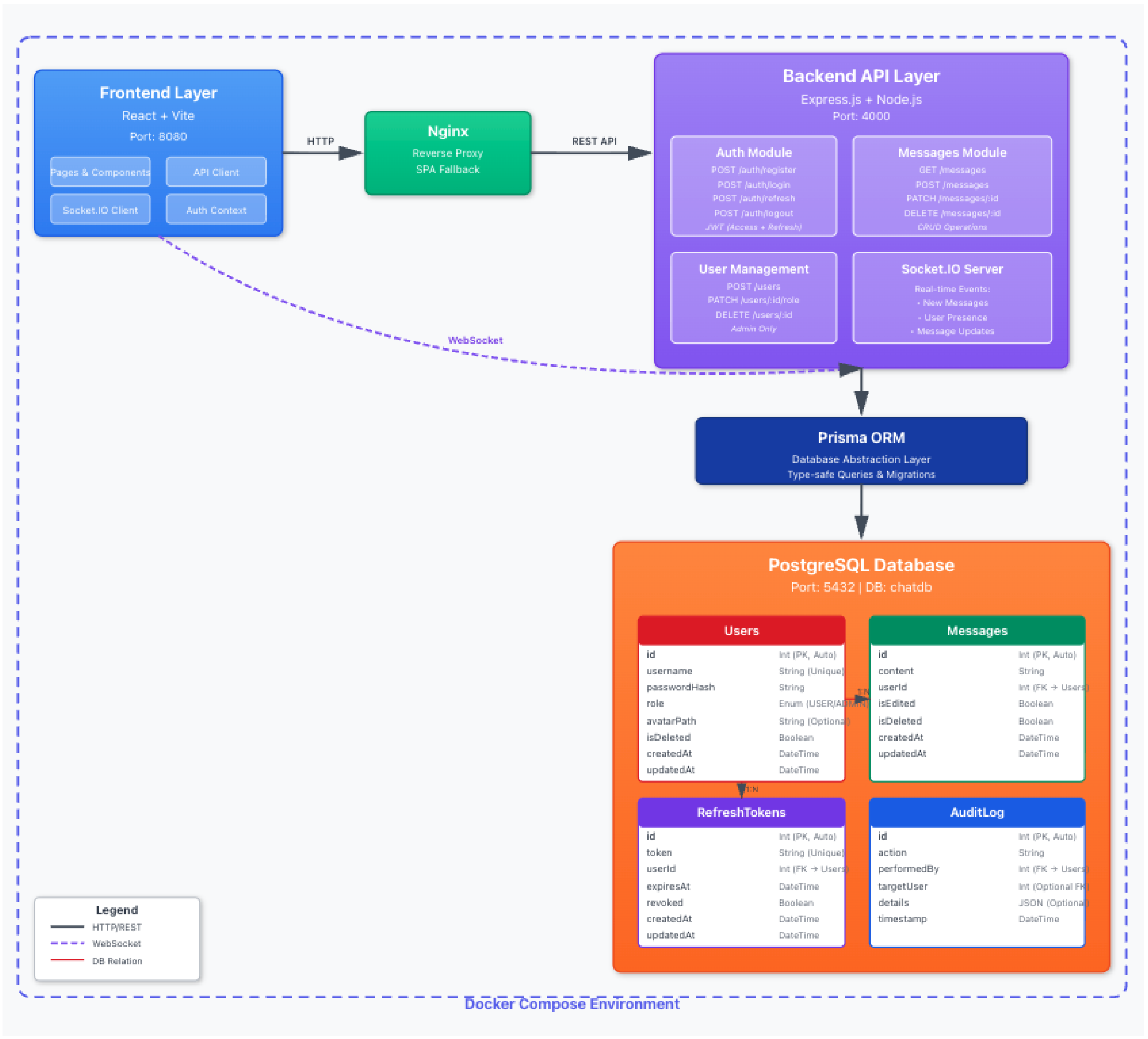
Chat Room - System Architecture Document

Executive Summary

This document provides a comprehensive architectural overview of the Chat Room application, a full-stack real-time messaging platform. The system implements a modern three-tier architecture utilizing Docker containerization, JWT-based authentication with refresh tokens, WebSocket communication via Socket.IO, and PostgreSQL database with Prisma ORM.



1. System Overview

1.1 Purpose

The Chat Room application is a real-time messaging platform enabling users to communicate in a shared chat environment with user authentication, message persistence, real-time updates, and administrative controls.

1.2 Core Features

- User Management: Registration, login, role-based access control (USER/ADMIN)
- Real-Time Messaging: Instant message delivery using WebSocket
- Message Persistence: Full CRUD operations stored in PostgreSQL
- Administrative Controls: User management, message moderation, audit logging
- Security: JWT-based authentication with access and refresh tokens
- Presence System: Real-time user online/offline status tracking

1.3 System Scope

- Frontend: Single-page React application
- Backend: REST API with WebSocket support
- Database: PostgreSQL relational database
- Deployment: using Docker Compose

2. Technology Stack

2.1 Backend Stack

- Node.js & Express.js
- Socket.IO

2.2 Frontend Stack

- React
- TypeScript

2.3 Database

- PostgreSQL
- Prisma ORM

3. System Architecture

3.1 High-Level Architecture

Three Layers:

Client Layer (Port 8080):

- React SPA with components for pages, admin, and audit
- State management using React Context and Hooks
- API client using Axios
- Socket.IO client for real-time updates

Application Layer (Port 4000):

- Node.js + Express.js server
- Middleware: CORS, Helmet security headers, JWT verification, error handling
- Route handlers: /auth, /messages, /users, admin/audit
- Socket.IO server for real-time messaging and presence

Data Layer (Port 5432):

- PostgreSQL database
- Tables: users, messages, refresh_tokens, audit_logs
- Managed via Prisma ORM

3.2 Request Flow

Message write flow: Client sends message via REST API → Server persists message → Server broadcasts message events via Socket.IO → Clients update UI.

4. Component Design

4.1 Backend Components

Authentication Module

- Handles registration, login, token generation and validation
- Dual-token pattern: Short-lived access tokens (15 min), long-lived refresh tokens (14 days)
- Access tokens stored in memory, refresh tokens in httpOnly cookies
- argon2 hashing

Message Module

- RESTful CRUD operations for messages
- Real-time broadcasting via Socket.IO
- Authorization checks ensure users can only modify own messages

User Management Module

- USER and ADMIN roles
- Admin auto-creation via environment variables
- Soft deletion for users
- Comprehensive audit logging

Socket.IO Integration

- Access token-based authentication during connection handshake.
- Server broadcasts real-time events:
- Presence system tracks online/offline status
- Message creation is handled via REST API and broadcast to clients using Socket.IO.

4.2 Frontend Components

Component Structure

- Pages: Login&Register Page, ChatPage, AdminPage, AuditPage

State Management

- React Context API with useState
- No useReducer or external state management library is used.

API Client

- Axios-based HTTP client without interceptors.
- Session refresh is performed explicitly during application bootstrap.

5. Security Architecture

5.1 Authentication Security

Token Protection:

- Access tokens: 15-minute lifetime, stored in memory, sent in Authorization header
- Refresh tokens: 14-day(by default)lifetime, httpOnly cookie, Secure flag, SameSite=Strict

CSRF Protection:

- The `/auth/refresh` endpoint is protected using a Double Submit Cookie CSRF pattern due to the use of cookie-based refresh tokens.
- A CSRF token is stored in a cookie and must be sent in a custom request header, with the server validating that both values match.

Password Security:

- argon2 hashing
- Minimum 8 characters recommended

5.2 Authorization Security

Middleware-Based Protection:

- Authentication middleware verifies JWT
- Authorization middleware checks user roles
- Resource-level checks verify ownership
- Rate limiting is applied specifically to authentication endpoints such as /auth/login and /auth/refresh **Role Permissions** to mitigate brute-force attacks
- USER: Send/edit/delete own messages, view all messages
- ADMIN: All USER permissions plus user management, delete any message, view audit logs

5.3 Network Security

CORS Configuration:

- Strict origin validation (only frontend allowed)
- Credentials enabled for cookie sharing
- Limited HTTP methods

Security Headers:

- Helmet.js applies industry-standard headers
- Content-Security-Policy, X-Frame-Options, X-Content-Type-Options
- Strict-Transport-Security for HTTPS enforcement

5.4 Database Security

SQL Injection Prevention:

- Prisma ORM uses parameterized queries exclusively
- No string concatenation of user input
- TypeScript type safety

Least Privilege:

- In containerized and local development environments, the database runs with a superuser role (e.g., postgres) for simplicity.

- Least-privilege database roles (SELECT / INSERT / UPDATE only) are an architectural recommendation for production deployments.

6. Data Architecture

6.1 Database Schema

User Table:

- id: UUID primary key (prevents enumeration)
- username: Unique, indexed
- password: argon2 hashing
- role: ENUM (USER, ADMIN)
- avatarUrl: Optional
- createdAt, deletedAt ,lastLoginAt: Timestamps
- Relations: messages, refreshTokens, auditLogs
- Indexes: username

Message Table:

- id: UUID primary key
- content: TEXT (unlimited length)
- authorId: Foreign key to User
- deletedAt: Soft delete timestamp
- createdAt, Timestamp
- updatedAt: Timestamp (optional)
- Indexes: createdAt, authorId + createdAt

RefreshToken Table:

- id: UUID primary key
- tokenHash: Hashed refresh token
- userId: Foreign key to User
- expiresAt: Expiration timestamp
- revokedAt: Manual revocation timestamp

- replacedByTokenId: Reference to replacement token (optional)
- createdAt: Timestamp
- Indexes: userId, expiresAt
-

AuditLog Table:

- id: UUID primary key
- action: Action type string
- entityType: Affected entity type
- entityId: Affected entity ID
- actorUserId: Acting user (nullable)
- before: JSON (previous state, optional)
- after: JSON (new state, optional)
- metadata: JSON for additional context
- createdAt: Timestamp
- Indexes: createdAt, actorUserId + createdAt, action + createdAt

6.2 Entity Relationships

- User → Messages: One-to-Many
- User → RefreshTokens: One-to-Many
- User → AuditLogs: One-to-Many

Foreign Key Behaviors:

- Messages: Logical cleanup handled via soft delete (application-level)
- RefreshTokens: Logical cleanup handled via application logic
- Audit logs preserve actorUserId references, as users are soft-deleted and no physical deletion is performed.

6.3 Data Access Patterns

Common Read Queries:

- Fetch recent messages (90% of reads): Uses createdAt index with application-level filtering on deletedAt.
- User authentication: Uses username unique index

Token validation:

- Refresh token lookup is performed using userId and expiration constraints.
- Audit log retrieval: Uses timestamp index

Write Patterns:

- Message creation: Message persistence and audit logging are performed as separate operations

- User registration: creates user and logs action
- Token management: Refresh token issuance and rotation are handled as separate operations (create/revoke/replace), and audit logging is written in a separate call.

Note: The least-privilege database access model described in this section applies to production deployments.In local and containerized development environments, the database may run with a superuser role (e.g., postgres) for simplicity.

Soft Delete & Referential Integrity:

- The application follows a soft delete strategy using a deletedAt timestamp column.
- The primary application database user operates under a least-privilege model and is restricted to `SELECT`, `INSERT`, and `UPDATE` permissions only, with no `DELETE` access.
- As a result, foreign key `ON DELETE CASCADE` behavior is not relied upon during normal application flow.
- All logical deletions are handled explicitly at the application layer via `UPDATE` operations.
- Physical deletion and cascade behavior may be performed only by a privileged administrative or maintenance database role, outside of the regular application runtime, for cleanup or compliance purposes.

7. Real-Time Communication

7.1 Socket.IO Architecture

Connection Lifecycle:

- Client establishes a WebSocket connection using an access token
- Server authenticates the connection during the handshake.
- Presence state is updated and broadcast to connected clients.
- On disconnect, presence state is updated accordingly.

Event Types:

- message: create
- message: update
- message: delete
- presence: update

Presence System:

- Tracks connected users in memory
- Broadcasts presence updates to all clients.
- Provides real-time online/offline status.

7.2 Message Broadcasting

Flow:

1. Client sends message via REST api
2. Server validates authentication and persists the message using Prisma.
3. Server broadcasts message events via Socket.IO.
4. Clients update UI in real-time
5. Audit log entry is created.

Error Handling:

- Invalid messages rejected with error event
- Disconnected clients automatically reconnect

8. Deployment Architecture

8.1 Docker Compose Setup

Three Containers:

Frontend Container:

- Nginx serving React build
- Port 8080 exposed
- Depends on backend availability

Backend Container:

- Node.js application
- Port 4000 exposed
- Environment variables for configuration

Database Container:

- PostgreSQL
- Port 5432 (internal only
- **8.2 Environment Configuration**

Backend Environment Variables:

- DATABASE_URL: PostgreSQL connection string
- JWT_ACCESS_SECRET: Secret key for token signing
- JWT_REFRESH_SECRET: Secret for refresh tokens
- CORS_ORIGIN: Allowed frontend origin
- INITIAL_ADMIN_USERNAME, INITIAL_ADMIN_PASSWORD: Initial admin credentials
- NODE_ENV: production/development

Frontend Environment Variables:

- VITE_API_BASE_URL: Backend API endpoint
- VITE_SOCKET_URL: Socket.IO server endpoint

8.3 Deployment Process

1. Build Docker images for frontend and backend
2. Start PostgreSQL container
3. Run Prisma migrations to set up schema
4. Start backend container
5. Create initial admin user if configured
6. Start frontend container
7. Verify all services healthy

9. API Design

9.1 Authentication Endpoints

POST /auth/register

- Creates new user account
- Returns the created user entity
- Logs registration in audit table
- Authentication tokens are issued only during login.

POST /auth/login

- Validates credentials
- Returns access token and sets refresh token cookie
- Logs login event

POST /auth/refresh

- Validates refresh token from cookie
- Returns new access token
- Extends session

POST /auth/logout

- Revokes refresh token
- Clears cookie
- Logs logout event

9.2 Message Endpoints

GET /messages

- Returns recent non-deleted messages
- Includes user information
- Ordered by createdAt ascending
- Protected endpoint (authentication required)

POST /messages

- Creates new message
- Requires authentication
- Broadcasts via Socket.IO
- Returns created message

PUT /messages/:id

- Updates message content
- Requires ownership or admin role
- Broadcasts update via Socket.IO
- Logs modification

DELETE /messages/:id

- Soft deletes message (sets deletedAt)
- Requires ownership or admin role
- Broadcasts deletion via Socket.IO
- Logs action

9.3 User Management Endpoints

GET /users

- Lists active users (deletedAt = NULL) for authenticated users
- Requires authentication

POST /users

- Creates a new user account (admin operation).
- Requires admin role.
- Returns the created user entity.

PUT /users/:id/role

- Changes user role
- Requires admin role
- Logs role change

DELETE /users/:id

- Soft deletes user
- Requires admin role
- Associated refresh tokens are invalidated or soft-handled at the application layer
- Logs deletion

9.4 Audit Endpoints (Admin Only)

GET /admin/audit

- Returns audit logs
- Supports filtering by action, user, date range
- Ordered by timestamp descending

10. Frontend Architecture

10.1 Page Components

LoginPage:

- Login and registration forms
- Client-side validation
- Error message display
- Redirects to chat on success

ChatPage:

- Message input with send button
- Online user list sidebar
- Real-time updates via Socket.IO
- Edit/delete buttons for own messages

AdminPage:

- User management table
- Role change controls
- User deletion

10.2 Real-Time Updates

Socket.IO Client:

- Connects with JWT token on mount
- Listens for server events
- Updates local state on events
- Handles reconnection automatically
- Cleans up on unmount

Conclusion

The Chat Room application implements a robust, architecture suitable for real-time messaging. The three-tier design with Docker containerization provides flexibility and maintainability. JWT-based authentication with refresh tokens ensures security while maintaining good user experience. PostgreSQL with Prisma ORM offers reliable data persistence with type safety. Socket.IO enables real-time communication with automatic fallbacks.