Polytech Marseille

Ecole d'ingénieurs – Aix-Marseille Université

Département Informatique – Semestre 9

Cloud Chat

Application de messagerie en temps réel

Rapport de réalisation de projet

Réalisé par :

BOUCHTITA Achraf – achraf.bouchtita@etu.univ-amu.fr

BELAHSEN Ilyes – ilyes.belahsen@etu.univ-amu.fr

DIANI Lina – lina.diani@etu.univ-amu.fr

BOUKHRIS Mohamed Lyazid – mohamed-lyazid.boukhris@etu.univ-amu.fr

Année universitaire : 2025 – 2026

Architecture microservices – React – NestJS – WebSocket – Docker – AWS

Table des matières

Chapitre 1

Introduction

1.1 Contexte du projet

Dans le cadre de la formation en ingénierie informatique à Polytech Marseille, le projet **Cloud Chat** a été réalisé durant le semestre 9. Ce projet s'inscrit dans une démarche d'apprentissage des architectures distribuées modernes et de la mise en œuvre d'applications web temps réel.

L'objectif principal est de concevoir et développer une application de messagerie instantanée complète, depuis la conception de l'architecture backend jusqu'à l'interface utilisateur, en passant par le déploiement cloud.

1.2 Objectifs

Les objectifs fixés pour ce projet sont les suivants :

- Développer une application de chat en temps réel avec gestion de salons (rooms).
- Implémenter une architecture en microservices conteneurisée.
- Mettre en place un système d'authentification sécurisé basé sur JWT.
- Utiliser les WebSockets pour la communication temps réel.
- Déployer l'application localement via Docker Compose et sur le cloud AWS.
- Acquérir une expérience concrète sur les technologies NestJS, React, MySQL et Terraform.

1.3 Organisation du rapport

Ce rapport présente dans un premier temps l'architecture globale du projet et les technologies employées. Ensuite, il détaille les fonctionnalités implémentées, le modèle de données, le mécanisme de communication temps réel et la stratégie de sécurité. Enfin, il aborde le volet déploiement avant de dresser un bilan des difficultés rencontrées et des solutions apportées.

Chapitre 2

Architecture du projet

2.1 Vue d'ensemble

L'application Cloud Chat repose sur une **architecture en microservices**. Chaque service est isolé dans son propre conteneur Docker, communique via des API REST internes, et possède sa propre responsabilité métier. Cette approche garantit une séparation claire des préoccupations, une scalabilité indépendante de chaque composant et une maintenabilité accrue.

L'architecture se compose de **quatre services principaux** et d'une **base de données partagée** (MySQL), elle-même segmentée en deux bases logiques distinctes.

2.2 Diagramme d'architecture

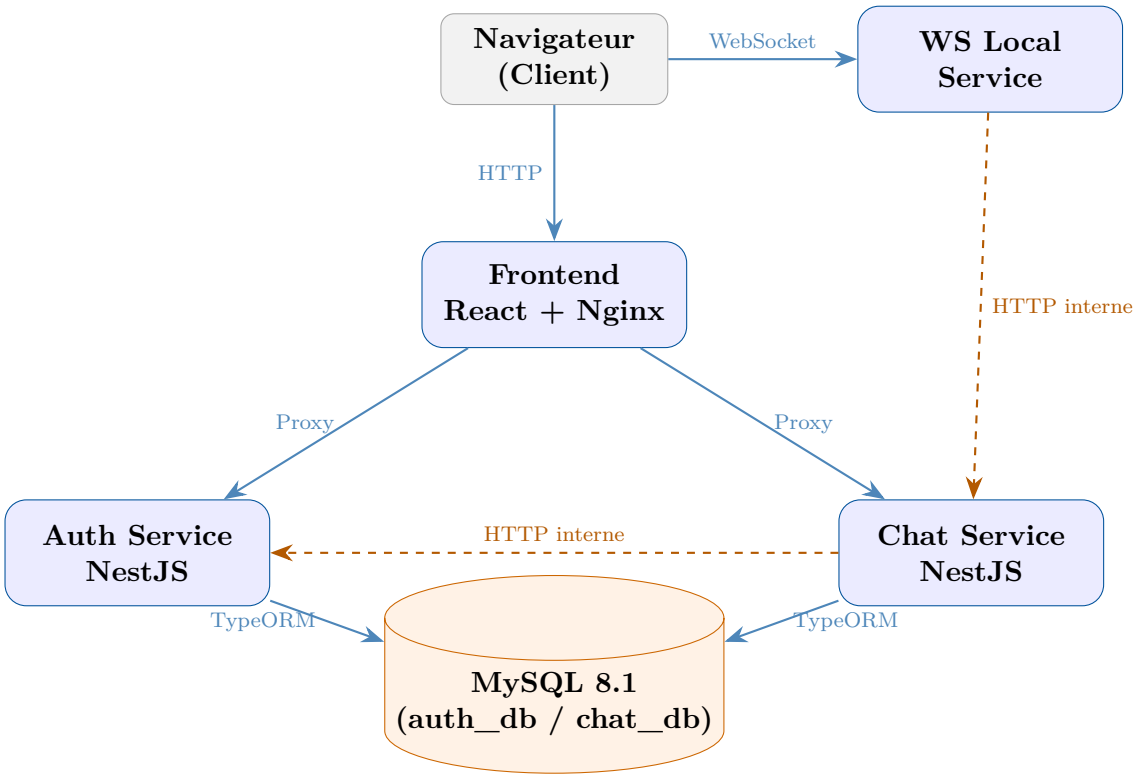

FIGURE 2.1 – Architecture globale de l'application Cloud Chat

2.3 Description des services

2.3.1 Frontend (React + Nginx)

Le frontend est une **Single Page Application (SPA)** développée avec React 19 et TypeScript, stylisée avec Tailwind CSS. En production, elle est servie par un serveur **Nginx** qui joue également le rôle de **reverse proxy**, redirigeant les requêtes API vers les services backend appropriés.

— Les routes `/auth/*` et `/users/*` sont redirigées vers l'`auth-service`.
— Les routes `/rooms/*`, `/messages/*` et `/room-members/*` sont redirigées vers le `chat-service`.
— Toutes les autres requêtes servent l'application React (fallback SPA).

2.3.2 Auth Service (NestJS)

Le service d'authentification gère l'ensemble du cycle de vie des utilisateurs :

— Inscription et connexion avec hachage des mots de passe.
— Génération et validation de **JSON Web Tokens (JWT)**.
— Système de **refresh tokens** avec rotation automatique.
— Fonctionnalité de **réinitialisation de mot de passe** par token temporaire.
— Exposition d'une API interne sécurisée par clé API pour la communication inter-services.

2.3.3 Chat Service (NestJS)

Le service de chat est responsable de toute la logique métier liée à la messagerie :

— Gestion des **rooms** (création, suppression, listing).
— Gestion des **membres** des rooms (ajout, suppression, vérification d'appartenance).
— Stockage et récupération des **messages**.
— Exposition d'endpoints internes pour le traitement des événements WebSocket.
— Validation des tokens JWT via appel au service d'authentification.

2.3.4 WS Local Service (Node.js)

Le service WebSocket est un serveur léger écrit en Node.js natif (sans framework) utilisant la bibliothèque `ws`. Il sert de **relais** entre les clients WebSocket et le chat-service :

— Accepte les connexions WebSocket des clients avec authentification par token.
— Attribue un identifiant unique (`connectionId`) à chaque connexion.

— Relaye les événements (`$connect`, `sendMessage`, `$disconnect`) vers le chat-service via HTTP interne.
— Distribue les messages sortants aux clients connectés appropriés.

2.3.5 Base de données MySQL

Une instance **MySQL 8.1** héberge deux bases de données logiquement séparées :

— `auth_db` : tables des utilisateurs, refresh tokens et tokens de réinitialisation.
— `chat_db` : tables des rooms, messages et membres de rooms.

Cette séparation respecte le principe d'isolation des données par domaine métier, chaque service n'accédant qu'à sa propre base.

Chapitre 3

Technologies utilisées

3.1 Stack technique

Couche	Technologie	Justification
Frontend	React 19	Bibliothèque UI déclarative, écosystème riche, composants réutilisables.
Routage	React Router 7	Gestion des routes côté client pour la SPA.
Style	Tailwind CSS 3	Framework CSS utilitaire permettant un développement rapide et cohérent.
Build	Vite 7	Outil de build ultra-rapide basé sur ES modules.
Langage	TypeScript	Typage statique réduisant les erreurs et améliorant la maintenabilité.
Backend	NestJS	Framework Node.js structuré (modules, injection de dépendances, décorateurs).
ORM	TypeORM	Mapping objet-relationnel avec support des migrations et des entités décorées.
Auth	JWT	Authentification stateless, adaptée aux architectures microservices.
WebSocket	ws (Node.js)	Bibliothèque WebSocket native, légère et performante.
BDD	MySQL 8.1	SGBDR mature et fiable avec support des transactions ACID.
HTTP Client	Axios	Client HTTP avec intercepteurs, gestion des erreurs et support des tokens.
Icônes	Lucide React	Bibliothèque d'icônes SVG légère et modulaire.
Conteneurs	Docker	Isolation et portabilité des services.
Orchestration	Docker Compose	Définition et lancement de l'ensemble des services en une commande.
Cloud	AWS (ECS, RDS)	Déploiement scalable en production.
IaC	Terraform	Provisionnement déclaratif de l'infrastructure cloud.

TABLE 3.1 – Technologies et outils utilisés dans le projet

3.2 Choix architecturaux

3.2.1 Pourquoi les microservices ?

L'architecture en microservices a été choisie pour plusieurs raisons :

1. **Séparation des responsabilités** : chaque service a un périmètre fonctionnel clair (authentification vs messagerie).
2. **Scalabilité indépendante** : le service de chat peut être répliqué sans toucher au service d'authentification.
3. **Résilience** : la défaillance d'un service n'entraîne pas l'arrêt complet de l'application.
4. **Apprentissage** : cette architecture permet de se confronter aux problématiques réelles de la communication inter-services, du routage et de l'orchestration.

3.2.2 Pourquoi NestJS ?

NestJS impose une structure modulaire inspirée d'Angular avec un système d'injection de dépendances. Cela favorise l'organisation du code, la testabilité et la lisibilité. Le support natif de TypeORM, de la validation par décorateurs et des guards d'authentification en fait un choix cohérent pour des API REST backend.

3.2.3 Pourquoi un service WebSocket séparé ?

Le service WebSocket est volontairement découplé du chat-service pour permettre :

— Une gestion indépendante des connexions persistantes.
— Une compatibilité avec le déploiement AWS (API Gateway WebSocket + Lambda).
— Une architecture où le chat-service reste purement REST et stateless.

Chapitre 4

Fonctionnalités

4.1 Authentification

4.1.1 Inscription et connexion

L'application propose une page unique combinant les formulaires d'inscription et de connexion. L'utilisateur peut basculer entre les deux modes via un lien interactif.

— **Inscription** : saisie du nom d'utilisateur, email et mot de passe (avec confirmation). Le mot de passe est haché côté serveur avant stockage.
— **Connexion** : saisie de l'email et du mot de passe. En cas de succès, le serveur retourne un **access token** (JWT, durée 15 minutes) et un **refresh token** (stocké en base).

4.1.2 Gestion des sessions

Le mécanisme de session repose sur la rotation des refresh tokens :

1. À la connexion, le refresh token est stocké dans le `localStorage` du navigateur.
2. Lors d'un rechargement de page, le `AuthProvider` React tente automatiquement un rafraîchissement du token.
3. Le serveur invalide l'ancien refresh token et en émet un nouveau (**rotation**).
4. À la déconnexion, le refresh token est révoqué côté serveur.

4.1.3 Réinitialisation de mot de passe

Un flux complet de réinitialisation de mot de passe est implémenté :

1. L'utilisateur saisit son email sur la page « Mot de passe oublié ».
2. Le serveur génère un token temporaire de 48 octets (haché en SHA-256 et stocké en base avec une durée d'expiration d'une heure).
3. Un message neutre est affiché côté client (anti-énumération des comptes).
4. L'utilisateur accède au lien de réinitialisation et définit un nouveau mot de passe.

Note sur l'envoi du lien de réinitialisation : aucun service d'envoi d'emails n'étant intégré dans le projet (ni en local ni sur AWS), le lien de réinitialisation est retourné

directement dans la réponse de l'API et affiché dans la **console du navigateur** (via `console.log`). L'utilisateur peut simplement ouvrir la console de son navigateur (accessible via `F12`), copier le lien affiché et le coller dans la barre d'adresse pour accéder à la page de changement de mot de passe. Ce mécanisme permet de tester et de vérifier le flux complet de réinitialisation sans disposer d'une infrastructure d'envoi d'emails.

4.2 Chat général

Le chat général est une room spéciale nommée `general` accessible à tous les utilisateurs authentifiés. Il constitue le point d'entrée principal pour la messagerie.

— Chargement automatique des messages existants via l'API REST.
— Connexion WebSocket pour la réception des nouveaux messages en temps réel.
— Affichage de l'auteur et de l'horodatage de chaque message.

4.3 Rooms privées

Les rooms privées permettent aux utilisateurs de créer des espaces de discussion restreints.

4.3.1 Création de room

Depuis la page d'accueil, l'utilisateur peut créer une room via une modale. Le nom de la room doit être unique. Le créateur devient automatiquement propriétaire (`ownerUserId`) et premier membre de la room.

4.3.2 Gestion des membres

Le propriétaire de la room dispose de privilèges spéciaux :

— **Ajouter un membre** : en saisissant le nom d'utilisateur.
— **Retirer un membre** : suppression d'un membre existant.
— **Supprimer la room** : suppression complète de la room et de ses données.

Les membres non-propriétaires peuvent uniquement consulter la liste des membres et quitter la room.

4.3.3 Navigation dans les rooms

La page « Mes Rooms » affiche toutes les rooms dont l'utilisateur est membre (hors chat général). Chaque room est représentée par une carte indiquant le nom et le créateur.

4.4 Interface utilisateur

L'interface est conçue pour être **simple, intuitive et responsive** :

- **Page d'accueil** : trois cartes d'action (Chat Général, Mes Rooms, Créer une Room).
- **Layout de chat** : barre latérale ou header avec titre, zone de messages scrollable, zone de saisie en bas.
- **Profil utilisateur** : modale accessible depuis le header pour consulter et modifier ses informations.
- **Paramètres de room** : modale dédiée pour la gestion des membres.
- **Routes protégées** : redirection automatique vers la page de connexion pour les utilisateurs non authentifiés.

Chapitre 5

Modèle de données

5.1 Base auth_db

5.1.1 Table `users`

Colonne	Type	Contrainte	Description
id	UUID	PK, auto	Identifiant unique de l'utilisateur.
email	VARCHAR	UNIQUE	Adresse email de l'utilisateur.
username	VARCHAR	UNIQUE	Nom d'utilisateur.
password_hash	VARCHAR	NOT NULL	Mot de passe haché (bcrypt).
created_at	DATETIME	auto	Date de création du compte.
updated_at	DATETIME	auto	Date de dernière modification.

Table 5.1 – Structure de la table `users`

5.1.2 Table `refresh_tokens`

Colonne	Type	Contrainte	Description
id	INT	PK, auto	Identifiant du token.
userId	VARCHAR(36)	INDEX	Référence vers l'utilisateur.
tokenHash	VARCHAR(64)	INDEX	Hash SHA-256 du refresh token.
expiresAt	DATETIME	NOT NULL	Date d'expiration du token.
revokedAt	DATETIME	NULL	Date de révocation (si applicable).
createdAt	DATETIME	auto	Date de création.

Table 5.2 – Structure de la table `refresh_tokens`

5.1.3 Table `password_reset_tokens`

Colonne	Type	Contrainte	Description
id	INT	PK, auto	Identifiant du token.
userId	VARCHAR	INDEX	Référence vers l'utilisateur.
tokenHash	VARCHAR(64)	INDEX	Hash du token de réinitialisation.
expiresAt	DATETIME	NOT NULL	Date d'expiration.
usedAt	DATETIME	NULL	Date d'utilisation.
createdAt	DATETIME	auto	Date de création.

TABLE 5.3 – Structure de la table `password_reset_tokens`

5.2 Base chat_db

5.2.1 Table `rooms`

Colonne	Type	Contrainte	Description
id	INT	PK, auto	Identifiant de la room.
ownerUserId	VARCHAR(36)	INDEX	UUID du créateur de la room.
ownerUsernameSnapshot	VARCHAR(255)	NULL	Nom du créateur (snapshot).
name	VARCHAR(255)	UNIQUE	Nom unique de la room.
createdAt	DATETIME	auto	Date de création.

TABLE 5.4 – Structure de la table `rooms`

5.2.2 Table `room_members`

Colonne	Type	Contrainte	Description
id	INT	PK, auto	Identifiant de l'enregistrement.
roomId	INT	INDEX, FK	Référence vers la room.
userId	VARCHAR(36)	INDEX	UUID du membre.
usernameSnapshot	VARCHAR(255)	NULL	Nom de l'utilisateur (snapshot).
joinedAt	DATETIME	auto	Date d'adhésion à la room.

TABLE 5.5 – Structure de la table `room_members` (contrainte UNIQUE sur roomId + userId)

5.2.3 Table `messages`

Colonne	Type	Contrainte	Description
id	INT	PK, auto	Identifiant du message.
roomId	INT	INDEX, FK	Référence vers la room (CASCADE).
userId	VARCHAR(36)	INDEX	UUID de l'auteur.
usernameSnapshot	VARCHAR(255)	NULL	Nom de l'auteur (snapshot).
type	ENUM	TEXT/FILE	Type de message.
content	TEXT	NULL	Contenu textuel du message.
fileUrl	VARCHAR(2048)	NULL	URL du fichier (si type FILE).
createdAt	DATETIME	auto	Date d'envoi.

TABLE 5.6 – Structure de la table `messages`

5.3 Stratégie de snapshot

Un point notable du modèle de données est l'utilisation de champs `usernameSnapshot`. Plutôt que de faire des jointures coûteuses entre les bases `auth_db` et `chat_db` (qui sont sur des domaines séparés), le nom d'utilisateur est copié au moment de l'action (création de room, envoi de message, adhésion). Cela garantit que le nom affiché est toujours disponible même si l'utilisateur modifie son profil ultérieurement.

Chapitre 6

Communication temps réel

6.1 Vue d'ensemble

La communication en temps réel constitue le cœur fonctionnel de l'application. Le protocole **WebSocket** a été retenu pour sa capacité à maintenir une connexion bidirectionnelle persistante entre le client et le serveur, contrairement au modèle requête-réponse classique du HTTP. Ce choix permet une latence minimale pour la distribution des messages.

L'architecture WebSocket a été conçue de manière à s'adapter à deux environnements d'exécution distincts : un **déploiement local** basé sur Docker Compose et un **déploiement cloud** sur AWS. Dans les deux cas, le `chat-service` reste purement REST et stateless, la gestion des connexions persistantes étant déléguée à une couche dédiée.

6.2 Architecture locale (Docker Compose)

En environnement local, la communication temps réel repose sur un modèle en trois couches :

1. **Client** (navigateur) : établit une connexion WebSocket native vers le `ws-local-service`.
2. **WS Local Service** (relais) : serveur Node.js léger utilisant la bibliothèque `ws`. Il maintient les connexions persistantes en mémoire et relaye les événements vers le `chat-service` via HTTP interne.
3. **Chat Service** (logique métier) : traite les événements, persiste les messages et retourne la liste des messages à distribuer.

Le `ws-local-service` maintient en mémoire une **map** associant chaque `connectionId` (UUID) à l'objet WebSocket correspondant. Cette approche est simple et performante pour un déploiement mono-instance.

6.2.1 Flux de connexion

1. Le client ouvre une connexion WebSocket avec son token JWT en paramètre d'URL.
2. Le `ws-local-service` attribue un `connectionId` unique (UUID) à la connexion.
3. Un événement `$connect` est envoyé au `chat-service` pour enregistrer la session.
4. Le `chat-service` valide le token JWT auprès de l'`auth-service`, enregistre la correspondance `connectionId` ↔ `userId` dans un store en mémoire, et associe l'utilisateur à la room.

6.2.2 Flux d'envoi de message

1. Le client envoie un JSON `{action: "sendMessage", roomId, content}`.
2. Le `ws-local-service` relaye vers `/internal/ws/send-message`.
3. Le `chat-service` persiste le message en base, identifie tous les `connectionId` des membres de la room, et retourne un tableau `outbound` contenant les données à distribuer.
4. Le `ws-local-service` envoie le message à chaque client connecté identifié.

6.2.3 Flux de déconnexion

1. À la fermeture de la connexion WebSocket, un événement `$disconnect` est envoyé.
2. Le `chat-service` supprime la session de son store en mémoire.
3. L'entrée est retirée de la map locale du `ws-local-service`.

6.3 Architecture cloud (AWS)

Sur AWS, l'architecture WebSocket locale n'est pas directement transposable : les services ECS Fargate sont stateless et ne peuvent pas maintenir de connexions WebSocket persistantes de manière fiable. L'architecture cloud repose donc sur des services managés AWS.

6.3.1 Composants

1. **API Gateway WebSocket** : service managé qui gère les connexions WebSocket à l'échelle. Il expose un endpoint `wss://` et route les événements (`$connect`, `sendMessage`, `$disconnect`) vers une fonction Lambda.
2. **AWS Lambda** : fonction serverless écrite en Node.js qui remplace le `ws-local-service`. Elle traite chaque événement WebSocket de manière indépendante.
3. **Amazon DynamoDB** : base de données NoSQL qui remplace le store en mémoire pour le stockage des sessions WebSocket. La table `ws-connections` utilise `roomId` comme clé de partition et `connectionId` comme clé de tri, avec un index secondaire global (GSI) sur `connectionId` pour les recherches inverses.
4. **Chat Service (ECS)** : identique à l'environnement local, il expose un endpoint interne `/internal/ws/lambda-send-message` appelé par la Lambda pour persister les messages.

6.3.2 Flux de connexion

1. Le client ouvre une connexion WebSocket vers l'endpoint API Gateway avec son token JWT et le `roomId` en paramètres.
2. API Gateway invoque la fonction Lambda avec l'événement `$connect`.

3. La Lambda vérifie le token JWT directement (sans appel au `auth-service`).
4. En cas de succès, la Lambda enregistre la session dans DynamoDB avec un TTL de 2 heures pour le nettoyage automatique des connexions obsolètes.

6.3.3 Flux d'envoi de message

1. Le client envoie un JSON `{action: "sendMessage", roomId, content}`.
2. API Gateway invoque la Lambda avec l'événement `sendMessage`.
3. La Lambda identifie l'expéditeur via une requête DynamoDB (index `connectionId`).
4. Le message est persisté via un appel HTTP au `chat-service`.
5. La Lambda interroge DynamoDB pour récupérer toutes les connexions de la room.
6. Pour chaque connexion, la Lambda utilise l'**API Gateway Management API** (`PostToConnectionCommand`) pour envoyer le message au client. Les connexions obsolètes (code HTTP 410) sont automatiquement supprimées de DynamoDB.

6.3.4 Flux de déconnexion

1. À la fermeture de la connexion, API Gateway invoque la Lambda avec l'événement `$disconnect`.
2. La Lambda recherche les entrées associées au `connectionId` dans DynamoDB via le GSI.
3. Toutes les entrées correspondantes sont supprimées de la table.

6.4 Comparaison des deux architectures

Aspect	**Local (Docker)**	**Cloud (AWS)**
Gestion des connexions	Serveur Node.js persistant (`ws`)	API Gateway WebSocket (managé)
Stockage des sessions	Map en mémoire (mono-instance)	DynamoDB (distribué, TTL)
Traitement des événements	`ws-local-service` (processus long)	Lambda (invocation par événement)
Distribution des messages	Envoi direct via objet WebSocket	`PostToConnectionCommand` (API)
Scalabilité	Limitée à une instance	Automatique (serverless)
Validation JWT	Via appel au `auth-service`	Directe dans la Lambda

Table 6.1 – Comparaison des architectures WebSocket locale et cloud

6.5 Sécurité de la couche WebSocket

Les mécanismes de sécurité sont appliqués dans les deux environnements :

— Le token JWT est validé à chaque connexion (via l'`auth-service` en local, directement dans la Lambda sur AWS).
— La communication entre la couche WebSocket et le `chat-service` est protégée par une **clé API interne** (`x-internal-api-key`).
— Les connexions non authentifiées sont rejetées (code `1008` en local, code HTTP `401` sur AWS).
— En local, un système de **file d'attente** (`pending queue`) gère les messages envoyés avant que la connexion ne soit pleinement établie.
— Sur AWS, le TTL DynamoDB assure le **nettoyage automatique** des sessions orphelines après 2 heures.

Chapitre 7

Authentification et sécurité

7.1 Stratégie JWT

L'authentification repose sur le standard **JSON Web Token (JWT)** selon le flux suivant :

1. **Access Token** : JWT signé avec une clé secrète, expirant après 15 minutes. Transmis dans le header `Authorization: Bearer <token>`.
2. **Refresh Token** : chaîne aléatoire dont le hash SHA-256 est stocké en base. Expire après plusieurs jours. Permet d'obtenir un nouveau couple access/refresh token.
3. **Rotation** : à chaque rafraîchissement, l'ancien refresh token est invalidé et un nouveau est émis. Ce mécanisme limite la fenêtre d'exploitation en cas de vol de token.

7.2 Protection des routes

Côté frontend, un composant `PrivateRoute` vérifie l'état d'authentification avant d'afficher la page demandée. Si l'utilisateur n'est pas connecté, il est redirigé vers `/login`.

Côté backend, chaque endpoint protégé est gardé par un **guard JWT** qui valide le token et extrait les informations utilisateur.

7.3 Communication inter-services

La communication entre le `ws-local-service` et le `chat-service` est sécurisée par une **clé API interne** partagée. Le guard `InternalApiKeyGuard` vérifie la présence et la validité du header `x-internal-api-key` sur tous les endpoints internes.

De même, le `chat-service` valide les tokens JWT des utilisateurs en appelant l'`auth-service` via un module client dédié (`AuthClientModule`).

7.4 Bonnes pratiques de sécurité

— Mots de passe hachés avec **bcrypt** (jamais stockés en clair).
— Tokens de réinitialisation hachés en SHA-256 avant stockage.
— Messages d'erreur neutres pour prévenir l'énumération des comptes.

— Séparation des bases de données par domaine.
— Variables d'environnement pour les secrets (jamais dans le code source).
— Validation côté client et côté serveur.

7.5 Axes d'amélioration en matière de sécurité

Au cours du développement, plusieurs points d'amélioration en matière de sécurité ont été identifiés. Les corrections les plus critiques ont été appliquées directement dans le code, tandis que d'autres restent des pistes pour une éventuelle mise en production.

Gestion des secrets : les variables sensibles (mots de passe, clés JWT, clés API internes) étaient initialement écrites en dur dans les fichiers de configuration (`variables.tf`, `docker-compose.yml`). Elles ont été externalisées vers des fichiers `.env` et `terraform.tfvars`, tous deux exclus du dépôt Git via le `.gitignore`.

Configuration CORS : la politique CORS, initialement ouverte à toutes les origines (`origin: true`), a été restreinte à l'URL du frontend uniquement via une variable d'environnement `FRONTEND_URL`.

Pistes non implémentées : pour une mise en production complète, il conviendrait d'ajouter un mécanisme de *rate limiting* sur les endpoints sensibles (connexion, réinitialisation de mot de passe), de protéger les endpoints utilisateur par un guard d'authentification, de désactiver la synchronisation automatique du schéma TypeORM (`DB_SYNC`) au profit de migrations contrôlées, et de stocker l'état Terraform dans un backend distant sécurisé (S3).

Chapitre 8

Déploiement

8.1 Déploiement local avec Docker Compose

Le fichier `docker-compose.yml` définit l'ensemble de l'infrastructure locale :

Service	Port	Image	Rôle
db	3306	mysql :8.1	Base de données avec healthcheck.
auth-service	5002	Build local	Service d'authentification.
chat-service	5001	Build local	Service de messagerie.
frontend	5000	Build local	SPA React + reverse proxy Nginx.
ws-local-service	4001	Build local	Relais WebSocket.

TABLE 8.1 – Services Docker Compose

Un réseau Docker bridge (`chat-network`) interconnecte tous les services. Un volume persistant (`db_data`) assure la durabilité des données MySQL. Un script d'initialisation (`01-init.sh`) crée automatiquement les bases et les utilisateurs MySQL au premier démarrage.

La synchronisation du schéma est gérée par TypeORM avec l'option `DB_SYNC=true`, qui crée automatiquement les tables à partir des entités TypeScript.

8.1.1 Build multi-étapes

Les Dockerfiles utilisent un pattern de **build multi-étapes** :

1. **Étape builder** : image Node.js pour installer les dépendances et compiler le code TypeScript.
2. **Étape production** : image légère (Alpine/Nginx) contenant uniquement les artefacts compilés.

Ce pattern réduit considérablement la taille des images finales et élimine les dépendances de développement.

8.2 Déploiement AWS

Le déploiement sur le cloud AWS constitue l'étape de mise en production de l'application. Il vise à transposer l'architecture locale conteneurisée vers une infrastructure managée, scalable et hautement disponible. L'ensemble de l'infrastructure est provisionné de manière déclarative via l'outil **Terraform** (Infrastructure as Code), garantissant la reproductibilité et la traçabilité des déploiements.

8.2.1 Approche Infrastructure as Code (IaC)

Le choix de Terraform comme outil d'IaC repose sur sa capacité à décrire l'état souhaité de l'infrastructure dans des fichiers de configuration déclaratifs (fichiers `.tf`). Terraform compare cet état souhaité avec l'état réel de l'infrastructure (stocké dans le fichier `terraform.tfstate`) et calcule automatiquement le plan de modifications à appliquer. Cette approche présente plusieurs avantages dans un contexte académique et professionnel :

— **Reproductibilité** : l'infrastructure peut être recréée à l'identique à partir des fichiers de configuration.
— **Versionnement** : les fichiers Terraform sont versionnés dans Git, permettant de tracer l'évolution de l'infrastructure.
— **Idempotence** : l'application successive du même plan produit toujours le même résultat.

8.2.2 Architecture réseau (VPC)

L'infrastructure réseau repose sur un **Virtual Private Cloud (VPC)** qui isole l'ensemble des ressources dans un réseau virtuel dédié. Ce VPC est structuré en plusieurs sous-réseaux répartis sur deux zones de disponibilité (AZ) pour assurer la résilience :

— **Sous-réseaux publics** : hébergent les composants accessibles depuis Internet (Application Load Balancers, NAT Gateway). Ils disposent d'une route vers une **Internet Gateway** pour le trafic entrant et sortant.
— **Sous-réseaux privés** : hébergent les services applicatifs (ECS Fargate) et la base de données (RDS). Ces ressources ne sont pas directement accessibles depuis Internet. L'accès sortant (pour télécharger des images Docker, par exemple) est assuré via un **NAT Gateway** situé dans un sous-réseau public.

Cette séparation réseau constitue une bonne pratique de sécurité en limitant la surface d'attaque : seuls les points d'entrée explicitement configurés (load balancers, API Gateway) sont exposés publiquement.

8.2.3 Services AWS utilisés

Service AWS	Rôle dans l'architecture
ECS (Fargate)	Exécution serverless des conteneurs Docker (auth-service, chat-service, frontend) sans gestion de serveurs.
ECR	Registre privé d'images Docker, permettant le stockage et la distribution des images vers ECS.
RDS (MySQL 8.0)	Base de données relationnelle managée remplaçant le conteneur MySQL local, avec sauvegardes automatiques.
API Gateway (HTTP)	Point d'entrée HTTP unifié avec routage vers les services ECS via des intégrations VPC Link.
API Gateway (WebSocket)	Gestion managée des connexions WebSocket avec routage par action (`$connect`, `sendMessage`, `$disconnect`).
Lambda	Fonction serverless Node.js traitant les événements WebSocket en remplacement du `ws-local-service`.
DynamoDB	Base NoSQL pour le stockage distribué des sessions WebSocket avec TTL automatique.
VPC	Réseau virtuel isolé structuré en sous-réseaux publics et privés sur deux zones de disponibilité.
CloudWatch Logs	Centralisation et consultation des logs de tous les services pour le diagnostic et le monitoring.
Cloud Map	Service de découverte DNS interne permettant la communication inter-services par nom logique.

TABLE 8.2 – Services AWS utilisés pour le déploiement cloud

8.2.4 Orchestration des conteneurs avec ECS Fargate

Les trois services applicatifs (auth-service, chat-service, frontend) sont déployés sur **Amazon ECS** avec le mode de lancement **Fargate**. Ce mode serverless dispense de provisionner et gérer des instances EC2 : AWS alloue automatiquement les ressources de calcul nécessaires à chaque conteneur.

Chaque service est défini par une **task definition** spécifiant l'image Docker (stockée dans ECR), les ressources allouées (CPU, mémoire), les variables d'environnement et la configuration réseau. Un **service ECS** assure le maintien du nombre souhaité d'instances (tâches) en cours d'exécution et gère le redéploiement lors des mises à jour.

La résolution DNS interne est assurée par **AWS Cloud Map** (Service Discovery), permettant aux services de communiquer entre eux par nom logique (par exemple, `auth-service.chat-app.loc` plutôt que par adresse IP.

8.2.5 Base de données managée (RDS)

En remplacement du conteneur MySQL local, une instance **Amazon RDS** (MySQL 8.0) est provisionnée dans les sous-réseaux privés du VPC. RDS offre des fonctionnalités de gestion automatisée (sauvegardes, mises à jour de sécurité, monitoring) qui simplifient l'exploitation en production. L'instance est accessible uniquement depuis les services ECS du même VPC, grâce aux règles du **security group** associé.

8.2.6 Processus de déploiement

Un script Bash (`deploy-aws.sh`) automatise l'ensemble du processus de déploiement continu :

1. **Authentification** auprès du registre ECR via les credentials AWS.
2. **Récupération** des URLs des registres Docker via les outputs Terraform.
3. **Build et push** des images Docker pour chaque service vers ECR.
4. **Redéploiement** des services ECS via la commande `force-new-deployment`, qui déclenche le remplacement progressif des tâches en cours par de nouvelles tâches utilisant les images mises à jour.
5. **Vérification de santé** par une boucle de polling qui attend que tous les services soient opérationnels.
6. **Affichage** des URLs d'accès (API Gateway HTTP, API Gateway WebSocket, frontend) en fin de déploiement.

Ce script permet de déployer une nouvelle version de l'application en une seule commande, réduisant le risque d'erreur humaine et assurant la cohérence du déploiement.

Chapitre 9

Difficultés rencontrées et solutions

Ce chapitre présente les principales difficultés techniques rencontrées au cours du développement du projet, accompagnées des solutions mises en œuvre pour y remédier.

9.1 Architecture WebSocket et microservices

Difficulté

L'un des défis majeurs a été de concevoir une architecture WebSocket compatible avec le paradigme des microservices. Les WebSockets maintiennent des connexions persistantes et stateful, ce qui est en contradiction avec le principe stateless des services REST. De plus, le `chat-service` NestJS devait rester purement REST pour être déployable sur ECS sans gestion d'état.

Solution

La solution adoptée a été de créer un **service WebSocket dédié** (`ws-local-service`) qui agit comme un relais. Ce service maintient les connexions persistantes avec les clients et communique avec le `chat-service` via HTTP interne. Le `chat-service` retourne un tableau `outbound` indiquant quels messages envoyer à quels `connectionId`. Cette architecture permet au chat-service de rester stateless tout en supportant la communication temps réel. Pour le déploiement AWS, cette même logique a été adaptée via API Gateway WebSocket et Lambda.

9.2 Communication inter-services et validation des tokens

Difficulté

Les deux services backend (auth et chat) possèdent chacun leur propre base de données et ne partagent aucun modèle. Le `chat-service` doit pourtant valider les tokens JWT des utilisateurs et résoudre les noms d'utilisateur, informations détenues uniquement par l'`auth-service`.

Solution

Un module client dédié (`AuthClientModule`) a été créé dans le `chat-service`. Ce module effectue des appels HTTP vers l'`auth-service` pour valider les tokens et récupérer les informations utilisateur. La communication est sécurisée par une clé API interne. De plus, une stratégie de **snapshots** des noms d'utilisateur a été mise en place : le nom est copié dans les entités du chat-service au moment de l'action, évitant ainsi les jointures inter-bases systématiques.

9.3 Gestion des tokens JWT et persistance de session

Difficulté

La gestion de la session utilisateur côté frontend posait plusieurs problèmes : le token JWT expire après 15 minutes, le rechargement de page perd l'état React, et le stockage du token doit être sécurisé contre les attaques XSS.

Solution

Un système de **refresh token avec rotation** a été implémenté. Le refresh token est stocké dans le `localStorage` et le composant `AuthProvider` tente automatiquement un rafraîchissement au chargement de la page. La rotation des tokens (invalidation de l'ancien à chaque rafraîchissement) limite la fenêtre d'exploitation en cas de compromission. L'access token n'est jamais persisté, uniquement conservé en mémoire React.

9.4 Configuration du reverse proxy Nginx

Difficulté

Avec une architecture multi-services, le frontend doit communiquer avec plusieurs backends (auth-service sur le port 5002, chat-service sur le port 5001). En environnement Docker, les noms de service ne sont pas accessibles depuis le navigateur du client. De plus, le fonctionnement en SPA nécessite que toutes les routes non-API retournent `index.html`.

Solution

La configuration Nginx du frontend a été conçue pour servir de **reverse proxy unifié**. Chaque préfixe d'URL est redirigé vers le service approprié via `proxy_pass` en utilisant les noms de service Docker (résolution DNS interne). La directive `try_files $uri $uri/ /index.html` assure le fallback SPA. L'API frontend utilise un `baseURL` vide (same-origin), rendant la configuration transparente.

9.5 Initialisation de la base de données

Difficulté

Au premier lancement, deux bases de données distinctes doivent être créées avec leurs utilisateurs respectifs et les permissions appropriées. De plus, les services backend doivent attendre que MySQL soit pleinement opérationnel avant de tenter de se connecter, sous peine de crasher au démarrage.

Solution

Un script d'initialisation shell (`01-init.sh`) est monté dans le répertoire `docker-entrypoint-initdb.d` de l'image MySQL. Ce script crée les bases, les utilisateurs et attribue les privilèges de manière automatique. Les variables d'environnement sont injectées via le fichier `.env`. Pour la synchronisation du démarrage, Docker Compose utilise des **healthchecks** avec la commande `mysqladmin ping` et une directive `depends_on` avec `condition: service_healthy`.

9.6 Adaptation locale vs cloud

Difficulté

L'architecture locale (Docker Compose) et l'architecture cloud (AWS) diffèrent fondamentalement sur plusieurs points : le WebSocket est géré par un serveur Node.js en local mais par API Gateway + Lambda sur AWS ; la base MySQL est un conteneur local mais RDS en cloud ; le stockage des sessions WS est en mémoire en local mais sur DynamoDB en cloud.

Solution

Le code a été conçu pour être **agnostique vis-à-vis de l'infrastructure** autant que possible. Le `chat-service` expose les mêmes endpoints internes (`/internal/ws/*`) que le service local ou les fonctions Lambda invoquent. Les variables d'environnement permettent de configurer les URLs et les secrets sans modifier le code source. Pour les sessions WS, un store abstrait (`WsSessionStore`) encapsule la logique de stockage, permettant de basculer entre mémoire locale et DynamoDB.

9.7 Gestion du CORS et des variables d'environnement au runtime

Difficulté

Les applications React sont compilées en fichiers statiques lors du build Docker. Les variables d'environnement (comme l'URL de l'API ou du WebSocket) sont donc figées au moment du build, ce qui pose problème lorsque l'application doit être déployée sur différents environnements (local, staging, production) sans reconstruire l'image.

Solution

Un script `docker-entrypoint.sh` génère dynamiquement un fichier `env.js` au démarrage du conteneur. Ce fichier injecte les variables d'environnement dans l'objet global `window.__ENV__`. Le code React lit ses configurations depuis cet objet plutôt que depuis `process.env`, permettant une configuration au runtime sans rebuild.

Chapitre 10

Conclusion

10.1 Bilan

Le projet Cloud Chat a permis de mener à bien la conception et le développement d'une application de messagerie en temps réel complète, en suivant une approche professionnelle basée sur les microservices.

Les objectifs initiaux ont été atteints :

- Une application fonctionnelle avec authentification, chat général et rooms privées.
- Une architecture modulaire et conteneurisée.
- Un système de communication temps réel via WebSocket.
- Un déploiement local opérationnel via Docker Compose.
- Une infrastructure cloud définie en Terraform et déployable sur AWS.

10.2 Compétences acquises

Ce projet a permis l'acquisition et le renforcement de nombreuses compétences :

- **Architecture logicielle** : conception d'architectures microservices, gestion de la communication inter-services, patterns de sécurité.
- **Développement backend** : NestJS, TypeORM, JWT, guards, modules.
- **Développement frontend** : React 19, TypeScript, Tailwind CSS, gestion d'état avec Context API.
- **Temps réel** : protocole WebSocket, gestion des connexions persistantes, pattern relay.
- **DevOps** : Docker, Docker Compose, builds multi-étapes, Nginx, reverse proxy.
- **Cloud** : AWS ECS, RDS, API Gateway, Lambda, Terraform, CI/CD.

10.3 Perspectives d'amélioration

Plusieurs axes d'amélioration pourraient être envisagés :

- Ajout d'indicateurs de présence (utilisateurs en ligne).
- Support de l'envoi de fichiers et d'images.

— Notifications push pour les messages non lus.
— Chiffrement de bout en bout des messages.
— Mise en place de tests unitaires et d'intégration automatisés.
— Interface d'administration pour la gestion des utilisateurs et la modération.

Ce projet constitue une expérience riche et formatrice, confrontant aux réalités du développement d'applications distribuées modernes. Les choix architecturaux et les difficultés surmontées représentent un socle solide pour les projets à venir en contexte professionnel.