Chat Rooms Project - Technical Report

Project Name: Chat Rooms
Technology Stack: Next.js, Node.js, Socket.IO, TypeScript, Docker

By: Abhinav Mathew
B240115CS

Executive Summary

The Chat Rooms project is a real-time messaging application that enables users to create temporary chat rooms and communicate securely with others. Built using modern web technologies, it features a React-based frontend with Next.js and a Node.js backend powered by Socket.IO for real-time communication. The application includes message encryption, room management, and a responsive user interface.

Folder Structure

```
Chat-Rooms/
|— .git/
|— .next/
|— node_modules/
|— .gitignore
|— Chat_Rooms_Project_Report.docx
|— README.md
|— eslint.config.mjs
|— next-env.d.ts
|— next.config.ts
|— package-lock.json
|— package.json
|— postcss.config.mjs
|— tsconfig.json
|— app/
│     ├── favicon.ico
│     ├── globals.css
│     ├── layout.tsx
│     ├── page.tsx
│     │
│     ├── components/
│     │     ├── Chatmessage.tsx
│    │    ├── Chatroom.tsx
│    │    ├── Chatroombox.tsx
│    │    ├── Createroom.tsx
│    │    ├── Messagebar.tsx
│    │    ├── SocketContext.tsx
│    │    └── UsernameModal.tsx
│    │
│    ├── ui/
│    │    └── Toast.tsx
│    │
│    └── utils/
│         └── encryption.ts
│
|─ backend/
│    ├── node_modules/
│    ├── Dockerfile
│    ├── package-lock.json
│    ├── package.json
│    ├── rooms.json
│    └── server.js
│
└── public/
   |─ file.svg
   |─ globe.svg
   |─ next.svg
   |─ vercel.svg
   |─ window.svg
```

Project Architecture Overview

1. Frontend Architecture (Next.js Application)

The frontend is built using Next.js 16.0.8 with App Router and React 19.2.1, providing a modern, server-side rendered web application with the following key characteristics:
• Framework: Next.js with App Router architecture
• Language: TypeScript for type safety
• Styling: Tailwind CSS for responsive design
• State Management: React Context for socket management
• Real-time Communication: Socket.IO client integration

2. Backend Architecture (Node.js Server)

The backend is a standalone Express.js server with Socket.IO integration:
• Runtime: Node.js with Express framework
• Real-time Engine: Socket.IO for WebSocket communication
• Data Persistence: File-based storage using JSON files
• Security: CORS configuration for cross-origin requests

Core Functionality Analysis

1. Real-Time Communication System

Socket Context Provider (SocketContext.tsx)
• Establishes and manages a single WebSocket connection across the entire application
• Implements connection state management with 'isConnected' state tracking
• Provides username persistence using localStorage
• Creates a custom useSocket() hook for component-level socket access

Key Features:
• Automatic reconnection handling
• Centralized socket state management
• Username registration system
• Environment-specific backend URL configuration

2. Room Management System

Backend Room Handling (server.js)
The server implements a comprehensive room management system with the following capabilities:
Room Creation:
• Temporary rooms with configurable durations (1-1440 minutes)
• Automatic room expiry with cleanup timers
• Owner-based room permissions
• Persistent storage in rooms.json

3. Message Encryption System

Encryption Implementation (encryption.ts)
The application implements a custom Caesar cipher-based encryption system:
Encryption Process:
1. Message converted to Base64 encoding
2. Caesar cipher applied with 7-character shift
3. Double Base64 encoding for additional obfuscation

Security Features:
• Client-side encryption before transmission
• Server-agnostic message handling (server never sees plaintext)
• Automatic decryption on message receipt
• Fallback handling for encryption errors

Technical Implementation Details

1. Development Environment Setup

Concurrent Development:
The project uses 'concurrently' package to run both frontend and backend simultaneously through the dev:full script, enabling efficient development workflow.

Development Dependencies:
• TypeScript for type safety
• ESLint for code quality
• Tailwind CSS for styling
• Nodemon for backend hot-reload

Key Features Summary

Implemented Features
1. Real-time messaging with Socket.IO
2. Temporary room creation with configurable duration
3. Message encryption using custom Caesar cipher
4. Responsive UI with Tailwind CSS
5. User authentication via username system
6. Room management with automatic cleanup
7. Message history during session
8. Auto-scroll chat interface
9. Toast notifications for user feedback
10. Cross-origin support for development

Usage Workflow

User Journey:
1. Access Application: Navigate to localhost:3000
2. Set Username: Enter preferred username in modal
3. View Rooms: Browse available chat rooms
4. Create Room: Optionally create new temporary room
5. Join Room: Click to join existing room
6. Send Messages: Type and send encrypted messages
7. Real-time Chat: Receive messages from other users instantly
8. Auto-cleanup: Rooms expire automatically based on duration

Conclusion

The Chat Rooms project successfully demonstrates a full-stack real-time messaging application with modern web technologies. The implementation showcases proper separation of concerns between frontend and backend, real-time communication patterns, basic security measures, and a clean, responsive user interface.

The project serves as an excellent foundation for understanding Socket.IO integration, React state management, and full-stack development workflows with Next.js and Node.js. The modular architecture and TypeScript implementation provide a solid foundation for future enhancements and production deployment with appropriate scaling considerations.

THE PROJECT IS ALREADY HOSTED VIA VERCEL(FRONTEND) AND BACKEND IS HOSTED ON RAILWAY

YOU CAN ACCESS THE PROJECT DIRECTLY WITH THIS LINK

chat-rooms-beta.vercel.app

MORE DETAILED WORK IS PROVIDED ON THE SCAN BELOW

SERVERCODE EXPLANATION

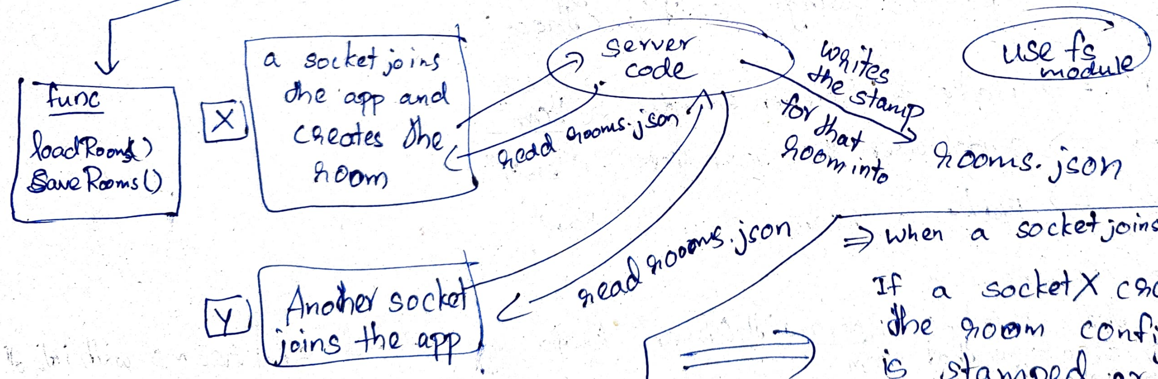
server.js

① Import npm packages (Install everything via npm)

② CORS is middleware [app.use(cors())] ⇒ where app is the express instance.

③ Express cannot handle websockets alone, so we use raw http module.
↳ Server config is stored in "io" variable.

④ Room management



func
loadRooms()
Save Rooms()
X
a socket joins the app and creates the room
server code
read rooms.json
writes the stamp for that room into
rooms.json
use fs module
Y
Another socket joins the app
read rooms.json

⇒ when a socket joins the app, rooms.json is read and broadcast

If a socket X creates a room, the room config and its id is stamped or written into the ~~jso~~ rooms.json file.

⇒ So when a socket Y joins after this, he will be able to see the room created by socket X

⑤ In memory mappings

userSocketMap ⇒ userId → socket ~~ID~~.id
userRoomsMap ⇒ socket.id → Set(roomIds)
socketUsernameMap ⇒ socked.id → username
roomTimers ⇒ roomId → timeout ID

⑥ <u>Socket Lifecycle</u>

io variable is used to set up a websocket connection, then allows a socket to join that connection with a particular socket id, and that socket can listen to different events using the "on" method and sends an event using "emit" method.

<u>Events in the app</u>

'register-user' ⇒ make a listener ready <u>"on"</u>
'existing-rooms' ⇒ sends user the room list "emit",
'create-room' ⇒ make a room "on"
→ newRoom is new one,
then the variable <u>roomwithOwner</u> keeps a record of rooms with info of
⇒ who owns that room
⇒ when ~~its~~ expires

'room-created' ⇒ new room made ⇒ io.emit ⇒ It is a broadcasting (updates everyone with new room)

The RoomTimers Map will handle the ~~recod~~ record of Timeouts along with timer id and roomowner.

So every room, the countdown of that particular room is broadcasted.

'join_chatroom' ("on") ⇒ Join a Room ⇒ along with the data required, this event groups the sockets which are trying to join into same room id.

↳ for this

$\boxed{\text{⊗ socket.join(roomId)}}$ is done

⇒ io.to(roomId).emit("user_joined_room", { - - - })

↳ What this does ~~it~~ is that it sends a toast to a specific room ~~for~~, when a user join that room
(I'm only talking about the purpose)

'leave_chatroom' ⇒ socket leaves the room ⇒ "on"
(use socket.leave(roomId))

'delete_room' ⇒ delete a room ⇒ "on"
(only owners can delete their own rooms)

if someone ~~deletes~~ deletes a room.

$\boxed{\text{io.in(roomId).socketsLeave(roomId)}}$ ⟹ kicks everyone out of that room

Room Expiration ⟹ handleRoomExpiry()

- Notify users
- Deletes rooms
- Kick every one
- Remove the timer
- Saves the state and json data changed.

```
setInterval(() => {
    io.to(roomId).emit('timer_update', {---})}, 1000)
```

Every single second, room timer is updated and broadcasted to everyone in that particular room!

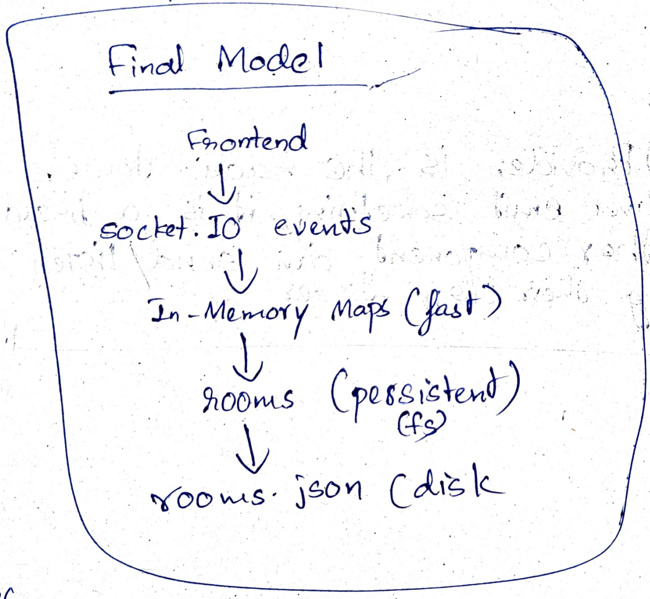
Messaging

```
socket.on('send-msg', (data) => {
        socket.to(data.room).emit('recieve-msg', data)
    });
```

⟹ Sender does not get his message as someone else's (logical)
⟹ Senders' msg is broadcasted into ~~to~~ the ~~room~~ current room.

'disconnect' ⇒ ~~out~~ socket leaves app ⇒ "on"

⟹ Cleanup is done, all rooms that socket ownered will be wiped from rooms.json, and the room statuses will be updated.



Extra Stuff

Encryption: Basic E2EE ⇒ Just used a simple cipher to hide the actual messages from server, server sees only some jibberish, but ~~the~~ just before the message is broadcasted, the message is deciphered in client side.

⇒ SocketContext.tsx : Creates a custom hook "useSocket", this is of course used in frontend, it provides a centralised socket Management, one socket listen to all events globally,

↳ Think like this

→ SocketProvider is the radio tower
→ when server emit something, it is a broadcast from tower
→ every other component can send/listen without building their own tower.