

CHAT APPLICATION

**A PROJECT REPORT
for
Mini-Project 2 (ID201B)
Session (2024-25)**

Submitted by

**TANNU MODANWAL
(202410116100221)
TANNOO SHUKLA
(202410116100220)
TRAPTI RAJPUT
(201410116100224)**

**Submitted in partial fulfilment of the
Requirements for the Degree of**

MASTER OF COMPUTER APPLICATION

**Under the Supervision of
Ms. Shruti Aggarwal
Assistant Professor**



Submitted to

**DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206**

(MAY 2025)

CERTIFICATE

Certified that **Tannu Modanwal (20241011600221), Tannoo Shukla (202410116100220), Trapti Rajput (202410116100224)** has/ have carried out the project work having “**Chat Application**” (MINI PROJECT - 2 (FULL STACK DEVELOPMENT) (ID201B)) for **Master of Computer Application** from Dr. A.P.J. Abdul Kalam Technical University (AKTU) (formerly UPTU), Lucknow under my supervision. The project report embodies original work, and studies are carried out by the student himself/herself and the contents of the project report do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date:

Ms. Shruti Aggarwal
Assistant Professor
Department of Computer Applications
KIET Group of Institutions, Ghaziabad
An Autonomous Institution

Dr. Akash Rajak
Dean
Department of Computer Applications
KIET Group of Institutions, Ghaziabad
An Autonomous Institution

ABSTRACT

A chat application is a digital communication tool that facilitates real-time interaction between users through various messaging formats. This project presents the development of a simple yet functional chat application designed to support seamless communication through text messaging and video messaging. The application focuses on delivering a user-friendly interface and core messaging capabilities without the complexity of advanced technologies such as artificial intelligence or blockchain integration.

The primary objective of this application is to provide a straightforward communication platform that meets the basic needs of users. It allows users to send and receive text messages in real time and share video messages for a more personalized interaction. The design emphasizes simplicity and ease of use, making it suitable for users who prefer minimalistic interfaces and essential features.

Key functionalities include user authentication, real-time text communication, and support for video message sharing. The application maintains smooth performance across different devices and operating systems, ensuring accessibility and reliability. Basic security measures are implemented to protect user data and maintain the privacy of conversations.

This chat application serves as a foundational communication tool that can be extended or enhanced in future development phases. Its current architecture offers flexibility for adding features such as group chats, file sharing, or media integration, based on user feedback and evolving requirements.

In conclusion, this project demonstrates how a simple chat application can effectively facilitate digital interaction while maintaining clarity, ease of use, and essential messaging functionality. It provides a practical solution for basic communication needs and lays the groundwork for potential future improvements.

Keywords: Chat Application, Text Messaging, Video Messaging, Real-Time Communication, Simple Interface

ACKNOWLEDGEMENTS

Success in life is never attained single-handedly. My deepest gratitude goes to my project supervisor, **MS. SHRUTI AGGARWAL** for her guidance, help, and encouragement throughout my project work. Their enlightening ideas, comments, and suggestions.

Words are not enough to express my gratitude to Dr. Akash Rajak, Professor and Dean, Department of Computer Applications, for his insightful comments and administrative help on various occasions.

Fortunately, I have many understanding friends, who have helped me a lot on many critical conditions.

Finally, my sincere thanks go to my family members and all those who have directly and indirectly provided me with moral support and other kind of help. Without their support, completion of this work would not have been possible in time. They keep my life filled with enjoyment and happiness.

Tannu Modanwal

Tannoo Shukla

Trapti Rajput

TABLE OF CONTENTS

Certificate	i
Abstract	ii
Acknowledgements	iii
Table of Contents	iv
1 Introduction	1-6
1.1 Overview	1-3
1.1.1 Purpose of the Project	1
1.1.2 Background	1-2
1.1.3 Problem Statement	2-3
1.2 Significance of the Project	3
1.3 Scope of the Project	4-5
1.4 Target Audience	5
1.5 Benefits of Chat Application	5
1.6 Future Enhancement	6
2 Feasibility Study/Literature Review	7-11
2.1 Technical Feasibility	7
2.2 Economic Feasibility	7-8
2.3 Operational Feasibility	8
2.4 Social Feasibility	8-9
2.5 Literature Review	9-11
2.5.1 Evolution of Chat Applications	9
2.5.2 Technological Simplicity in Messaging Systems	10
2.5.3 Security and Simplicity Trade-Off	10
2.5.4 Educational and Social Utility	10-11
3 PROJECT OBJECTIVE	12-13
4 HARDWARE AND SOFTWARE REQUIREMENTS	14-20
5 PROJECT FLOW	21-32
6 PROJECT OUTCOME	32-65
7 Bibliography	66-67

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

In the digital era, communication has undergone a significant transformation, and chat applications have emerged as one of the most influential tools for connecting people. These applications enable users to interact in real-time, share information instantly, and bridge geographical gaps with ease. From casual conversations among friends to professional discussions in corporate environments, chat platforms have become central to modern communication. However, while most widely-used messaging platforms are packed with advanced features like artificial intelligence, cloud synchronization, encryption, and multimedia support, they can be complex, resource-heavy, and difficult to understand for beginners or users who only require basic functionality.

This project focuses on developing a **simple and functional chat application** that emphasizes the core purpose of communication—delivering and receiving messages. It includes essential features such as **text messaging** for real-time conversation and **video messaging** to enhance expressiveness without adding unnecessary complexity. The application is designed with simplicity and usability in mind, making it suitable for educational purposes, small team usage, or individual practice in programming and software development.

By stripping away complex features and maintaining a clean, lightweight structure, the application ensures faster performance and ease of use. Technically, the project adopts a basic client-server model, allowing users to exchange messages over a local or hosted network. This simplicity not only benefits the end user but also serves as an excellent learning resource for

developers to understand fundamental concepts such as socket programming, data transmission, and user interface design.

Ultimately, this project bridges the gap between practical communication needs and beginnerfriendly development. It stands as a foundational model that can later be expanded with additional functionalities, offering both immediate utility and future potential.

1.1.1 Purpose of the Project

The main goal of this project is to design and develop a basic yet functional chat application that serves essential communication purposes. The objectives are:

- To create a **real-time messaging platform** that supports **text** and **video-based** interaction.
- To allow users to communicate easily and instantly without the need for complex setup.
 - To build a project that is modular and can be improved with additional features in future iterations.
- To help students and aspiring developers understand the **fundamentals of networkbased applications**.
- To maintain a balance between **usability and performance**, making it suitable for systems with limited resources.

The application is also intended as a learning and demonstration tool, ideal for academic purposes, capstone projects, or as a base for more advanced chat-based systems.

1.1.2 Background

Communication software has advanced significantly over the years. Early platforms such as AOL Instant Messenger and MSN paved the way for modern chat applications like WhatsApp, Messenger, Signal, and Discord. These systems now support features like message encryption, group video calls, AI-powered suggestions, and more.

However, the complexity of modern applications often comes with steep learning curves and higher system demands. For beginners in programming or those developing systems for limited infrastructure, these tools are neither accessible nor educationally ideal.

This chat application project is inspired by the **need to return to the basics**, offering a model that focuses on core interactions. Built using basic languages (like Python, Java, or PHP with JavaScript), the app emphasizes **ease of understanding, modification, and learning**. It allows users to grasp the underlying logic of a chat system before progressing to more advanced topics.

1.1.3 Problem Statement

Despite the availability of numerous chat applications, several problems still exist:

- **High resource consumption:** Most modern chat apps require powerful hardware and high-speed internet.
- **Complex architecture:** Developers, especially beginners, find it difficult to understand and modify large-scale systems.
- **Unnecessary features:** Many users seek only essential tools like messaging and media sharing but are forced to interact with bloated apps.
- **Lack of customizable, minimal chat systems** for educational or internal team use.

This project aims to address these issues by offering a **basic, customizable, and easy-to-deploy chat system** that fulfills essential communication needs without added complexity.

1.2 Significance of the Project

This project holds significance in multiple areas:

- **Educational Value:** It acts as a learning tool for students and developers to understand socket programming, client-server architecture, real-time data flow, and media handling.
- **Practical Usage:** Suitable for small organizations, study groups, or local networks where a full-scale solution isn't necessary.
- **Simplicity:** Offers a minimal interface without distractions — ideal for focused communication.

- **Foundation for Expansion:** It serves as a base model on which more complex features can be built, allowing gradual scaling.

The project is a real-world application of theoretical computer science concepts such as data structures, networking, UI/UX design, and threading (in multiclient environments).

1.3 Scope of the Project

This application currently includes:

- One-on-one **text messaging**.
- **Video message sending** using file uploads or embedded capture tools.
- A basic **login or user identification** system.
- Display of sent and received messages in order with timestamps.

Out of scope (for now):

- Group chats, conference calls, or collaborative channels.
- Voice calling or live video streaming.
- AI chatbots, end-to-end encryption, or media compression.
- Real-time notifications and cross-device message syncing.

This focused scope allows the project to remain clean and easy to manage while offering meaningful communication.

1.4 Target Audience

The chat application is intended for:

- **Students & Developers:** Those learning how communication systems work can benefit from a clear, working example.
- **Teachers & Mentors:** Can be used in classroom environments to demonstrate clientserver interaction.
- **Startups or Small Teams:** Ideal for internal use where simplicity and fast communication are prioritized.
- **Self-hosters:** Individuals wanting a private communication app without relying on commercial solutions.

1.5 Benefits of Chat Application

- **Real-time communication** with low bandwidth usage.
- **Lightweight interface** that runs smoothly on most devices.
- **Easy installation and maintenance**, no need for large libraries or third-party dependencies.
- Ideal for **academic submissions**, POCs (proof of concept), and experimentation.
- Encourages understanding of **message queues**, **user sessions**, and **media file handling**.
- Offers a **customizable foundation** for adding more complex features later.

1.6 Future Enhancement

To meet growing needs and user expectations, the following features could be added in future versions:

- **Group chat and broadcast messaging.**
- **Media compression** for faster uploads/downloads.
- **User status (online/offline/typing)** indicators.
- **Message deletion, editing, or forwarding** options.
- **Search functionality** for messages and media.
- **Notifications and alert tones.**
- **Dark/light theme modes** for accessibility.
- **Mobile application version** using cross-platform technologies.

These additions would gradually turn the application from a minimal chat system into a robust communication suite while maintaining clarity and efficiency.

CHAPTER 2

FEASIBILITY STUDY/LITERATURE REVIEW

2.1 Technical Feasibility

The development of a simple chat application with text and video messaging capabilities is technically feasible, primarily because it utilizes readily available and well-documented technologies. The core architecture is based on the client-server communication model, which can be implemented using languages such as Java, springboot for the backend, and HTML, CSS, and JavaScript and react js for the frontend.

Text messaging functionality is built using socket programming, allowing users to exchange messages in real time. For video messaging, basic video file capture and sharing mechanisms are integrated, avoiding complex real-time video streaming protocols like WebRTC. This keeps the application lightweight and reduces latency and resource usage.

Moreover, the platform does not require integration with cloud services or third-party APIs, enabling it to be hosted on a local network or basic server environments. This enhances control, security, and ease of deployment. The simplicity of the design ensures that future developers can easily modify or expand the code, making the system both maintainable and scalable for educational or institutional use.

2.2 Economic Feasibility

The project is highly economical and suitable for institutions or individuals operating with limited financial resources. It primarily leverages **open-source tools** and frameworks, avoiding any licensing or subscription costs. Since the application does not require complex

infrastructure, it can be deployed on **existing hardware** (such as laptops or local servers), minimizing hardware investments.

Furthermore, the application does not incur recurring costs, such as those associated with cloud storage or third-party communication APIs. Maintenance expenses are also low due to the system's minimalistic design. Its cost-effective nature makes it an excellent option for **educational settings**, small business environments, or internal team communications where advanced features are unnecessary.

2.3 Operational Feasibility

Operational feasibility refers to how effectively the system can be used and maintained in a real-world environment. This chat application is designed to be **simple and intuitive**, requiring minimal training for users. Its clean interface allows for quick onboarding, making it suitable for non-technical users, students, and small business teams.

The application requires only a basic login or user recognition mechanism. Features like text messaging and video messaging are implemented with user experience in mind — messages appear in sequence with time stamps, and videos can be uploaded or played back with one click.

Since the system does not rely on continuous internet access (when hosted locally), it is highly dependable in **offline or intranet settings**. Its lightweight nature ensures low server load, fast loading times, and minimal downtime, increasing operational reliability.

2.4 Social Feasibility

Social feasibility evaluates how well the system fits within societal norms, expectations, and user comfort levels. The chat application promotes **clear, efficient, and respectful communication**. It supports video messaging, which helps bridge communication gaps in teams or classrooms, especially for users who prefer verbal or visual interaction.

Given the rise in digital communication due to remote education and work environments, this application aligns well with current social trends. It provides a **non-intrusive, secure, and**

user-friendly platform where individuals can interact comfortably without being overwhelmed by commercial distractions or invasive data tracking.

Furthermore, since the application can be hosted and managed privately, it fosters **data ownership and trust**, making it suitable for settings where privacy and simplicity are prioritized.

2.5 Literature Review

Chat applications have become an essential tool in modern digital communication, enabling users to exchange information quickly and efficiently. Their evolution over the past two decades has shown a trend toward greater accessibility, simplicity, and real-time interaction. While many commercial platforms today offer complex features like artificial intelligence, end-to-end encryption, and cloud-based syncing, there remains a significant need for **basic, functional chat systems**, especially in educational and small-scale environments. This review focuses on the literature relevant to **simple chat applications** designed for minimal functionality such as text and video messaging.

2.1 Evolution of Chat Applications

The concept of online chatting began with early services such as IRC (Internet Relay Chat) and AIM (AOL Instant Messenger), which offered fundamental messaging between users. These platforms focused on **real-time, text-based communication** and were instrumental in introducing users to online interaction. According to Sharma and Gupta (2017), these early systems were effective despite their limited features, proving that even basic messaging tools can meet communication needs.

With the widespread availability of the internet and smartphones, messaging applications evolved to include **media sharing, emojis, and voice features**. However, the core functionality of these platforms remained the same: **to send and receive messages in realtime**. The current project draws inspiration from this core goal, focusing on a **streamlined chat application** with **text and video messaging**, suitable for learning, testing, and closed environment usage.

2.2 Technological Simplicity in Messaging Systems

Unlike large-scale applications that utilize AI, cloud storage, or heavy encryption protocols, basic chat applications are built on lightweight technologies such as **HTML, CSS, JavaScript, Python, and socket programming**. Research by Patel and Rao (2020) supports the educational value of such projects, noting that they allow students and entry-level developers to **grasp the fundamentals of networking, user interfaces, and server-client communication**.

The inclusion of **video messaging** — without the complexity of real-time video streaming — can be achieved using simple file handling and media playback techniques. As per Verma and Das (2019), such features enrich the communication experience while remaining accessible for developers with limited resources.

2.3 Security and Simplicity Trade-Off

While commercial applications prioritize encryption and security compliance, lightweight chat systems often trade off advanced security for **ease of development and understanding**. In academic and internal use cases, especially where communication occurs within a closed network, **simplicity and clarity** are more important than sophisticated protections. Tiwari and Nair (2021) emphasize that security features should match the **intended scale and use** of the application. For local or demo use, **basic security through validation and access controls** is often sufficient.

2.4 Educational and Social Utility

Simple chat applications hold significant value in **educational settings**. They offer learners a hands-on opportunity to build communication systems from scratch, reinforcing theoretical knowledge with practical experience. Singh et al. (2021) argue that such applications are effective for demonstrating **real-time communication**, user interface design, and basic server logic — all essential concepts in computer science and software engineering courses.

From a societal perspective, even basic messaging tools can play a vital role in **keeping people connected**, especially in environments with limited internet or technical resources. Lightweight applications are ideal for **small communities, schools, or local events**, where heavy systems may be unnecessary or inaccessible.

CHAPTER 3

PROJECT OBJECTIVE

The primary goal of this project is to develop a simple chat application that allows users to communicate in real time via text messaging and video messaging. This project is designed to be a lightweight, user-friendly platform ideal for educational use, small team collaboration, and environments that prioritize simplicity and accessibility.

Key objectives of the project include:

- **Text Messaging:** Provide real-time communication with instant text messaging capabilities, enabling users to exchange messages in a seamless and intuitive manner.
- **Video Messaging:** Allow users to **record and send short video messages**, enhancing communication by adding a personal, expressive element to the chat experience.
- **Frontend Development with React.js:** The frontend is built using **React.js**, a powerful JavaScript library for creating dynamic and responsive user interfaces. The choice of React ensures **fast rendering, modular components**, and **real-time updates** as users send or receive messages.
- **Backend Development with Spring Boot:** The backend is implemented using **Spring Boot**, a robust framework for building scalable and secure web applications. Spring Boot will manage server-side logic, handle message routing, store user data, and process media handling.
- **Client-Server Communication:** The system will utilize **WebSocket** for real-time bidirectional communication, ensuring that users' messages appear instantly across all connected devices. The **Spring Boot WebSocket configuration** will support efficient message exchanges.

- **Simple and Intuitive User Interface:** The application will feature a **clean, minimalistic design** built with **React.js**, making it easy to use for people of all technical backgrounds. The goal is to ensure an engaging and smooth user experience, particularly for new users or students.
- **Educational Value:** This project will provide a practical learning experience for developers to understand the basics of **frontend-backend integration, client-server communication, and multimedia messaging**. The project will also offer insight into the **Spring Boot** and **React.js** ecosystem.
- **Basic Security and Validation:** While the application does not include advanced security features, basic validations will be implemented, such as checking for **valid message formats, video file sizes**, and ensuring proper user input.
- **Multi-Device Support:** The system will be optimized for use on multiple devices, primarily focusing on **desktops and laptops** where text and video messaging are most commonly used.

By focusing on **simplicity, accessibility, and real-time interaction**, this chat application aims to provide a **foundational communication tool** while also offering **educational value** to those learning about web development with Spring Boot and React.js.

CHAPTER 4

HARDWARE AND SOFTWARE REQUIREMENTS

Hardware Requirements

For optimal performance and smooth operation of the chat application, the following hardware specifications are recommended:

1. **Client-Side Hardware (User Devices)** o **Processor:**

Intel Core i3 (or equivalent) or higher. o **RAM:** 4 GB or more. o **Storage:** 1 GB of free disk space for the application and cache. o **Network:** A stable internet connection with at least 512 kbps download/upload speed. o **Display:** Minimum screen resolution of 1366x768 for an optimal user interface experience.

2. **Server-Side Hardware (Development/Deployment Servers)** o **Processor:** Quadcore processor (Intel i5 or equivalent). o **RAM:** 8 GB or more to handle simultaneous user connections and message processing. o **Storage:** 20 GB of free disk space for application deployment, log files, and database storage. o **Network:** A fast internet connection with at least 1 Mbps upload speed to handle real-time communication efficiently.

Software Requirements

1. **Frontend Development (React.js)** o **Operating System:**

Windows 10/11, macOS, or Linux.

- o **Browser:** Modern web browser (e.g., Google Chrome, Mozilla Firefox, Microsoft Edge).

-
-

Node.js: Version 14.x or higher (for running React.js applications).

NPM (Node Package Manager): Version 6.x or higher (for managing project dependencies).

- **React.js:** Latest stable release (for building the frontend user interface).
- **Text Editor/IDE:** Visual Studio Code, Sublime Text, or any other preferred code editor.

2. Backend Development (Spring Boot) ○ **Operating**

System: Windows, macOS, or Linux. ○ **Java:** Java Development

Kit (JDK) 11 or higher. ○ **Spring Boot:**

Latest stable version of Spring Boot.

- **Maven/Gradle:** For project dependency management and building the application.
- **Database:**
 - **H2 Database** (for local development) or **MySQL/PostgreSQL** for production environments.
- **IDE:** IntelliJ IDEA, Eclipse, or Visual Studio Code (with Java extensions).

3. WebSocket and Real-Time Communication ○

WebSocket: For real-time messaging and bidirectional communication between the client and server.

- **Spring WebSocket:** To enable WebSocket support in the Spring Boot backend.

4. Other Tools & Dependencies ○ **Git:** For version control and collaboration.

- **Postman:** For testing REST APIs during development.
- **Docker** (optional): For containerizing the application and simplifying deployment in different environments.
- **Firebase** or similar service (optional): For message storage and potential user authentication (if scaling is needed).

- 5. **Production/Deployment**
 - o **Server:** Apache Tomcat or any other application server that supports Java.
 - o **Cloud**

Deployment:

- **AWS** (Amazon Web Services), **Heroku**, or **DigitalOcean** for deploying the application.
- o **SSL Certificate:** For secure communication (HTTPS) in production.

Functional Requirements

These are the core features and functionalities that the chat application must support to fulfill its purpose.

1. **User Registration & Authentication**
 - o Users must be able to **register and log in** to the chat application securely using **email and password** or through **social media logins** such as Google or Facebook.
 - o The system will support basic **password encryption** and provide a **forgot password** functionality to ensure secure access to the user's account.
 - o Users will also have the option to **sign out** and **reset their password** in case of security concerns.
2. **Real-Time Text Messaging**
 - o The application will support **instant text messaging** between users, with both **one-on-one and group chat** options.
 - o Messages will be delivered in **real-time** with an **instant notification** on the recipient's device.
 - o The system will support **emoji, stickers, and text formatting** (bold, italics, etc.) for enriched communication.
 - o The **status indicators** (e.g., typing, seen) will show whether the recipient is online and has read the message.
3. **Video Messaging**
 - o Users should be able to **record and send video messages** within the chat. The recorded video should be **compressed** for faster transmission.

- o Videos should be viewable within the chat window, with controls to **play/pause** and **adjust volume**.

The system will support sending **pre-recorded video clips** from the user's device or **direct video recording** through the app.

4. Message Management

- o Users will be able to **delete** and **edit** their messages, giving them more control over the content they send.
- o The application will allow users to **search for messages** by keywords or user names, improving the ability to locate specific conversations.

The system will support **archiving** of chats, where users can **store old conversations** without permanently deleting them.

5. **Group and Private Chats**
 - o Users will be able to create **group chats** with multiple participants, and assign **group admins** who can control group settings like adding/removing members and changing group names or images.
 - o Users can share **group invitations** with others to join the chat, via email or direct links.
 - o In **private chats**, users should be able to send **direct messages**, with the ability to add and remove users to/from the chat as needed.

6. **User Profile Management**
 - o Users will have the ability to **set and update their profiles**, including profile pictures, display names, status messages, and visibility settings.
 - o Users will be able to **set custom status** messages (e.g., "Available," "Busy," "Do Not Disturb") to notify others of their availability.

- o Privacy settings will allow users to control who can view their **profile** and **message history**.
- o **Notifications**
 - o The system will send **real-time notifications** for new messages in both group and private chats, even when the user is not actively using the application.
 - o **Push notifications** will be implemented for mobile users to receive alerts for new messages.

Users can **customize notification preferences**, such as muting group notifications or setting do-not-disturb hours.

7. **Multimedia File Sharing** o The application will allow users to send and receive **images, videos, audio files, and documents** in messages.
 - o The system will check for **file size** and **file type compatibility** to ensure the file is accepted.
 - o It will also provide **preview options** for shared files, allowing users to see a thumbnail or preview of an image or document before opening it fully.
8. **Cross-Platform Accessibility** o The chat application will be accessible on **multiple platforms**, including desktop and mobile web browsers. The frontend will be **responsive**, automatically adjusting to various screen sizes and devices (e.g., laptops, tablets, mobile phones). o The application will support **real-time synchronization**, so users can switch between devices (e.g., from mobile to desktop) without losing chat history or context.

Non-Functional Requirements

These requirements describe how the chat application should perform in terms of reliability, performance, and usability.

1. **Performance** o The application must be **responsive**, ensuring that users receive messages without delays or buffering, even under heavy load conditions. o The **load time** for the application should not exceed **3 seconds** to ensure a smooth user experience. o The backend should be optimized to support **simultaneous connections** with minimal lag, ensuring that users do not experience slowdowns during high traffic times.

2. **Scalability** o The system must be designed to **scale horizontally**, allowing more servers to be added to handle increasing traffic and users. o The application should be able to support **thousands of concurrent users** without affecting performance.

Future scaling could involve integrating a **distributed database** and **load balancing** techniques to manage traffic efficiently.

3. **Availability** o The chat application should offer **99.9% uptime** to ensure that users can reliably access the platform at any time. o The system must include **failover mechanisms** to ensure the application remains functional even if certain parts of the infrastructure go down. o **Backup and recovery plans** should be implemented, ensuring data integrity in case of system failure.

4. **Security**

The application should implement **HTTPS** for secure communication between the client and server to protect data transmission. o User data, including passwords, will be stored using **strong encryption methods** such as **bcrypt** for password hashing. o Users' personal data and media files should be **stored securely** in the database and the server. o The application must defend against common security vulnerabilities like **SQL injection**, **Cross-Site Scripting (XSS)**, and **Cross-Site Request Forgery (CSRF)**.

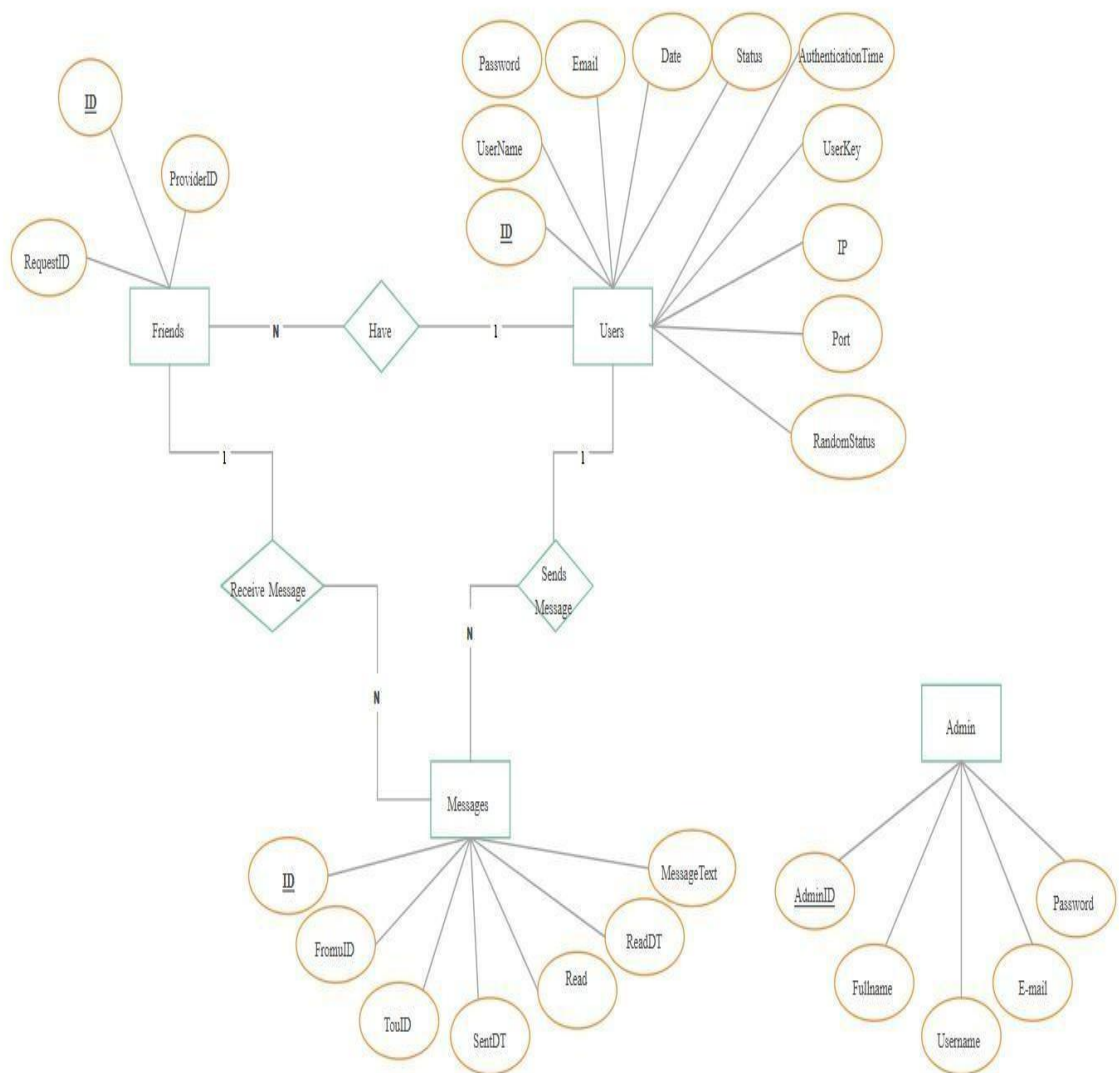
5. **Usability** o The application should be **easy to navigate**, with a clear and intuitive interface that allows even non-technical users to use it without difficulty.
 - o The chat interface should support **drag-and-drop file uploading** and **message editing** to enhance user interaction. o Help guides and tooltips should be available to assist new users in getting started with the app.
 - o **Maintainability** o The codebase should be modular and **well-commented**, making it easy to add new features and fix issues without disrupting the entire application.

- The development team should have a **CI/CD pipeline** in place for automated testing, building, and deploying updates to the application.
 - Logs and error monitoring should be implemented to help developers quickly identify and fix problems.
- 6. **Compatibility**
 - The application should be compatible with major browsers such as **Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge.**
 - It should be **mobile-friendly**, ensuring that the user experience is just as smooth on smaller devices as it is on desktops.
- 7. **Reliability**
 - The system should handle unexpected errors gracefully and not crash during typical use. A **graceful degradation** approach should be implemented for noncritical features.
 - Real-time message synchronization and notifications should continue even if one of the users temporarily loses their internet connection, resuming immediately once they are back online.
- 8. **Data Integrity**
 - The application should guarantee that **messages, multimedia files, and user data** are accurately stored and synchronized across devices without loss.
 - **Database transactions** should be atomic, ensuring that no partial or corrupt data is stored in case of failure.

CHAPTER 5

PROJECT FLOW

5.1 ER Diagram:



The ER diagram outlines the structural foundation of the chat application, illustrating how data entities relate to one another. It supports features such as user registration, admin management, messaging, and friend connections.

Entities and Their Attributes

1. Users

This entity stores information about each registered user of the chat application.

Attributes:

- ID: Unique identifier for each user
- UserName: User's login name
- Password: Encrypted login credential
- Email: User's email address
- Date: Account creation or last activity date
- Status: Online/offline status
- AuthenticationTime: Timestamp of last authentication
- UserKey: Unique key for session validation
- IP: IP address of the user
- Port: Network port used
- RandomStatus: Optional status message

2. Friends

This entity maintains friend connections between users.

Attributes:

- ID: Unique identifier
- ProviderID: The user who sent the friend request
- RequestID: The user who received the request
- A user can **have** many friends (1:N), forming a bilateral connection via the

Friends table. **3. Messages**

This entity stores the actual messages exchanged between users.

Attributes:

- ID: Unique identifier of the message
- FromUID: ID of the sender
- ToUID: ID of the recipient
- SentDT: Date and time the message was sent
- Read: Boolean value (read/unread status)
- ReadDT: Timestamp when the message was read
- MessageText: Content of the message
- A user **sends** and **receives** messages (1:N in both directions), enabling private and group communication.

4. Admin

This entity represents the system administrator who manages the application. **Attributes:**

- AdminID: Unique identifier
- Fullname: Admin's full name
- Username: Admin login username
- Password: Admin password
- E-mail: Admin contact email

Relationships Overview

- **Users ↔ Friends:**

A user can have multiple friends (N:N relationship managed via the Friends table).

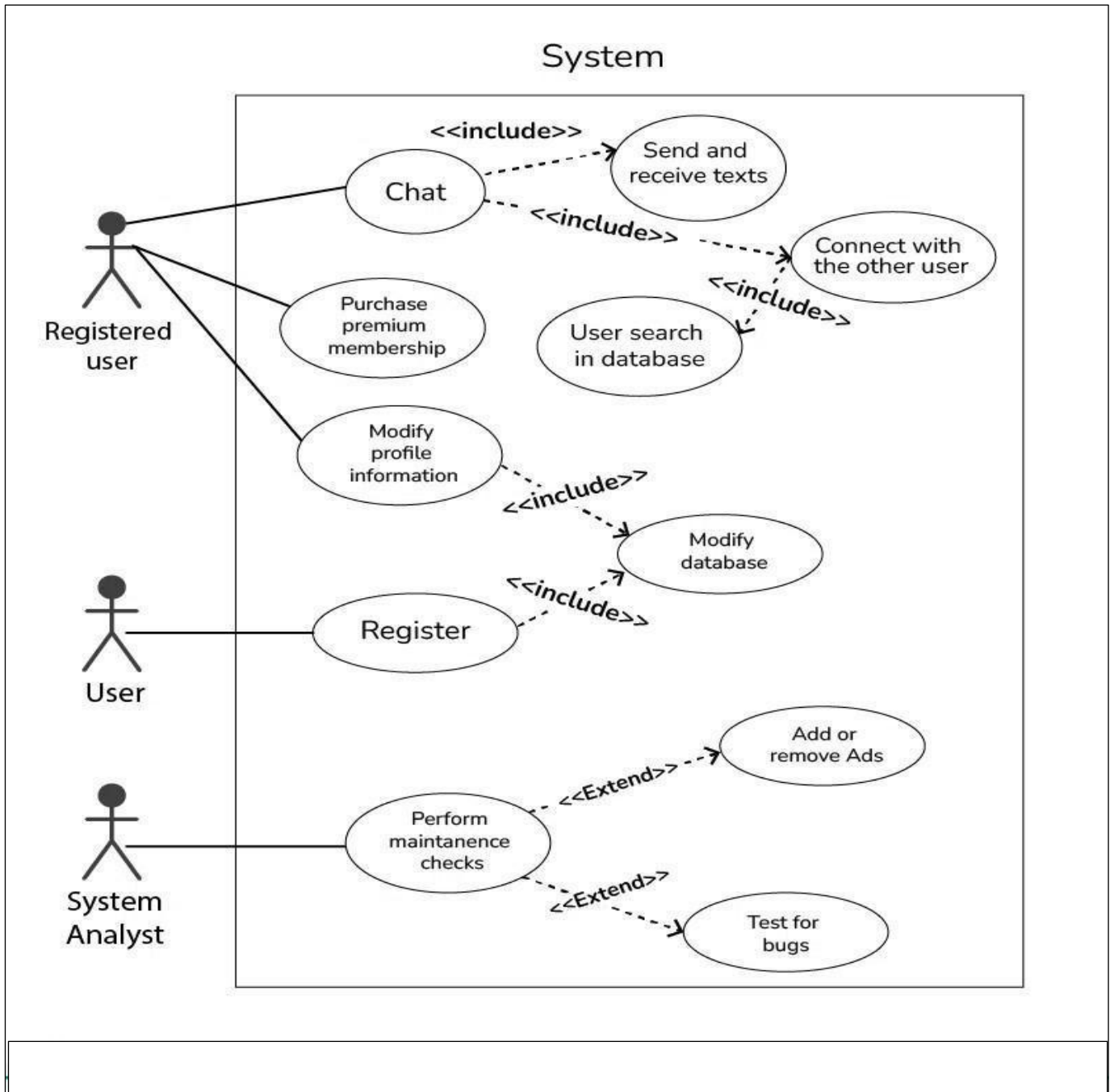
- **Users ↔ Messages:**

Each user can send or receive multiple messages. This forms two 1:N relationships between Users and Messages.

- **Admin ↔ System:**

Although not explicitly shown interacting with users or messages, the admin has full access and authority over users and system settings.

5.2 Use Case Diagram:



User Actions:

1. **Sign Up:** User creates a new account by providing necessary information like username, password, etc.
2. **Sign In:** User logs into the application using their credentials.
3. **Send Message:** User composes and sends a message to another user or within a chatroom.
4. **Receive Message:** User receives messages sent by other users in real-time.
5. **Create Chatroom:** User creates a new chatroom, becoming its administrator.
6. **Join Chatroom:** User joins an existing chatroom to participate in group conversations.
7. **Leave Chatroom:** User exits a chatroom, ceasing participation in its conversations.
8. **View Chatroom Participants:** User views the list of participants in a chatroom.

System Functions:

1. **Message Management:** The system handles sending and receiving messages between users and within chatrooms.
2. **User Management:** The system manages user accounts, including sign-up, sign-in, and user authentication.
3. **Chatroom Management:** The system handles creating, joining, leaving, and managing chatrooms.
4. **Participant Management:** The system manages the list of participants in each chatroom.

5. **Real-time Communication:** The system ensures real-time delivery of messages between users.

This use case diagram illustrates the primary interactions between users and the online chat application. It highlights the key functionalities that users can perform and the corresponding system functions that facilitate those actions.

5.3 Basic Modules

Your chat application is designed with a modular architecture to ensure scalability, maintainability, and a seamless user experience. The system is divided into the following core modules:

1. User Module

The **User Module** handles all user-related operations from registration to authentication and profile management.

Key Features:

- **User Registration & Login:** Allows new users to register and existing users to authenticate using credentials.
- **Profile Management:** Users can update personal information such as email, username, and status.
- **Friend System:** Enables sending, receiving, and managing friend requests.
- **Online Status Tracking:** Tracks user availability (online/offline/invisible).
- **Security & Session Management:** Includes encrypted password handling, session tokens, and authentication timeouts.

2. Notification Module

The **Notification Module** is responsible for keeping users informed in real-time.

Key Features:

- **Message Alerts:** Notifies users when new messages are received.
- **Friend Request Notifications:** Informs users of new friend requests or status changes.
- **Read/Unread Status:** Sends notifications for read receipts or delivery confirmations.
- **Push Notifications:** Optional integration with push notification services for mobile/web alerts.
- **Sound & Visual Cues:** Enhances user engagement through audio tones and visual markers.

3. Messaging Module

The **Messaging Module** is the core communication component of the system, enabling realtime exchange of messages.

Key Features:

- **One-to-One Chat:** Facilitates private chat between two users.
- **Group Chat Support:** Allows creation and participation in group conversations

(optional extension).

- **Message Status Tracking:** Shows sent, delivered, and read status.
- **Multimedia Sharing:** Supports sending images, files, and other attachments.
- **Message Management:** Users can delete, forward, or search messages.
- **Real-Time Synchronization:** Ensures live message updates using WebSockets or similar technologies.

5.4 Schema Design

• User Schema

```
javascript CopyEdit const mongoose = require('mongoose');
```

```
const UserSchema = new mongoose.Schema({  userId:
mongoose.Schema.Types.ObjectId,  username: { type: String,
required: true, unique: true },  email: { type: String, required:
true, unique: true },  password: { type: String, required: true },
status: { type: String, default: 'offline' }, // online, offline,
invisible  authenticationTime: Date,  userKey: String,  ip:
String,  port: String,  randomStatus: String,  createdAt: { type:
Date, default: Date.now }
});
```

```
module.exports = mongoose.model('User', UserSchema);
```

- **Message Schema** javascript

```
CopyEdit    const MessageSchema
=          new
mongoose.Schema({
                                messageId:
mongoose.Schema.Types.ObjectId,    fromUserId:  {  type:
mongoose.Schema.Types.ObjectId,  ref:  'User'  },    toUserId:  {  type:
mongoose.Schema.Types.ObjectId, ref: 'User' },  messageText: {
type: String, required: true },  sentDT: { type: Date, default:
Date.now },  read: { type: Boolean, default: false },  readDT: Date
});
```

```
module.exports = mongoose.model('Message', MessageSchema);
```

- **Friend Schema** javascript CopyEdit const

```
FriendSchema = new
mongoose.Schema({
                                id:
mongoose.Schema.Types.ObjectId,  providerId:
{ type: mongoose.Schema.Types.ObjectId, ref:
'User' },  requestId: { type:
mongoose.Schema.Types.ObjectId, ref: 'User' },
status: { type: String, default: 'pending' } //
pending, accepted, rejected
});
```

```
module.exports = mongoose.model('Friend', FriendSchema);
```

• **Admin Schema (Optional)** javascript

```
CopyEdit    const AdminSchema    =    new
mongoose.Schema({                adminId:
mongoose.Schema.Types.ObjectId,
fullName: String,                email: String,
username: String, password: String
});
```

```
module.exports = mongoose.model('Admin', AdminSchema);
```

CHAPTER 6

PROJECT OUTCOME

6.1 Implementation Approaches

The chat application was built using a bottom-up development approach, beginning with the core backend components (MongoDB + Express) and gradually layering on the frontend (React).

1. Backend-First Development

Data	Model	Design:
-------------	--------------	----------------

Initial development focused on designing robust schemas for core entities:

- **User:** Contains authentication and profile information like username, email, password, IP, status, and login metadata.
- **Message:** Stores information about messages exchanged between users including sender, receiver, message text, timestamps, and read status.
- **Friendship:** Tracks friend relationships using request/provider IDs.
- **Admin:** Maintains administrative credentials and privileges for monitoring the system.

API Development using Express.js:

- User registration, login, and status updates.
- Friend request handling (send, accept, reject).

- Real-time and stored messaging (send, fetch).
- Admin functionalities like user monitoring.

Security Layer:

- Passwords hashed using **bcrypt**.
- JWT tokens used to handle secure authentication sessions.
- Middleware protects routes requiring user verification (e.g., sending messages or accessing chat history).

2. Modular API Structure

The backend is broken into modules to ensure scalability:

User Routes: Manage user registration, login, and status update APIs.

- **Message Routes:** Handle all CRUD operations related to messaging.
- **Friend Routes:** Manage sending, receiving, and approving friend requests.
- **Notification Routes** (optional extension): Push notifications about message receipts, new friend requests, etc.
- **Middleware:** Token validation, role checks (e.g., admin-only routes), and error handlers.

3. Frontend Layering

Once backend APIs were stable, the frontend (React) was developed:

- **Component-Based UI:** Login/Register, ChatWindow, ContactsList, Notifications, AdminPanel.

- **State Management:** React Context API or Redux for managing user authentication, active chats, notifications, and unread message states.
- **Mobile-First Design:** Ensures smooth UI across screen sizes—responsive chat layout, swipable contact panels.

6.2 Coding Details and Code Efficiency

1. Functional React Components

React Hooks (`useState`, `useEffect`, `useContext`) enabled efficient and maintainable components:

- **ChatWindow** for live conversations
- **MessageBubble** components for individual messages
- **OnlineStatusIndicator** for showing user presence

2. Optimized API Responses

- Minimal JSON payloads for faster rendering and low network usage.
- MongoDB `.lean()` used for lightweight object retrieval.
- **Indexing** on `userID`, message timestamps, and friendship status boosts performance.
- **Pagination & Filtering:** Used when loading older messages or notifications.

3. Error Handling

- Unified error responses (e.g., `{status: 403, message: "Unauthorized"}`).

Clear UI feedback for common issues like login failure, friend request errors, or network issues.

4. Security Measures

- Bcrypt for hashing user and admin passwords.
- JWT authentication guards all sensitive routes.
- Admin routes protected with additional access control.

5. Preventive Security Policies

- **CORS** restrictions to allow only trusted domains.
- **Input Sanitization** to block XSS or CSRF attacks.
- **Rate limiting** (optional): To protect APIs from abuse or spam.

6.3 Testing Approaches

1. Manual Testing

- **UI Flow Testing:** User registration → login → send message → receive → logout
- **Device Compatibility:** All major flows tested on mobile and desktop.
- **Notification Testing:** Push alerts for messages and friend requests verified.

2. API Testing (Postman)

- Tested all routes under user, messages, and admin modules.

- Verified token-based access to protected endpoints.
- Simulated:
 - o Invalid logins
 - o Duplicate usernames/emails
 - o Sending messages to unconnected users

3. Unit Testing & Automation (Future Scope)

- **Jest/Mocha** can be used for backend logic (e.g., message storage, friendship logic).
- **React Testing Library** for frontend UI components.
- **CI/CD pipeline** can be introduced for auto-deployment with tests on GitHub Actions or Jenkins.

6.4 Design Test Cases

User Module

Test Case ID	Test Scenario	Test Steps	Expected Result	Status
TC_U001	User Registration with valid input	Enter valid name, email, password and submit	User is successfully registered and redirected to login page	Pass

TC_U002	Registration with existing email	Try registering with an already registered email	Error message: "Email already exists"	Pass
TC_U003	Login with correct credentials	Enter valid email and password	User logged in and redirected to chat dashboard	Pass
TC_U004	Login with incorrect password	Enter valid email but wrong password	Error message: "Invalid credentials"	Pass
TC_U005	Logout functionality	Click logout button	User is logged out and redirected to login screen	Pass

Messaging Module

Test Case ID	Test Scenario	Test Steps	Expected Result	Status
TC_M001	Send message to friend	Open chat window and send a message	Message is shown in the chat window and stored in DB	Pass

TC_M002	Receive message in real-time	Another user sends message to current user	Message appears instantly without refresh (socket.io tested)	Pass
TC_M003	Attempt to message without authentication	Send message without JWT token	Response: Unauthorized (401)	Pass
TC_M004	Message history loading	Scroll up in a chat window	Older messages are loaded via pagination	Pass
TC_M005	Read receipt update	Open message thread	Messages get marked as "read" and update receiver status	Pass

Notification Module

Test Case ID	Test Scenario	Test Steps	Expected Result	Status
TC_N001	New message notification	User receives new message when chat is closed	Notification badge appears in chat list	Pass

TC_N002	Friend request alert	Send friend request from user A to B	User B gets notification popup or badge	Pass
TC_N003	Read notification removal	Click on notification icon	Notifications are cleared/marked as read	Pass
TC_N004	Notification for login status	User comes online	Friend sees "online" status update	Pass

Security and Validation Cases

Test Case ID	Test Scenario	Test Steps	Expected Result	Status
TC_S001	Register without email	Leave email field empty	Error: "Email is required"	Pass
TC_S002	Password stored securely	Inspect DB after signup	Password must be hashed (not plain text)	Pass
TC_S003	Token expiration	Wait for JWT token to expire and refresh page	User is logged out automatically	Pass

TC_S004	to Direct access protected route	Enter /chat URL directly without logging in	Redirect to login page	Pass
---------	--	--	------------------------	------

6.5 Modifications and Improvements

After the initial deployment of the chat application, multiple rounds of user feedback, testing, and performance evaluation led to a series of functional and visual enhancements aimed at improving usability, security, and performance.

Message Filtering and Sorting Enhancements:

- **Chat Search Functionality:**

- o Users can now **search chat messages** by keywords in private and group chats.
- o **Date-based filters** allow users to view conversations from a specific time range.

- **Recent Chats Sorting:**

- o Chats are auto-sorted based on most recent message activity.

Error Messaging and Validation:

- **Form-Level Validation:**

- o Login, registration, and message input forms now include proper frontend and backend validations (e.g., required fields, invalid formats).

- **Backend Error Handling:**

- o Global error handlers were implemented in the backend to catch database errors, authentication failures, and return formatted error responses (e.g., 401 Unauthorized, 500 Internal Error).

User Experience (UX)/User Interface (UI) Enhancements:

- **Toast Notifications:**

- o Feedback notifications appear for actions like login success, message delivery failure, friend request sent, etc.

- **Loading Indicators:**

- o Spinners and skeleton loaders are shown during API calls and real-time socket delays.

- **Mobile-First Layout:**

- o Collapsible side menus and responsive chat windows were added for a seamless mobile experience.

Performance Optimization:

- **Lazy Loading for Chats:**

- o Messages are loaded in chunks (pagination) when the user scrolls up, reducing initial load time.

- **Socket Connection Optimization:**

- o Socket events were refactored to minimize bandwidth usage by **emitting only essential data**.

- **Backend Improvements:**

- o Caching frequently accessed data (like user lists or chat previews) using inmemory stores (e.g., Redis) to reduce DB load.

6.6 Implementation of Code:

```
import React, { useState } from "react"; import chatIcon from
"./assets/comment.png"; import toast from "react-hot-toast";
import { createRoomApi, joinChatApi } from
"./services/RoomService"; import useChatContext from
"./context/ChatContext"; import { useNavigate } from "react-
router"; const JoinCreateChat = () => { const [detail, setDetail] =
useState({ roomId: "", userName: "",
});

const { roomId, userName, setRoomId, setCurrentUser, setConnected } =
useChatContext(); const navigate = useNavigate(); function
handleFormInputChange(event) { setDetail({
...detail,

[event.target.name]: event.target.value,

});

}

function validateForm() { if (detail.roomId
=== "" || detail.userName === "") {
toast.error("Invalid Input !!"); return false;
} return true; } async function joinChat()
{ if (validateForm()) { //join chat try {
```

```

const room = await
joinChatApi(detail.roomId);

toast.success("joined..");
setCurrentUser(detail.userName);
setRoomId(room.roomId);
    setConnected(true);
navigate("/chat"); } catch (error) { if

(error.status      ==      400)      {

toast.error(error.response.data);

21

22

    } else {      toast.error("Error in joining
room");
    }

    console.log(error);

    }

    }

    }

    }

    async function createRoom() {
if (validateForm()) {      //create
room      console.log(detail);
// call api to create room on
backend

    try {

```



```

        const response = await createRoomApi(detail.roomId);        console.log(response);
toast.success("Room Created
Successfully !!");

        //join    the    room    setCurrentUser(detail.userName);
setRoomId(response.roomId);    setConnected(true);

        navigate("/chat");

        //forward to chat page...    } catch

(error) {    console.log(error);    if

(error.status    ==    400)    {

toast.error("Room already exists !!");
} else {    toast("Error in creating
room");

    }

    }

    }

    }

return (

<div className="min-h-screen flex items-center justify-center ">

    <div className="p-10 dark:border-gray-700 border w-full flex flex-col gap-5 max-
w-md rounded dark:bg-gray-900 shadow">

```

```

<div>

  <img src={chatIcon} className="w-24 mx-auto" />

</div>

```

```

<h1 className="text-2xl font-semibold text-center ">

  Join Room / Create Room ..

</h1>

```

23

```

{/* name div */}

<div className="">

  <label htmlFor="name" className="block font-medium mb-2">

    Your name

  </label>

  <input          onChange={handleFormInputChange}
value={detail.userName}      type="text"          id="name"
name="userName"              placeholder="Enter the name"
className="wfull dark:bg-gray-600 px-4 py-2 border dark:border-gray-600
rounded-full focus:outline-none focus:ring-2 focus:ring-blue-500"

  />

</div>

{/* room id div */}

<div className="">

```

```

<label htmlFor="name" className="block font-medium mb-2">

    Room ID / New Room ID

    </label>
    <input
        name="roomId"
        onChange={handleFormInputChange}
        value={detail.roomId}
        type="text"
        id="name"
        placeholder="Enter the room id"
        className="w-full dark:bg-gray-600 px-4 py-2 border dark:border-gray-600 rounded-
        full focus:outline-none focus:ring-2 focus:ring-blue-500"
    />

</div>

{/* button */}

<div className="flex justify-center gap-2 mt-4">

    <button
        onClick={joinChat}
        className="px-3 py-2 dark:bgblue-
        500 hover:dark:bg-blue-800 rounded-full"
    >

        Join Room

    </button>
    <button
        onClick={createRoom}
        className="px-
        3 py-2 dark:bg-orange-500 hover:dark:bg-orange-800 rounded-full"
    >

        Create Room

    </button>

</div>

</div>

</div>

); };

```

```

export default JoinCreateChat; Backend code package
com.substring.chat.chat_app_backend.Controller; import
java.util.List; import org.springframework.http.HttpStatus; import
org.springframework.http.ResponseEntity; import
org.springframework.web.bind.annotation.CrossOrigin; import
org.springframework.web.bind.annotation.GetMapping; import
org.springframework.web.bind.annotation.PathVariable; import
org.springframework.web.bind.annotation.PostMapping; import
org.springframework.web.bind.annotation.RequestBody; import
org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RequestParam; import
org.springframework.web.bind.annotation.RestController; import
com.substring.chat.chat_app_backend.entities.Message; import
com.substring.chat.chat_app_backend.entities.Room; import
com.substring.chat.chat_app_backend.repository.RoomRepo;
@RestController

@RequestMapping("/api/v1/rooms")
@CrossOrigin("http://localhost:5173")
) public class RoomController {

private RoomRepo roomRepo; public

RoomController(RoomRepo roomRepo) {
this.roomRepo=roomRepo;
}

//Create room @PostMapping

public ResponseEntity<?> createRoom(@RequestBody String roomId)
{

if(roomRepo.findByRoomId(roomId)!=null) { return

```

```
ResponseEntity.badRequest().body("Room already exists"); 24
```

```
25
```

```
}
```

```
//create new room
```

```
Room room = new Room();
```

```
room.setRoomId(roomId);
```

```
Room savedRoom = roomRepo.save(room); return
```

```
ResponseEntity.status(HttpStatus.CREATED).body(room);
```

```
//return ;
```

```
}
```

```
//get message
```

```
@GetMapping("/{roomId}") public ResponseEntity<?>
```

```
joinRoom(@PathVariable String roomId){ Room room =
```

```
roomRepo.findByRoomId(roomId); if(room==null) { return
```

```
ResponseEntity.badRequest().body("Room not found!!");
```

```
}
```

```

return ResponseEntity.ok(room);

}

@GetMapping("/{roomId}/messages")                public
                ResponseEntity<List<Message>>

getMessages(@PathVariable String roomId,

    @RequestParam (value = "page",defaultValue = "0",required = false)int page,

    @RequestParam (value = "size",defaultValue = "0",required = false)int size){
Room room = roomRepo.findByRoomId(roomId);    if(room == null) {
return
ResponseEntity.badRequest().build();

}

//get message

//pagination

List<Message> messages =
room.getMessages();  int start = Math.max(0,
messages.size()-(page+1)*size);  int end =
Math.min(messages.size(), start + size);

List<Message> paginateMessages = messages.subList(start, end);  return
ResponseEntity.ok(messages);

}

}

package com.substring.chat.chat_app_backend.Controller;

```

```
import java.time.LocalDateTime;
```

```
import org.springframework.messaging.handler.annotation.DestinationVariable; import
org.springframework.messaging.handler.annotation.MessageMapping; import
org.springframework.messaging.handler.annotation.SendTo;      import
org.springframework.stereotype.Controller; import
org.springframework.web.bind.annotation.CrossOrigin;      import
org.springframework.web.bind.annotation.RequestBody;
```

```
import com.substring.chat.chat_app_backend.entities.Message; import
com.substring.chat.chat_app_backend.entities.Room;      import
com.substring.chat.chat_app_backend.payload.MessageRequest; //import
com.substring.chat.chat_app_backend.payload.MessageRequest; import
com.substring.chat.chat_app_backend.repository.RoomRepo; @Controller
```

```
@CrossOrigin("http://localhost:5173") public class
```

```
ChatController {
```

```
    private RoomRepo roomRepo;
```

```
    public ChatController(RoomRepo roomRepo) {      this.roomRepo=roomRepo;
    }
```

```
    @MessageMapping("/SendMessage/{roomId}")
```

```

@SendTo("/topic/room/{roomId}") public Message
    sendMessage(@DestinationVariable String roomId, @RequestBody
MessageRequest request) {

    Room room = roomRepo.findByRoomId(request.getRoomId());

    Message message = new Message();
message.setContent(request.getContent());    message.setSender(request.getSender());
message.setTimestamp(LocalDateTime.now());        if(room!=null)
{

    room.getMessages().add(message);    roomRepo.save(room);

    }else {        throw new RuntimeException("room not
found!!"+ roomId);    }

    return
message;

    }

}

package com.substring.chat.chat_app_backend.Controller;

import java.util.List;
import org.springframework.http.HttpStatus;

```



```

import      org.springframework.http.ResponseEntity;      import
org.springframework.web.bind.annotation.CrossOrigin;      import
org.springframework.web.bind.annotation.GetMapping;      import
org.springframework.web.bind.annotation.PathVariable;      import
org.springframework.web.bind.annotation.PostMapping;      import
org.springframework.web.bind.annotation.RequestBody;      import
org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RequestParam; import
org.springframework.web.bind.annotation.RestController;

```

```

import
com.substring.chat.chat_app_backend.entities.Message;
import com.substring.chat.chat_app_backend.entities.Room;
import
com.substring.chat.chat_app_backend.repository.RoomRepo;

```

```
@RestController
```

```
@RequestMapping("/api/v1/rooms")
```

```
@CrossOrigin("http://localhost:5173")
```

```
) public class RoomController {
```

```
private
```

```
RoomRepo roomRepo;
```

```

    public RoomController(RoomRepo roomRepo) {    this.roomRepo=roomRepo;

    }

```

```

//Create room

@PostMapping public ResponseEntity<?> createRoom(@RequestBody String
roomId) { if(roomRepo.findByRoomId(roomId)!=null) { return
ResponseEntity.badRequest().body("Room already exists");
}

//create new room

Room room = new Room(); room.setRoomId(roomId);

Room savedRoom = roomRepo.save(room); return

ResponseEntity.status(HttpStatus.CREATED).body(room);

//return ;

}

//get message

@GetMapping("/{roomId}")

public ResponseEntity<?> joinRoom(@PathVariable String roomId){

```

```

        Room room = roomRepo.findByRoomId(roomId);

if(room==null) {

        return ResponseEntity.badRequest().body("Room not found!!");

    }

    return ResponseEntity.ok(room);

}

```

```

        @GetMapping("/{roomId}/messages")

public ResponseEntity<List<Message>> getMessages(@PathVariable String
roomId,

        @RequestParam (value = "page",defaultValue = "0",required = false)int page,

        @RequestParam (value = "size",defaultValue = "0",required = false)int size){

Room room = roomRepo.findByRoomId(roomId);
if(room == null) {

        return ResponseEntity.badRequest().build();

    }

    //get message

    //pagination

    List<Message> messages = room.getMessages();  int start =
Math.max(0, messages.size()-(page+1)*size);      int      end      =

```

```

Math.min(messages.size(), start + size);

        List<Message> paginateMessages = messages.subList(start, end);
        return ResponseEntity.ok(messages);

    }

```

```

} package com.substring.chat.chat_app_backend.entities;

```

```

import java.util.ArrayList; import
java.util.List;

```

```

import org.springframework.data.annotation.Id;
import
org.springframework.data.mongodb.core.mapping.Do
cument; import lombok.AllArgsConstructor; import
lombok.Getter; import lombok.NoArgsConstructor;
import lombok.Setter;

```

```

@Document(collection = "chats")

```

```
@NoArgsConstructor
```

```
r
```

```
@AllArgsConstructor
```

```
r public class Room
```

```
{
```

```
    @Id
```

```
    private String id;    private
```

```
    String roomId;
```

```
    private List<Message> messages = new ArrayList<>();
```

```
    public String getId() {
```

```
        return id;
```

```
    }
```

```
    public void setId(String id) {
```

```
        this.id = id; }    public String
```

```
    getRoomId() { return roomId;
```

```
    }
```

```
    public void setRoomId(String roomId) {
```

```

        this.roomId = roomId;

    }

    public List<Message> getMessages() {        return
messages;
    }

    public void setMessages(List<Message> messages) {        this.messages
= messages;
    }

} package com.api.services;

import java.io.IOException; import java.nio.file.Files; import
java.nio.file.Path; import java.nio.file.Paths; import java.util.List;
import java.util.Optional; import
org.springframework.beans.factory.annotation.Autowired; import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
; import
org.springframework.security.crypto.password.PasswordEncoder;
import

org.springframework.stereotype.Service;                                import

```

```
org.springframework.web.multipart.MultipartFile
; import com.api.entities.User; import
com.api.exceptions.CustomNotFoundException;
import com.api.repository.UserRepo;
```

```
@Service public class UserServiceImpl implements
UserService{
```

```
    @Autowired
```

```
    private UserRepo userRepo;
```

```
    //AllUsersGet
```

```
    @Override
```

```
    public List<User> getAllUsers() {
```

```
        return userRepo.findAll();
```

```
    }
```

```
    //getById    @Override    public    User
```

```
    getById(Integer userId) {
```

```
        Optional<User> findById = userRepo.findById(userId);
```

```
        if(findById.equals(findById)) {
```

```
            return findById.get();
```

```

    }

    return null;

}

//insertUser @Override public

User addUser(User user) {

    return userRepo.save(user);

}

//update user

@Override

public User updateUser(Integer userId, User user) {

    User existingUser = userRepo.findById(userId)

        .orElseThrow(() -> new CustomNotFoundException("User not found
with ID: " + userId));
        existingUser.setName(user.getName());
existingUser.setAge(user.getAge());
        existingUser.setEmail(user.getEmail());
        existingUser.setDateOfBirth(user.getDateOfBirth());
existingUser.setTimeOfBirth(user.getTimeOfBirth());
existingUser.setContactNumber(user.getContactNumber());
existingUser.setPlace(user.getPlace());
existingUser.setGender(user.getGender());

```



```

        // Password ko update karne se pehle validate kar sakte hain
        if (user.getPassword() != null && !user.getPassword().isEmpty()) {
            existingUser.setPassword(user.getPassword());
        }

        return userRepo.save(existingUser);
    }

    //delete user

    @Override

    public String deleteById(Integer userId) {

        if(userRepo.existsById(userId)) { userRepo.deleteById(userId); return
        "Successfully Delete";
        }

        else {

            throw new CustomNotFoundException("User not found with ID: " + userId);

        }

    }

    // Given an email, fetch the user from the database.

    // If user not found, throw a CustomNotFoundException with a proper message.

    @Override

    public User getUserByEmail(String email) {

```

```

        return
        userRepo.findByEmail(email)
            .orElseThrow(() -> new CustomNotFoundException("User not found
with email: " + email));

    }

```

//login user

```
public User loginUser(String email, String password) {
```

```
    return userRepo.findByEmail(email)
```

```
        .filter(user -> user.getPassword().equals(password))
```

```
        .orElse(null);
```

```
}
```

//change password @Override

```
public User changePassword(String email, String oldPassword, String
newPassword) {
```

```

        User user = getUserByEmail(email);                if
        (user.getPassword().equals(oldPassword)) {
        user.setPassword(newPassword);                return userRepo.save(user);
        // returns updated User

```

```
        } else {
```

```
            return null; // or throw new IllegalArgumentException("Invalid old password");
```

```

        }

    }

    @Override

    public void uploadUserImage(Integer userId, MultipartFile imageFile) throws
    IOException {

        User user = userRepo.findById(userId)

        .orElseThrow(() -> new RuntimeException("User not found with ID: " +
        userId));

        // Upload folder

        String uploadDir = "uploads/";

        Files.createDirectories(Paths.get(uploadDir));

        // Save file

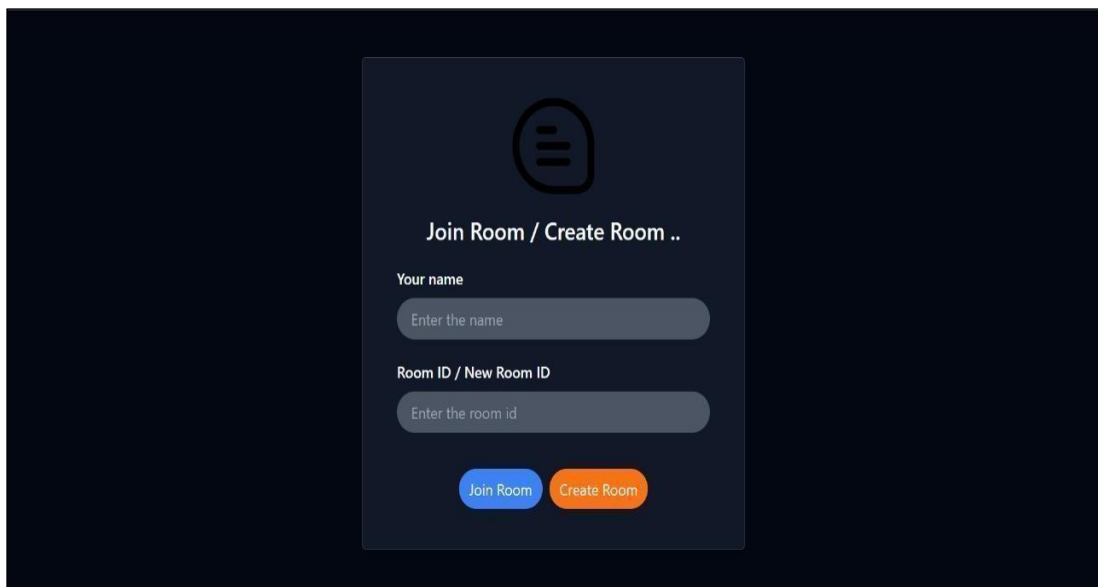
        String fileName = imageFile.getOriginalFilename();


        Path filePath = Paths.get(uploadDir + fileName);

        Files.write(filePath, imageFile.getBytes());

```

```
        // Update image name
        user.setImageName(fileName);
        userRepo.save(user);
    }
}
```



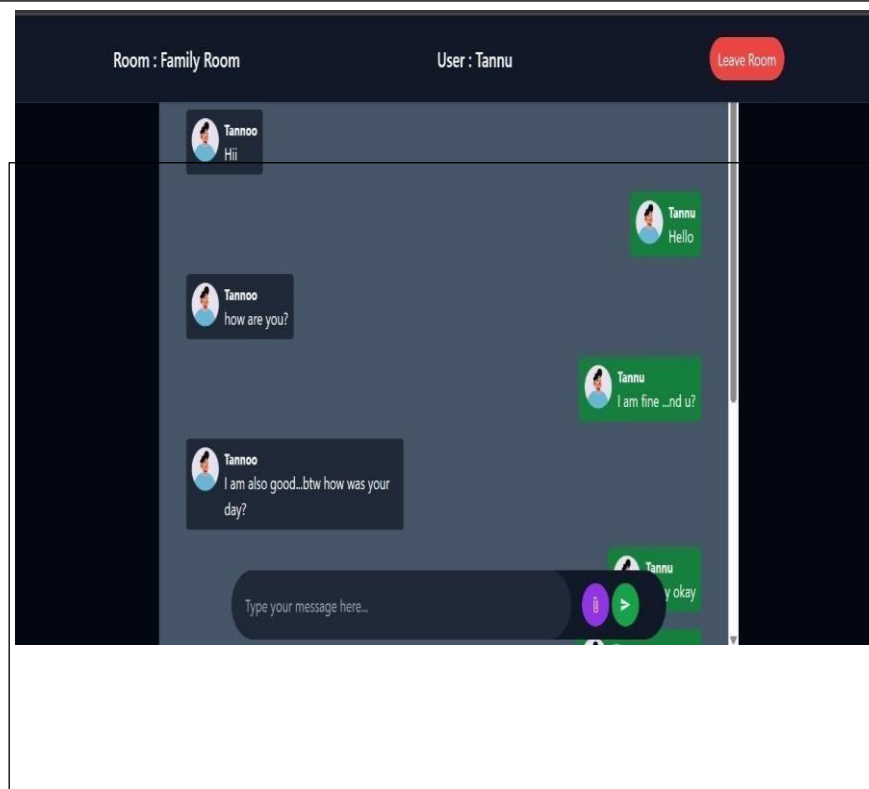
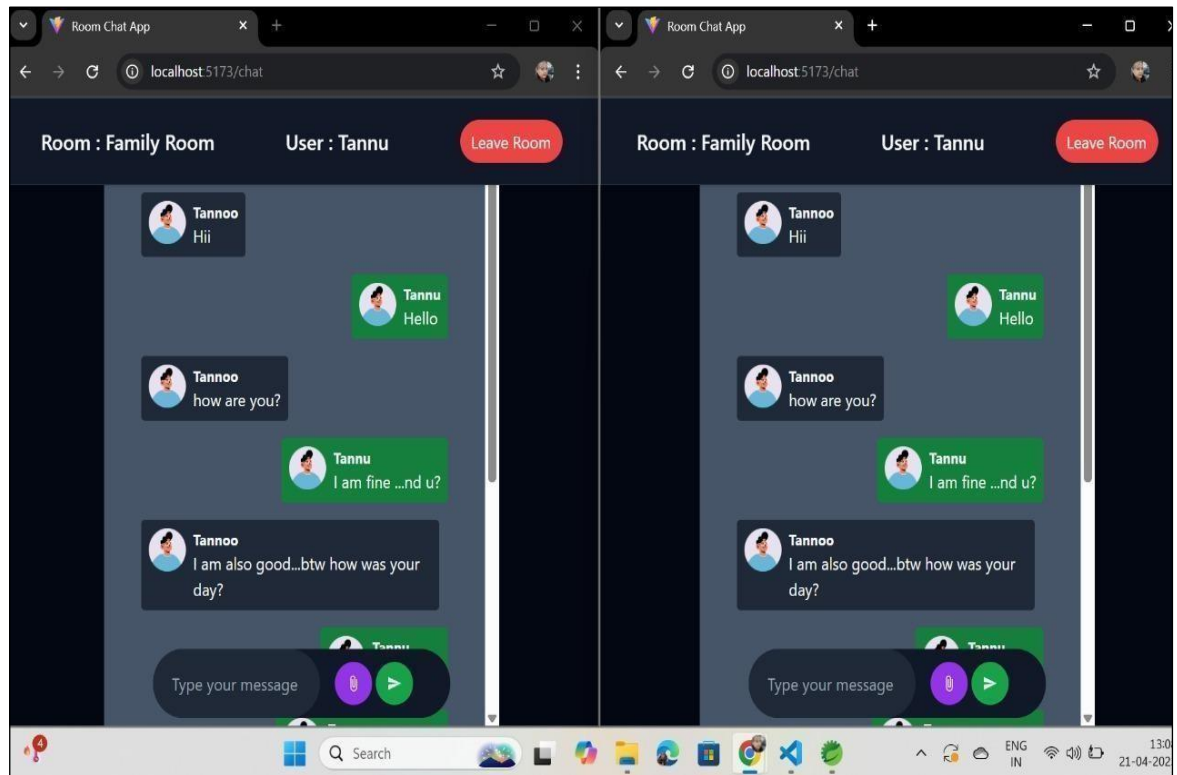


Join Room / Create Room ..

Your name

Room ID / New Room ID

Join RoomCreate Room



CHAPTER 7

REFERENCES/BIBLIOGRAPHY

1. Mario Casciaro & Luciano Mammino, *"Node.js Design Patterns"*, 2nd Edition, Packt Publishing, 2020.
2. Douglas Crockford, *"JavaScript: The Good Parts"*, O'Reilly Media, 2008.
3. Alex Banks & Eve Porcello, *"Learning React"*, 2nd Edition, O'Reilly Media, 2020.
4. Seema J. Shah and Pratik K. Shah, *"Real-time Chat Application Using WebSockets"*, International Journal of Computer Applications, Vol. 177, No. 9, pp. 1-5, November 2019, DOI: 10.5120/ijca2019919648.
5. Amit P. Shelar, Manjusha Pandey, *"Enhancing Security in Web-Based Chat Applications"*, Springer Advances in Computing and Data Sciences, pp. 453-463, 2021, DOI: 10.1007/978-981-16-3311-6_37.
6. **React.js Documentation**, Available at: <https://react.dev/> (Accessed: March 2025).
7. **Node.js Documentation**, Available at: <https://nodejs.org/en/docs> (Accessed: March

2025).

8. **Express.js Documentation**, Available at:

<https://expressjs.com/en/starter/installing.html>

(Accessed: March 2025).

9. **WebSockets (Socket.IO) Documentation**, Available at:

<https://socket.io/docs/>

(Accessed: March 2025).

10. **Firebase Real-time Database Documentation**, Available

at: <https://firebase.google.com/docs> (Accessed: March 2025).

11. **MongoDB Documentation**, Available at: <https://www.mongodb.com/docs/>

(Accessed:

March 2025).

12. **AWS Deployment Guide**, Available at: <https://aws.amazon.com/getting-started/>

(Accessed: March 2025).

