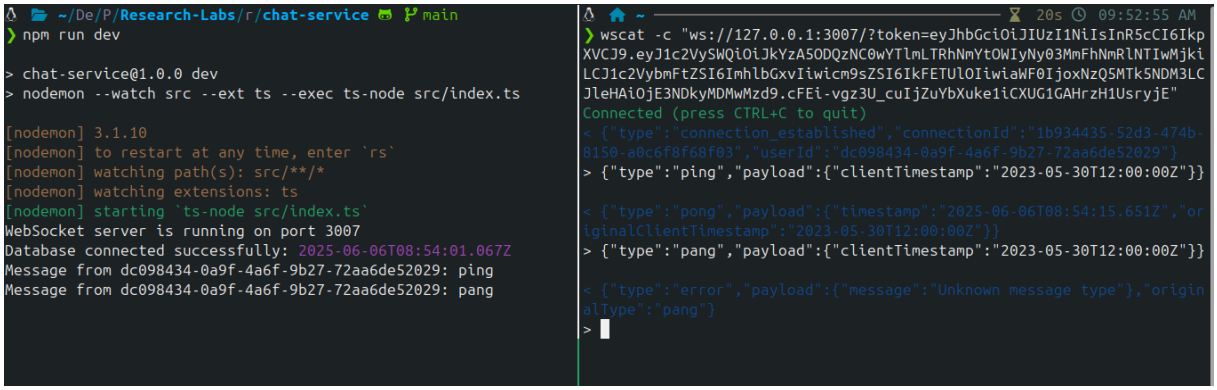


FIGURE 4.4 – Test du serveur chat par wscat

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.

4.7 Méthodologie de Développement

Approche Agile Adoptée

Le projet a suivi une méthodologie **Agile** adaptée aux contraintes académiques et à la taille réduite de l'équipe. Cette approche a été structurée en sprints courts de deux semaines, avec une répartition claire des responsabilités techniques.

Principes Appliqués :

— **Développement itératif** : livraisons fonctionnelles toutes les deux semaines avec démonstrations à l'encadrant

— **Collaboration intensive** : réunions de synchronisation régulières pour coordonner les intégrations entre services

— **Répartition par microservice** : chaque membre était responsable d'un microservice spécifique (authentification, publications, IA, chat), garantissant autonomie et spécialisation

— **Mises à jour continues** : chaque service était maintenu et amélioré indépendamment, avec intégration progressive via Docker Compose

— **Priorité au logiciel fonctionnel** : chaque sprint visait un livrable testable plutôt qu'une documentation exhaustive

Organisation des sprints :

- **Sprint 1** : Architecture, Docker, auth Node.js
- **Sprint 2** : API Rust pour publications, téléversement
- **Sprint 3** : Frontend Next.js, responsive design
- **Sprint 4** : Chat WebSocket, intégration IA, tests

Comparaison Agile vs Waterfall

Critère	Agile (choisi)	Waterfall (rejeté)
Flexibilité	Adaptable en cours de route	Coûteux à modifier
Retours	Continus	Tardifs
Livraisons	Incrémentales	Un bloc final
Documentation	Juste suffisante	Exhaustive dès le début

TABLE 4.1 – Comparaison de Méthodologie de Développement

4.7.1 Captures d'écran de l'application

Cette section présente les différentes interfaces de l'application développée, illustrant les fonctionnalités principales à travers des captures d'écran détaillées. Chaque interface a été conçue pour offrir une expérience utilisateur optimale tout en respectant les principes d'ergonomie et d'accessibilité.

Interface de connexion

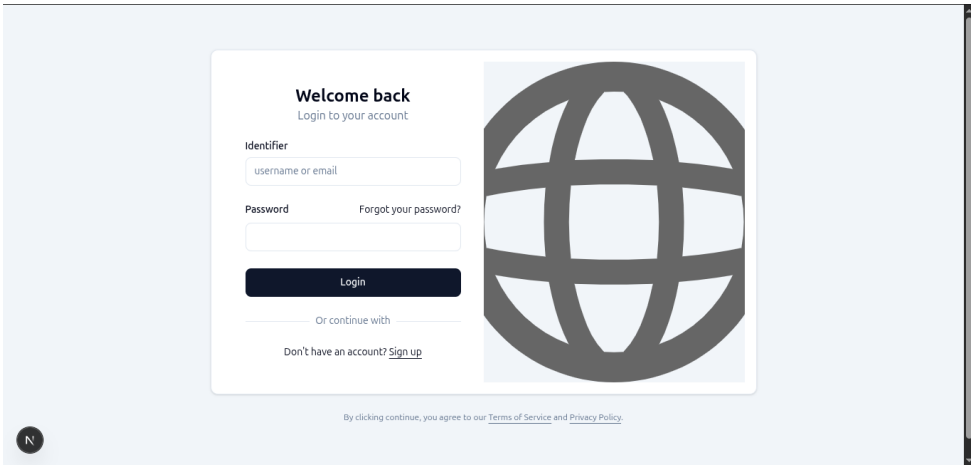


FIGURE 4.5 – Interface de connexion utilisateur

L'interface de connexion présente un design épuré et professionnel, caractérisé par :

— **Validation en temps réel** : Les champs de saisie affichent instantanément les erreurs de format ou les champs manquants

— **Messages d'erreur contextuels** : Affichage de messages clairs en cas d'échec d'authentification

— **Design responsive** : Interface adaptée aux différentes tailles d'écran

— **Sécurité visuelle** : Masquage automatique du mot de passe avec option d'affichage temporaire

Tableau de bord principal

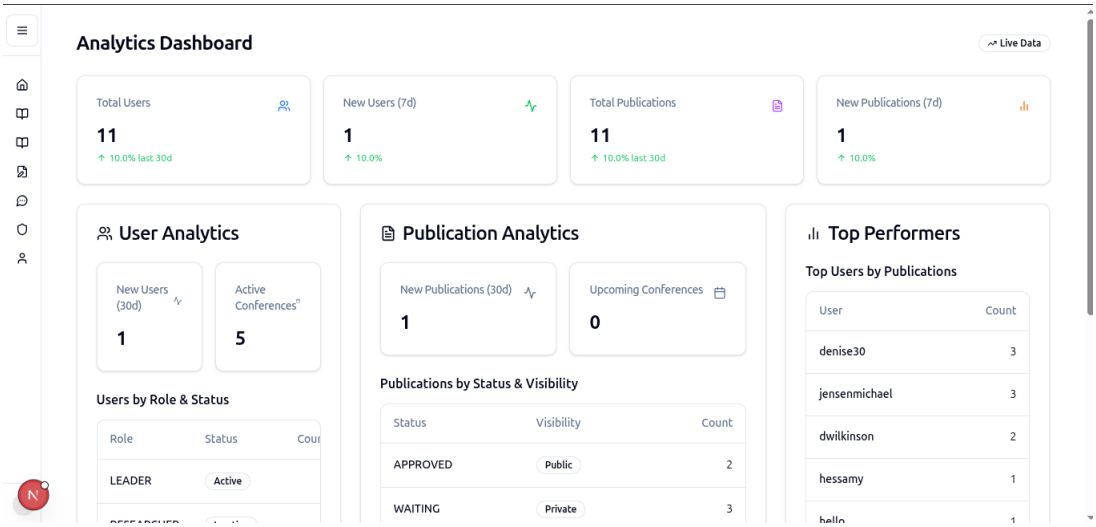


FIGURE 4.6 – Tableau de bord principal avec métriques et aperçu

Le tableau de bord offre une vue d'ensemble complète des activités du laboratoire :

— **Métriques en temps réel** : Statistiques sur les publications, conférences et membres actifs

— **Publications récentes** : Aperçu des dernières publications ajoutées avec leurs statuts

— **Notifications** : Système d'alertes pour les échéances importantes et les nouvelles activités

— **Navigation intuitive** : Menu latéral avec accès rapide à toutes les fonctionnalités

Module de gestion des publications

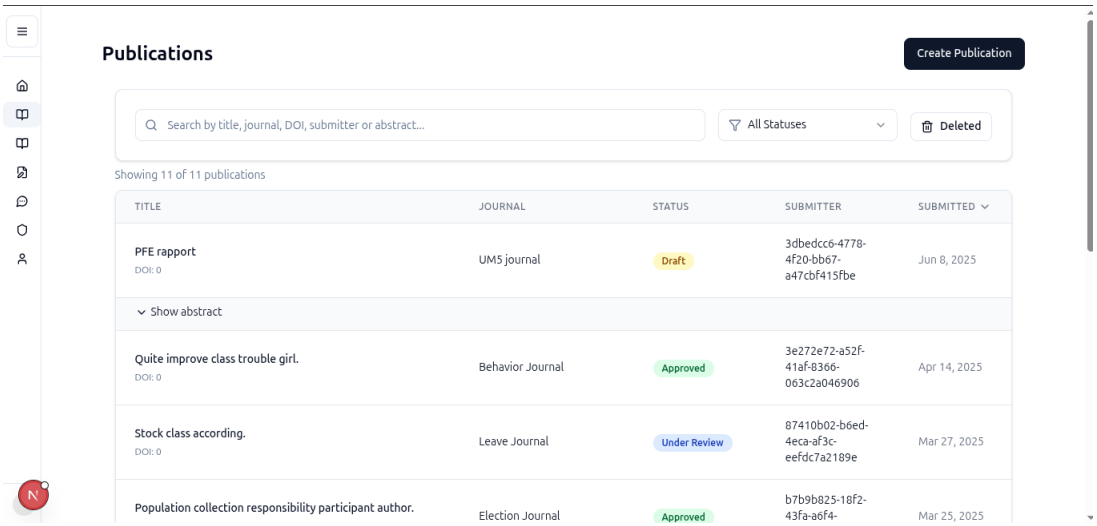


FIGURE 4.7 – Interface de gestion des publications avec filtres et pagination

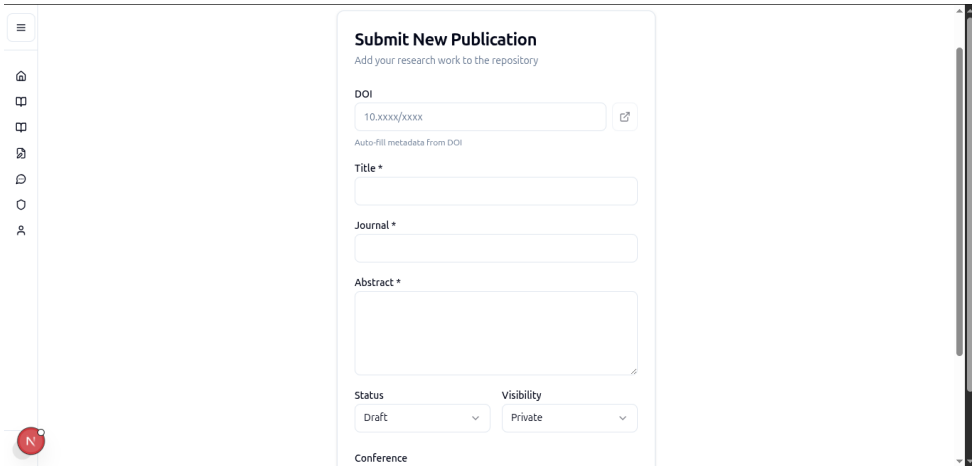


FIGURE 4.8 – Formulaire d'ajout d'une nouvelle publication

Le module publications comprend des fonctionnalités avancées :

— **Filtres dynamiques** : Recherche par auteur, date, type de publication, et mots-clés

— **Pagination intelligente** : Navigation fluide à travers de grandes collections de données

— **Formulaire d'ajout complet** : Saisie structurée avec validation des champs obligatoires

— **Upload de fichiers PDF** : Glisser-déposer avec prévisualisation et validation du format

— **Métadonnées enrichies** : Gestion des co-auteurs, affiliations, et classifications

Intégration de l'intelligence artificielle

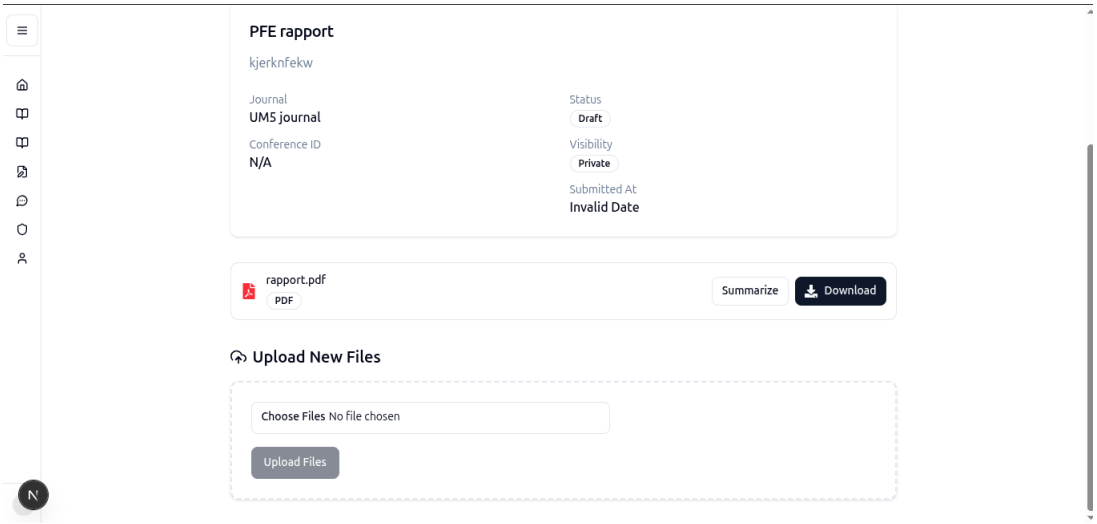


FIGURE 4.9 – Génération automatique de résumés par intelligence artificielle

L'interface d'IA démontre les capacités avancées du système :

— **Résumés automatiques** : Génération de synthèses à partir du contenu des publications PDF

— **Analyse sémantique** : Extraction des concepts clés et des contributions principales

— **Interface de révision** : Possibilité d'éditer et valider les résumés générés

— **Indicateurs de qualité** : Score de confiance et métriques sur la précision du résumé

Système de messagerie instantanée

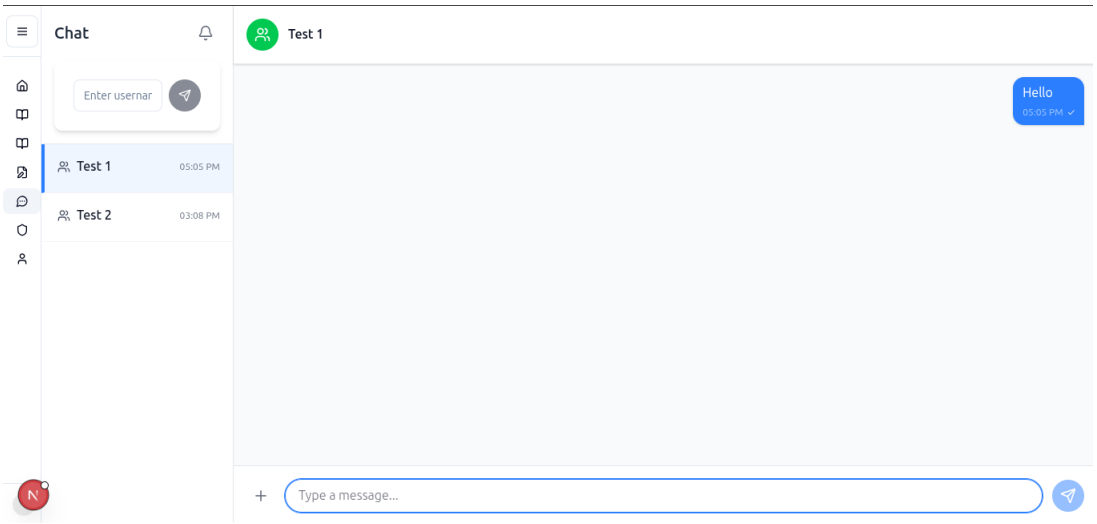


FIGURE 4.10 – Interface de chat avec liste des utilisateurs et groups

Le système de chat offre une communication fluide :

— **Groupes de messagerie** : Création de salons de discussion pour les équipes ou projets

— **Messages instantanés** : Communication bidirectionnelle via WebSockets

— **Historique persistant** : Conservation et recherche dans l'historique des conversations

— **Notifications push** : Alertes pour les nouveaux messages et mentions

— **Interface adaptative** : Design optimisé pour les conversations longues

Analyse de l'expérience utilisateur

Les captures d'écran révèlent plusieurs aspects importants de l'interface :

Cohérence visuelle L'ensemble de l'application maintient une identité graphique homogène avec une palette de couleurs professionnelle, une typographie lisible et des éléments d'interface standardisés.

Ergonomie et utilisabilité Chaque écran respecte les principes d'ergonomie web avec des zones de contenu bien délimitées, une hiérarchie visuelle claire et des actions intuitives.

Feedback utilisateur Les interfaces intègrent des mécanismes de retour (loading states, confirmations, messages d'erreur) pour guider l'utilisateur dans ses interactions.

Cette présentation visuelle confirme que l'application répond aux exigences fonctionnelles tout en offrant une expérience utilisateur moderne et professionnelle, adaptée aux besoins spécifiques d'un laboratoire de recherche.

4.8 Conclusion

Ce projet constitue une réponse concrète aux défis de gestion des laboratoires modernes. En plus de l'implémentation technique, nous avons mis l'accent sur les aspects organisationnels et les bonnes pratiques professionnelles.

Défis et solutions

- **Coordination microservices** : API Gateway, standards REST
- **Performance IA** : Redis cache + traitement async
- **Technologies multiples** : documentation, apprentissage continu

Alignement avec les objectifs secondaires

- Automatisation des tâches (upload, résumés, notifs)
- Centralisation des échanges (chat + tableau de bord)
- Sécurité (JWT, RBAC, audit)
- Évolutivité (Docker , Kubernetes prévu)

Perspectives

- Amélioration IA (modèles GPT-4)
- Dashboards analytiques (D3.js)
- CI/CD avec GitHub Actions, monitoring Prometheus
- Modules futurs : gestion de projets, revue par pairs, intégration PubMed
- Adoption : pilote universitaire, extension, commercialisation SaaS

Impact

- Compétences : Rust, TS, Python, DevOps, IA
- Méthodes pro : Agile, TDD, doc technique, outils collaboratifs

Conclusion Générale

Ce projet s'inscrit dans une volonté d'améliorer la gestion des activités de recherche à travers une plateforme centralisée.Il pose les bases d'un écosystème digital unifié pour la recherche, conciliant technologie, ergonomie et productivité scientifique. 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

[1] C. Richardson, *Microservices Patterns : With Examples in Java*. Manning Publications, 2018.

[2] E. Evans, *Domain-Driven Design : Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[3] M. Fowler, "Microservices," *martinfowler.com*, Mar. 2014. [Online]. Available : `https://martinfowler.com/articles/microservices.html`

[4] PostgreSQL Global Development Group, "PostgreSQL Documentation," *postgresql.org*, 2024. [Online]. Available : `https://www.postgresql.org/docs/`

[5] Mozilla Developer Network, "WebSocket API," *developer.mozilla.org*, 2024. [Online]. Available : `https://developer.mozilla.org/en-US/docs/Web/API/WebSocket`

[6] Docker Inc., "Docker Documentation," *docs.docker.com*, 2024. [Online]. Available : `https://docs.docker.com/`

[7] Next.js Team, "Next.js Documentation," *nextjs.org*, 2024. [Online]. Available : `https://nextjs.org/docs`

[8] Rust Foundation, "The Rust Programming Language," *doc.rust-lang.org*, 2024. [Online]. Available : `https://doc.rust-lang.org/`

[9] OWASP Foundation, "OWASP Top Ten Web Application Security Risks," *owasp.org*, 2021.

[10] T. Berners-Lee *et al.*, "Web Security," *W3C*, 2023.

[11] M. Haverbeke, *Eloquent JavaScript : A Modern Introduction to Programming*, 3rd ed. No Starch Press, 2018.

[12] S. Newman, *Building Microservices : Designing Fine-Grained Systems*, 2nd ed. O'Reilly Media, 2021.

[13] A. Verma, *Cloud Native Patterns : Designing Change-Tolerant Software*. Manning Publications, 2019.

[14] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, UC Irvine, 2000.

[15] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[16] R. Martin, *Clean Architecture : A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[17] F. Chollet, *Deep Learning with Python*, 2nd ed. Manning Publications, 2021.

[18] C. Walls, *Spring in Action*, 6th ed. Manning Publications, 2020.

[19] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[20] S. Souders, *High Performance Web Sites*. O'Reilly Media, 2007.

[21] OWASP Foundation, "OWASP Cheat Sheet Series," 2023. [Online]. Available : `https://cheatsheetseries.owasp.org/`