# Web Chat application

*A*

***Project Report***

*submitted in partial fulfillment of the*

*requirements for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE & ENGINEERING**

**by**


| Name | Roll No. |
|---|---|
| SHIVANSHU | R2142221234 |
| MANAN KUMAR | R2142220740 |
| TUSHAR KUKREJA | R2142221272 |
| SURYANSH CHAUHAN | R2142221191 |


*under the guidance of*

**MR NARENDAR KUMAR DEVANGAN**

16/12/2024


16/12


**School of Computer Science**

**University of Petroleum & Energy Studies**

**Bidholi, Via Prem Nagar, Dehradun, Uttarakhand**


**December – 2024**

## CANDIDATE'S DECLARATION

I/We hereby certify that the project work entitled **Web chat application** in partial fulfilment of the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING with specialization in Artificial intelligence and Machine learning(AI&ML) and submitted to the Department of Systemics, School of Computer Science, University of Petroleum & Energy Studies, Dehradun, is an authentic record of my/ our work carried out during a period from **August, 2024** to **December, 2024** under the supervision of **Mr. NARENDAR KUMAR DEVANGAN .**

The matter presented in this project has not been submitted by me/ us for the award of any other degree of this or any other University.

**(SHIVANSHU)**
**Roll No.** R2142221234

**(MANAN KUMAR)**
**Roll No.** R2142220740

**(TUSHAR KUKREJA)**
**Roll No.** R2142221272

**(SURYANSH CHAUHAN)**
**Roll No.** R2142221191

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

Date: 16/12/ 2024

**(Mr. NARENDAR KUMAR DEVANGAN)**
Project Guide

# ACKNOWLEDGEMENT

We wish to express our deep gratitude to our guide **Name**, for all advice, encouragement and constant support he/she has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.

We sincerely thanks to our respected **Name of HoD, Head Department of** Mr. Anil Kumar, for his great support in doing our project in Web chat application

We are also grateful to Dean SoCS UPES for giving us the necessary facilities to carry out our project work successfully. We also thanks to our Course Coordinator, (NAME) and our Activity Coordinator (NAME) for providing timely support and information during the completion of this project.

We would like to thank all our **friends** for their help and constructive criticism during our project work. Finally, we have no words to express our sincere gratitude to our **parents** who have shown us this world and for every support they have given us.

| | | | | |
|---|---|---|---|---|
| **Name** | **Shivanshu** | **Suryansh chauhan** | **Manan Kumar** | **Tushar Kukreja** |
| **Roll No.** | **R2142221234** | **R2142221191** | **R2142220740** | **R2142221272** |

# ABSTRACT

This project involves the development of a secure, real-time chat application that enables users to communicate through text messages. The application uses a combination of Java Swing for the graphical user interface (GUI), Node.js for handling the server-side logic, and MongoDB for storing user data and messages. The goal of the project is to create an efficient and scalable messaging platform that supports features such as user authentication, real-time communication, and secure data storage.

The system is designed using a client-server architecture. The client, developed in Java Swing, offers an intuitive interface where users can register, log in, and participate in live conversations. The server, built with Node.js, handles client requests, manages WebSocket connections for real-time communication, and processes interactions with the MongoDB database. MongoDB is used as the backend database, chosen for its ability to handle large volumes of unstructured data and its scalability. It stores user credentials and chat messages, ensuring that data persists even after the application is closed.

User authentication is a key component of the application. During registration, users create accounts, which are securely stored with hashed passwords using bcrypt to enhance security. Once logged in, users are connected to the server via WebSocket, enabling them to send and receive messages in real-time. The server is responsible for managing these WebSocket connections, broadcasting messages to the correct recipients, and ensuring the integrity of data being transmitted.

The application's real-time communication is powered by WebSockets, which allow for two-way communication between the client and server. This eliminates the need for continuous page refreshes, enabling instant message delivery. MongoDB ensures that each message, along with relevant metadata like the sender, receiver, and timestamp, is securely stored and retrievable for future reference.

The current version of the chat application primarily supports text-based messaging, but there are plans for future enhancements. These include the ability to send multimedia messages (such as images or videos), advanced administrative features for better user management, and improved security measures. Additionally, the system's scalability is designed to support a growing number of users, making the platform robust and capable of handling increasing traffic.

Overall, this project demonstrates how modern technologies like Node.js, MongoDB, and Java Swing can be integrated to build a feature-rich, real-time messaging application. It serves as a foundation for further improvements and provides a secure and reliable messaging solution that can be expanded in the future to meet the needs of a larger user base.

# TABLE OF CONTENTS

# 1. Introduction

## 1.1. History

The development of this chat application with login functionality was inspired by the need for a secure, scalable, and real-time communication system. Traditional chat applications often rely on centralized services, but this system leverages Node.js and MongoDB for flexibility and real-time interactions, empowering users to send and receive messages securely.

## 1.2. Requirement Analysis

This chat system has two primary functionalities:

- **User Authentication**: Ensures that only authorized users can access the system.
- **Real-Time Chat**: Users should be able to send and receive messages instantaneously.

The system must also be scalable, secure, and user-friendly, providing an intuitive interface and smooth user experience.

## 1.3. Main Objective

The main objective of this project is to develop a secure chat system that:

- Allows users to log in.
- Supports real-time messaging.
- Stores data securely in MongoDB.
- Ensures real-time data updates using Node.js and WebSockets.

## 1.4. Sub Objectives

- Integrating MongoDB for secure data storage.
- Developing the client-server architecture using Node.js.
- Implementing front-end features using Java Swing for a seamless chat interface.
- Handling error management and data synchronization between client and server.

## 1.5. System Overview

The system is a real-time chat application consisting of:

- A client-side GUI implemented using Java Swing, enabling users to log in and send messages.
- A server-side Node.js application that handles user authentication, message broadcasting, and data management.
- MongoDB as the database for storing user credentials and messages.

## 2. System Analysis

### 2.1. Existing System

Before the development of this chat system, most systems relied on basic server-client communication using traditional databases, often lacking real-time updates and complex user authentication features.

### 2.2. Motivations for Development

The motivation for developing this system comes from the need to:

- Provide a real-time messaging experience.
- Offer secure user authentication.
- Build a robust and scalable solution using modern technologies like Node.js and MongoDB.

### 2.3. Proposed System

The proposed system will consist of:

- **Client-Side**: A Java Swing interface for login and chat functionalities.
- **Server-Side**: A Node.js server that handles authentication, message broadcasting, and MongoDB interaction.
- **Database**: MongoDB for storing user data and messages.

### 2.4. Modules

#### 2.4.1. User Authentication

- **Login Page**: Users enter credentials to authenticate their identity.
- **Sign-Up Page**: Allows new users to register.
- **Password Security**: Implement hashing for password security using bcrypt.

#### 2.4.2. Real-Time Chat Functionality

- **Socket Communication**: WebSockets for establishing a real-time connection between client and server.
- **Message Broadcasting**: Messages are broadcasted to all connected clients instantly.

#### 2.4.3. Data Storage and Retrieval

- **MongoDB**: Used to store user data and chat messages. Schema designed to support users and their messages efficiently.
- **CRUD Operations**: Operations to create, read, update, and delete messages stored in MongoDB.

#### 2.4.4. Admin Panel

- **User Management**: Admin can view all users, manage permissions, and delete inappropriate messages.

## 3. Design

### 3.1. System Architecture

The system architecture follows a **client-server model**, where the client communicates with the server and the server interacts with the database. This architecture ensures a modular and scalable design, allowing the system to manage multiple users and provide real-time communication.

- **Client**: The client-side application is built using **Java Swing**, a GUI (Graphical User Interface) toolkit that provides a simple and effective way to create desktop applications. The client handles user input for login authentication, message sending, and receiving messages in real-time. It also manages the interface elements like buttons, text fields, and display areas for the chat messages.
- **Server**: The server is developed using **Node.js**, a popular JavaScript runtime built on Chrome's V8 engine. Node.js enables the server to handle concurrent requests efficiently using non-blocking I/O. The server processes incoming requests from clients, authenticates users, handles real-time communication through **WebSocket** connections, and ensures that data is stored and retrieved from the **MongoDB** database.
- **Database**: **MongoDB** is the chosen database for storing data, as it provides a flexible, scalable, and easy-to-manage NoSQL structure. MongoDB stores the user credentials (with hashed passwords) and the messages exchanged between users. It is chosen for its ability to handle large volumes of unstructured data, which is common in real-time chat applications.

#### 3.1.1. Front-End Design

The **front-end** of the application is developed using **Java Swing**, which allows the creation of cross-platform desktop applications. The user interface (UI) is intuitive and provides all the necessary features for the chat system.

- **Login Page**: The login page includes input fields for the **username** and **password**, with buttons for logging in or signing up. This interface communicates with the back-end server to authenticate the user.
- **Chat Interface**: After successful login, the user is presented with the main chat interface, which displays incoming and outgoing messages in a chat window. The chat interface includes:
    - A text area for displaying messages from all users.
    - A text field where the user can type their messages.
    - A send button to send messages.
- **User-Friendly Design**: The UI is designed to be simple and easy to use. The interface is responsive, ensuring that users can easily access all features, such as logging in, chatting, and sending messages, with minimal navigation.

#### 3.1.2. Back-End Design

The **back-end** of the system is built using **Node.js**, which is well-suited for handling real-time applications like chat systems.

- **WebSocket Communication**: One of the core features of the back-end is the use of **WebSockets**, which allows real-time, bidirectional communication between the client and the server. This means that when a user sends a message, the server broadcasts it immediately to the intended recipient without delay, providing a seamless real-time chat experience.

- **Request Handling**: Node.js processes incoming HTTP requests, including user authentication requests (login/sign-up) and message requests. It listens to specific routes (URLs) for different functionalities, such as logging in a user or sending a message.

- **Data Management**: The back-end is responsible for managing the flow of data between the client and the database. For example, when a user sends a message, the back-end processes the message and stores it in the database (MongoDB). Similarly, it fetches messages from the database when a user opens the chat interface.

- **Security**: Node.js is also responsible for handling security tasks, such as password hashing and validation. It uses libraries like **bcrypt** to securely hash user passwords before storing them in MongoDB. Additionally, authentication tokens (like JWT) can be used to ensure that only authorized users access the system.

### 3.1.3. Database Design

The database is designed to be simple, yet efficient, storing all essential data needed for the chat application.

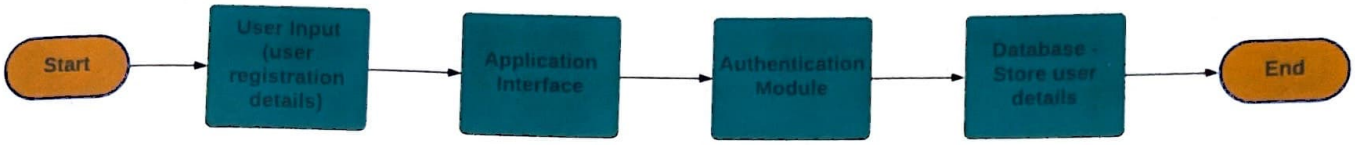
- **Users Collection**: The **Users** collection stores information about the users of the chat application. Each document in this collection represents a user and contains:
  - **Username**: A unique identifier for each user.
  - **Password**: The password is stored in a **hashed** format to ensure security. The password is never stored as plain text.
  - **Profile Details**: This could include the user's display name, email address, and other optional profile information.

- **Messages Collection**: The **Messages** collection stores the chat messages exchanged between users. Each document in this collection represents a message and contains:
  - **Sender**: The username of the person who sent the message.
  - **Receiver**: The username of the person who received the message.
  - **Message Content**: The actual content of the message.
  - **Timestamp**: The time at which the message was sent, allowing the system to display messages in chronological order.

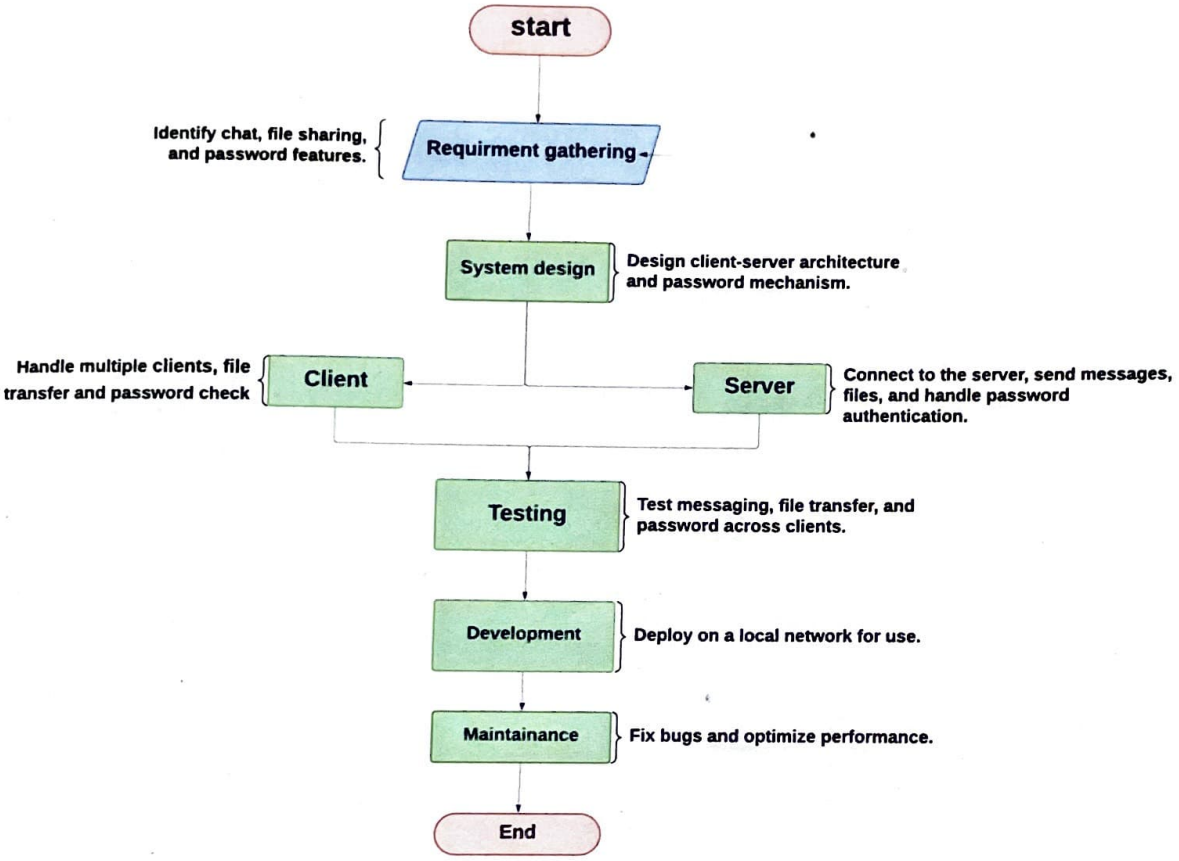
- **Schema**: MongoDB allows for flexible schema designs, so each collection can be easily updated or modified if the project requirements change. In this case, the collections are designed to handle all necessary information for user authentication and message management.

By storing user data and messages in MongoDB, the system can efficiently retrieve and display messages when needed, ensuring smooth communication between users.

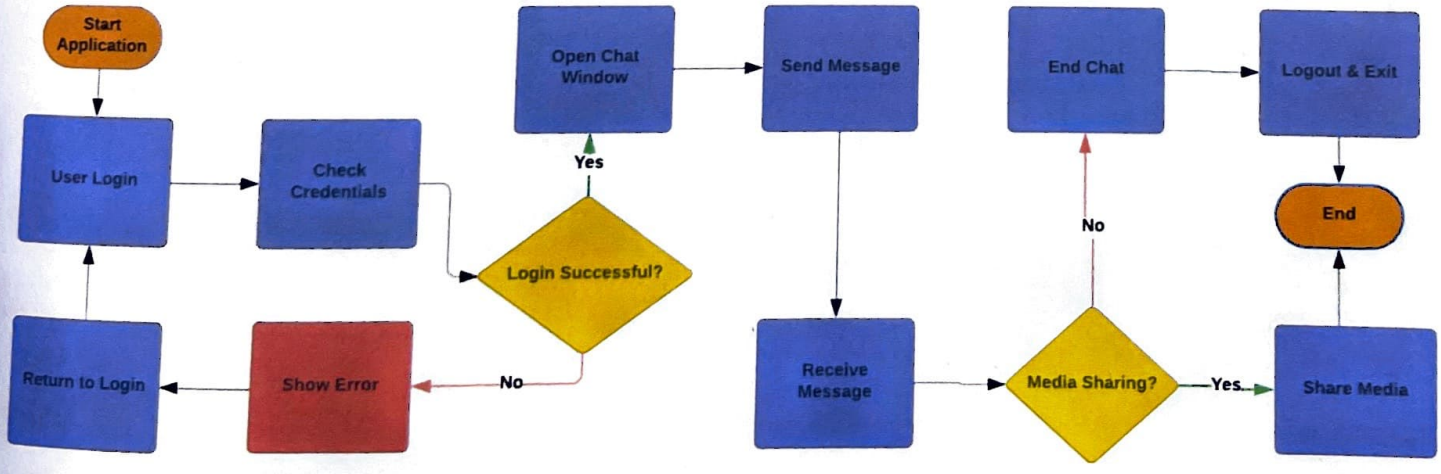
### 3.2 Use case model:



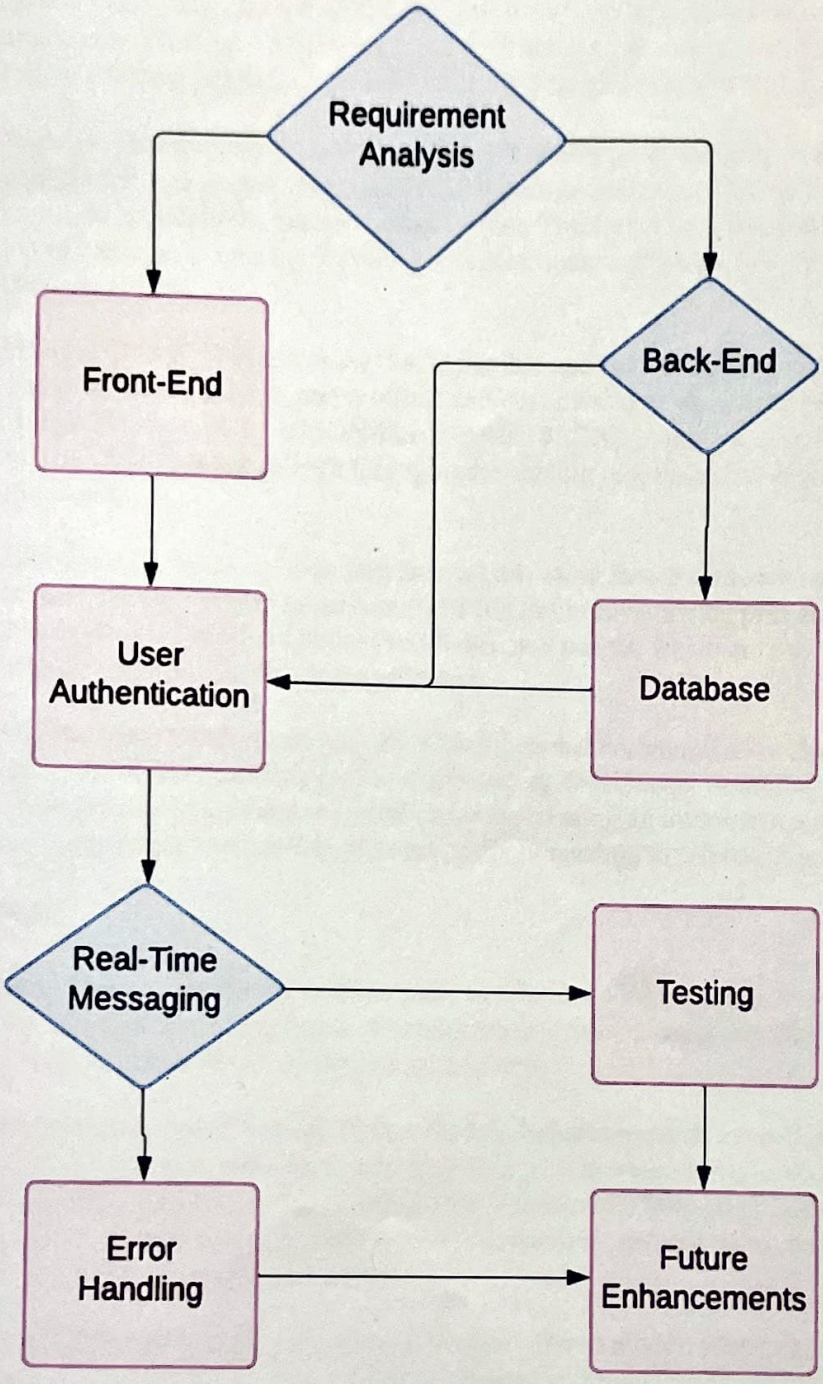

### 3.3 Flow digram:




### 3.4 Activity Diagram:



**3.5 Pert chart:**



## 4. Technology Stack

### 4.1. Overview of Technologies Used

The technology stack for this chat application has been carefully chosen to support the requirements of real-time communication, data storage, and user interface development. Each technology plays a critical role in enabling seamless functionality, high performance, and scalability.

- **Node.js**: This technology is utilized for server-side programming. Node.js is known for its non-blocking, asynchronous, and event-driven architecture, making it highly suitable for real-time applications, such as a chat system. The server uses **Socket.io**, a library built on top of Node.js, to manage WebSocket connections and handle real-time data transmission between clients.

- **MongoDB**: A NoSQL database that is flexible and scalable, MongoDB is used to store user data (such as credentials and profiles) and chat messages. As a document-based database, MongoDB stores data in JSON-like formats (BSON), making it easy to manage and retrieve data quickly. It supports high throughput, enabling real-time chat applications to scale effortlessly.

- **Java Swing**: The client-side interface is built using Java Swing, a widely used framework for developing graphical user interfaces (GUIs) in Java. Swing provides an extensive set of UI components such as buttons, text fields, and panels, which are used to create a user-friendly interface for the chat application.

- **WebSockets**: WebSockets provide a bi-directional communication channel between the client and server, enabling real-time messaging. WebSocket technology ensures that messages can be sent and received instantly, creating an interactive experience where users see messages as soon as they are sent, without needing to refresh or poll the server.

### 4.2. Node.js

**Node.js** is a powerful JavaScript runtime built on Chrome's V8 engine. It's designed to be lightweight and efficient, using a non-blocking, event-driven architecture that's perfect for I/O-intensive applications, such as a real-time chat system.

- **Asynchronous and Non-blocking**: Node.js handles requests asynchronously, meaning that while the server is processing one request (e.g., a message being sent), it can still handle other tasks, such as managing WebSocket connections from other clients. This allows the server to process multiple simultaneous connections without delay, ensuring smooth real-time communication for all users.

- **Event-Driven**: Node.js operates on an event-driven model, where an event (like a user sending a message) triggers specific actions (e.g., broadcasting the message to all connected users). This architecture is especially beneficial for chat systems, where events like sending or receiving messages need to be handled efficiently and in real-time.

- **Socket.io**: The **Socket.io** library is used in the Node.js server to establish WebSocket communication. Socket.io simplifies the creation and management of WebSocket connections and provides a fallback mechanism in case WebSockets are not supported by the client's browser. Socket.io allows for real-time, bi-directional communication between the client and the server, ensuring that messages are instantly delivered and received.

- **Scalability**: Node.js's single-threaded event loop ensures that it can handle a large number of concurrent connections with minimal overhead. This makes Node.js an ideal choice for scalable real-time applications like chat systems, where multiple users might be online and exchanging messages at the same time.

### 4.3. MongoDB

**MongoDB** is a popular NoSQL database that stores data in a flexible, JSON-like format, making it ideal for applications with rapidly changing data structures or high scalability needs. In this chat application, MongoDB is used to store both user data (such as usernames and hashed passwords) and message data (like the sender, receiver, content, and timestamp).

- **Document-Oriented Storage**: MongoDB stores data in collections of documents, rather than rows and columns as in traditional relational databases. Each document is a self-contained unit of data, stored in the BSON (Binary JSON) format. This allows for greater flexibility in how data is stored, making it easier to scale and handle diverse data types such as user profiles, messages, and timestamps.

- **Scalability**: MongoDB is designed to be highly scalable. As the application grows and the number of users increases, MongoDB's sharding and replication features allow the database to scale horizontally. This means that data can be distributed across multiple servers, providing higher throughput and fault tolerance without compromising performance.

- **Efficient Data Retrieval**: MongoDB supports powerful querying capabilities, making it easy to retrieve messages based on different criteria, such as the sender, receiver, or date. In a chat application, this enables the retrieval of messages in real-time, ensuring that when users access a chat, they can see the latest messages instantly.

- **Data Integrity**: MongoDB offers features like data consistency and atomic operations, which are essential for a chat application that needs to ensure that messages are not lost and are properly saved to the database. MongoDB's flexible schema means that the database structure can evolve as the application grows, adding new fields or collections as needed without disrupting the existing functionality.

## 5. Implementation

The implementation phase of the chat application involves putting together all the technologies and frameworks to create a working system. In this section, the implementation details of the core functionalities, including user authentication, real-time communication, database integration, and error handling, are discussed.

### 5.1. User Authentication

User authentication is a critical aspect of any web application, ensuring that users are properly identified before gaining access to the system. In this chat application, the following steps are used for user authentication:

- **Login and Registration**: The application allows users to create an account by registering with a username, email, and password. The registration data is validated both on the client and server side. Once a user registers, they can log in by entering their credentials (username and password) on the login page. The server then checks the provided credentials against the data stored in MongoDB to verify the user's identity. If the credentials match, the user is granted access to the chat system.

- **Password Hashing**: To ensure the security of user passwords, the **bcrypt** library is used to hash passwords before they are stored in the MongoDB database. Hashing ensures that even if the database is compromised, the actual passwords cannot be retrieved. bcrypt is a cryptographic algorithm that transforms the plain-text password into a hashed value, which is stored in the database. When a user attempts to log in, the server hashes the entered password and compares it to the stored hash. If they match, the user is authenticated.

The process of hashing is done in such a way that it's computationally difficult to reverse, offering a layer of protection against brute-force attacks. Furthermore, bcrypt also adds a unique salt to each password, ensuring that even identical passwords will have different hash values.

### 5.2. Real-Time Communication

Real-time communication is the backbone of any chat application, and the system employs **WebSocket** technology to enable this functionality.

- **Socket Connections**: When a user logs in, the client establishes a **WebSocket connection** with the server. This connection remains open, enabling two-way communication between the client and the server without the need to continuously refresh or request data. This allows for **real-time messaging**, where messages are sent and received instantly as they are typed and sent by users.

The WebSocket connection ensures low-latency, bidirectional communication between the client and server. For every new message typed by the user, the client emits a message event to the server, which is then broadcasted to other connected clients in real time. The **socket.io** library is used to manage these WebSocket connections, ensuring stable and efficient communication across multiple users.

- **Real-Time Messaging**: The real-time functionality ensures that when a message is sent by a user, it is immediately broadcast to all other users in the chat room. Additionally, the application allows for one-on-one messaging and group chats, all of which function seamlessly via WebSockets.

### 5.3. MongoDB Integration

MongoDB plays a pivotal role in storing the data of the chat application. The server uses **Mongoose**, an Object Data Modeling (ODM) library for MongoDB and Node.js, to interact with the database. Mongoose provides an easy-to-use interface to perform operations like reading, writing, and updating data in MongoDB.

- **Message Storage**: When a user sends a message, it is saved in the MongoDB database in the **Messages** collection. Each message document includes key details such as the **sender**, **receiver**, **message content**, and **timestamp** of the message. This ensures that all messages are stored in an organized manner, allowing easy retrieval later when the user requests them.
- **User Data Storage**: The **Users** collection in the database stores the user data, including the **username**, **hashed password**, and other profile details. During login, the server retrieves the user's details from MongoDB to authenticate the credentials.
- **Mongoose ORM**: Mongoose simplifies database operations by allowing developers to define **schemas** and **models**. For example, the **Message** schema defines the structure of a message, including fields like sender, receiver, content, and timestamp. Mongoose models are then used to interact with these schemas, making it easier to perform CRUD operations (Create, Read, Update, Delete) on MongoDB.

### 5.4. Error Handling

Proper error handling is an essential part of building a robust and user-friendly application. The chat application implements several layers of error handling to ensure that the system remains stable, and users are provided with meaningful feedback in case of issues.

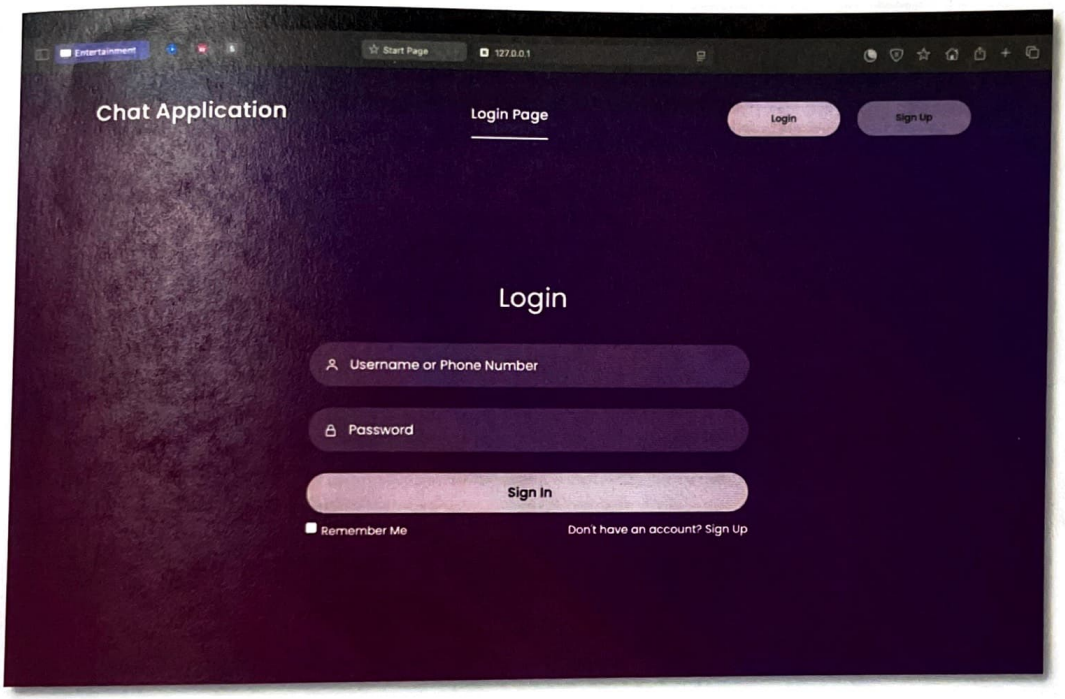
- **Network Issues**: If there are issues with the network, such as a dropped WebSocket connection or failure to connect to MongoDB, the application handles these errors gracefully. For instance, if the connection to MongoDB is lost, the server logs the error and attempts to reconnect automatically. Similarly, if a user's WebSocket connection is unexpectedly terminated, the system can automatically reconnect the user or notify them of the issue.
- **Invalid Data Inputs**: To prevent malicious or malformed data from entering the system, input validation is performed both on the client and server sides. The server checks that the required fields (such as the message content or user credentials) are present and valid. If a user tries to send a message with invalid data (e.g., a blank message or a forbidden character), the server responds with an error message, and the client can display an appropriate message to the user.
- **General Error Handling**: The server includes global error handlers to catch unexpected exceptions or unhandled promise rejections. These errors are logged for debugging purposes, and a generic error message is returned to the client to prevent the application from crashing or exposing sensitive details.

## 6. Output Screens

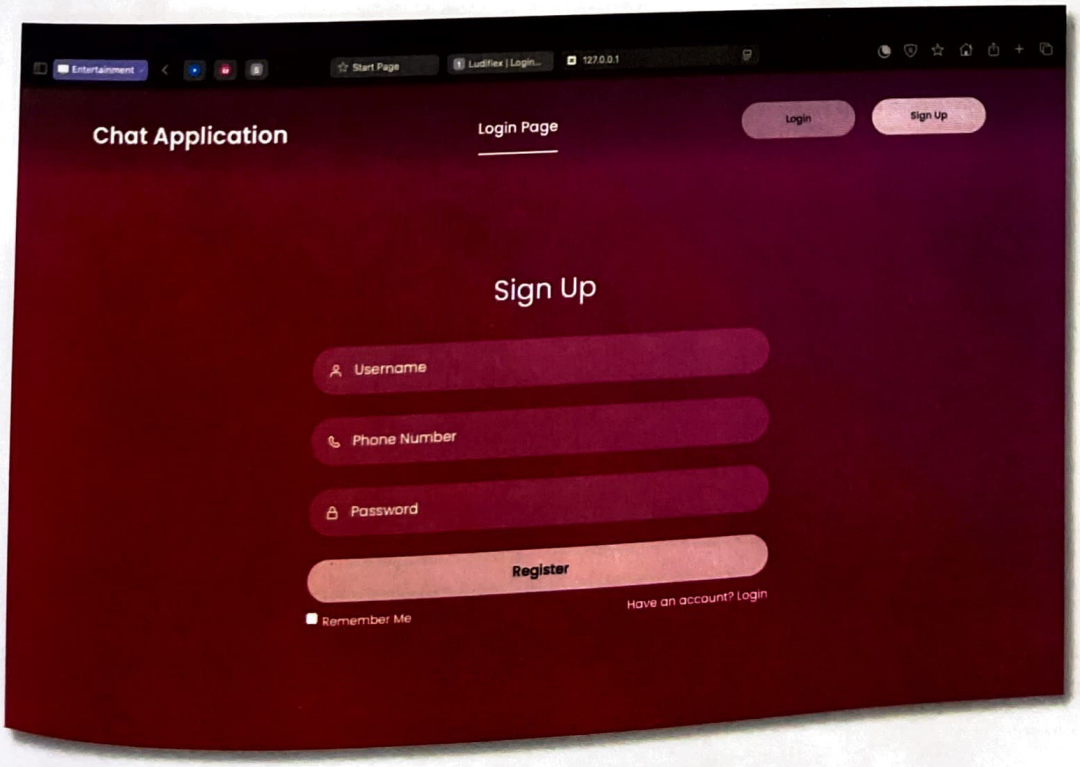
### 6.1. Login Page

The login page features fields for entering a username and password, along with buttons for logging in or signing up.

Login page:

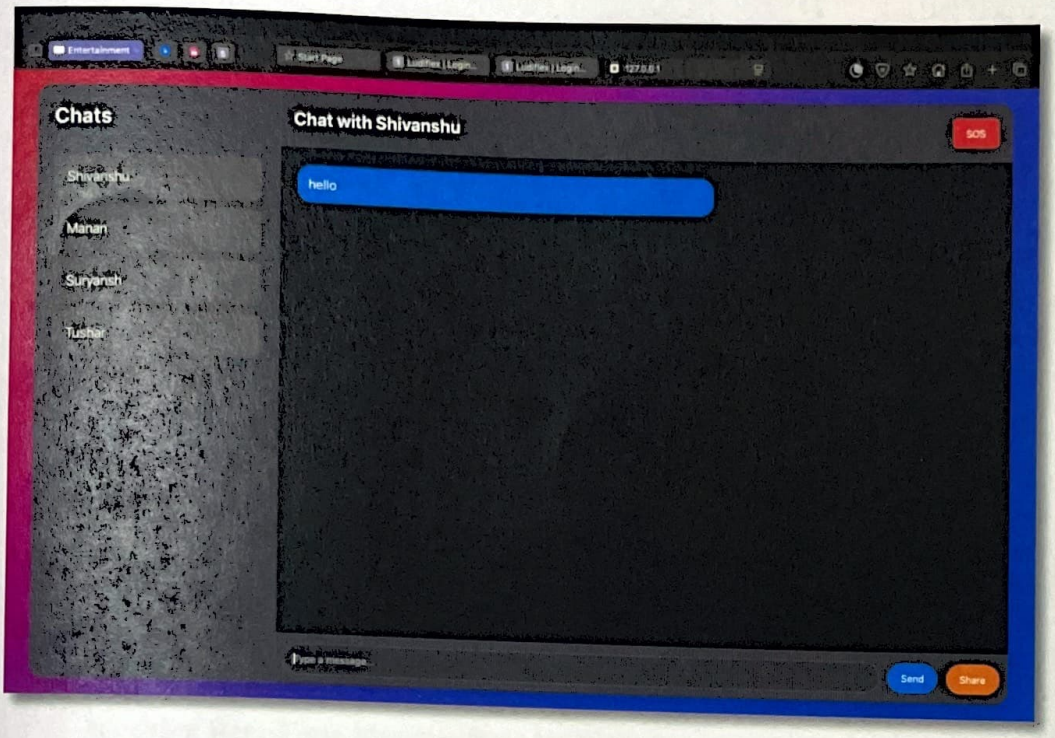


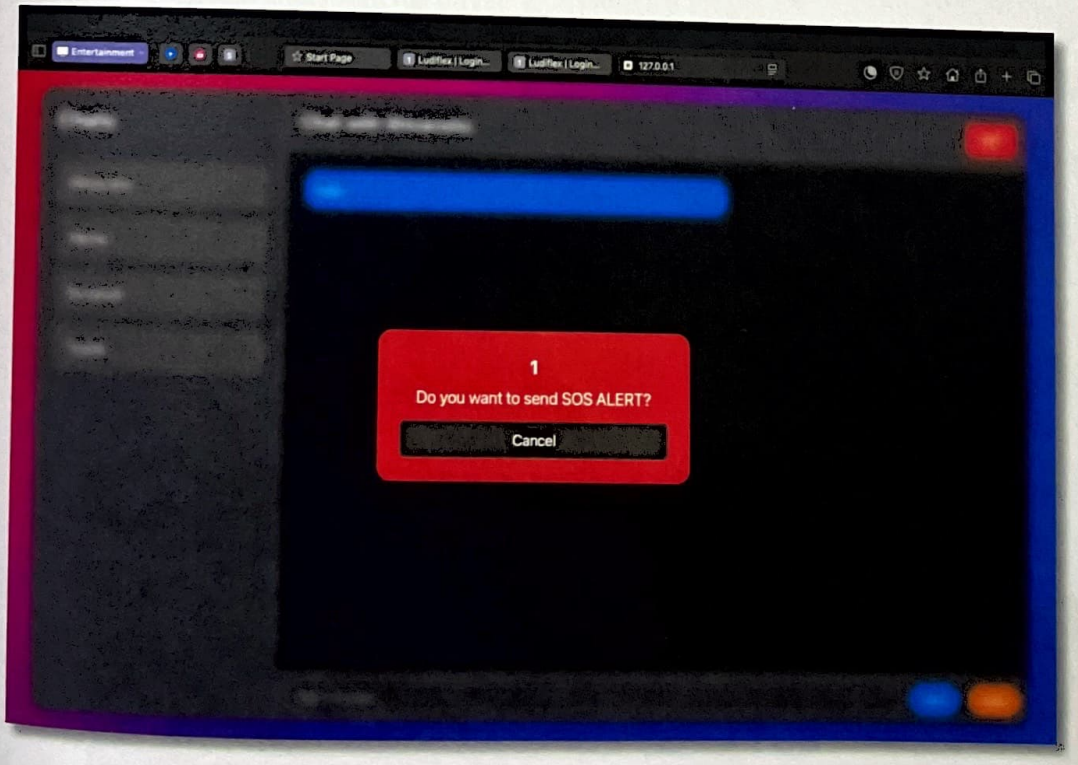

Signup page:



### 6.2. Chat Interface

The chat interface allows users to send and receive messages in real-time.



## 7. Testing and Debugging

Testing and debugging are crucial steps in the software development lifecycle to ensure that the application works as intended and is free of critical bugs or issues. In this section, we will discuss the test scenarios, including the validation of core functionalities, and present the results of these tests. The process ensures that the chat application is reliable, secure, and performs as expected.

### 7.1. Test Scenarios

Test scenarios are specific conditions or actions performed to validate the functionality and performance of different components of the chat application. The following key test scenarios were identified for this chat system:

- **Validating User Authentication (Login/Signup)**:
  One of the fundamental features of the chat application is the user authentication process, which includes both user sign-up and login. The test ensures that:
  - New users can successfully register with a valid username, password, and email.
  - Passwords are securely hashed and stored in the database.
  - Existing users can log in using their credentials and are authenticated correctly.
  - Edge cases such as empty fields, incorrect credentials, and already existing usernames are handled properly, displaying appropriate error messages to the user.
- These tests ensure the integrity of the authentication system, preventing unauthorized access and ensuring that only valid users can use the chat features.

- **Verifying Real-Time Message Broadcasting**:
  Real-time messaging is the core functionality of this chat application. The test for this scenario verifies that:
  - When one user sends a message, it is immediately broadcast to other connected users in the chat room (or private chat) without delays.
  - Messages are sent and received without any noticeable lag or errors.
  - All connected clients receive the message simultaneously, ensuring synchronization across all users.
  - Messages are correctly handled when there are multiple users connected, ensuring that the system can scale for multiple users at once.
- This test ensures that WebSocket communication is functioning correctly, and the real-time experience is consistent across users.

- **Testing MongoDB Data Storage and Retrieval**:
  MongoDB is responsible for storing user credentials and chat messages. This test scenario checks:
  - Whether new user data is correctly stored in the database after registration (with hashed passwords).
  - Whether messages are successfully stored in the database with the correct fields such as sender, receiver, content, and timestamp.
  - Whether the messages are correctly retrieved when a user logs in or requests chat history.
  - Data retrieval is tested for both individual users and group chats, ensuring that the application is able to fetch stored chat messages quickly and accurately.

- Ensuring that no data corruption or loss occurs during storage or retrieval, especially during high-volume usage.

- This test ensures that the database integration works seamlessly, and the application performs well in retrieving and storing data.

### 7.2. Test Results

After performing the above test scenarios, the results showed that the chat application is working as expected across all the key functionalities. Below is a breakdown of the test results:

- **User Authentication**:
  All tests related to user authentication passed successfully. The application correctly handles both registration and login, with secure password hashing in place. Invalid login attempts, such as incorrect passwords or missing fields, are properly handled by displaying appropriate error messages.

  - **Test Results**:
    - **Registration**: Passed (Valid username, email, and password successfully stored in MongoDB with hashing).
    - **Login**: Passed (Correct authentication based on stored hashed passwords).
    - **Invalid Login**: Passed (Incorrect credentials return the proper error message).

- **Real-Time Message Broadcasting**:
  The real-time message broadcasting functionality performed as expected. Messages sent by one user were instantly received by other connected users without noticeable delays. The system also handled multiple connected users, ensuring that no user experiences lag or missed messages.

  - **Test Results**:
    - **Single User Test**: Passed (Message sent by one user was received by the other).
    - **Multiple User Test**: Passed (Messages sent in group chat were received by all participants in real-time).
    - **Message Sync Test**: Passed (Messages synchronized between clients as expected, no missing or delayed messages).

- **MongoDB Data Storage and Retrieval**:
  MongoDB integration worked without any issues. User registration and chat messages were successfully stored and retrieved from the database. The system also handled fetching the complete chat history accurately, even with a large volume of messages.

  - **Test Results**:
    - **User Data Storage**: Passed (User data stored in MongoDB with hashed passwords).
    - **Message Storage**: Passed (All messages correctly stored in the database with necessary fields).
    - **Data Retrieval**: Passed (All messages are correctly fetched when a user logs in or requests chat history).

## 8. Limitations and Future Enhancements

While the current version of the chat application meets its primary goals of enabling real-time communication and user authentication, there are certain limitations in the existing system. Additionally, there are several areas where the system could be enhanced to provide a more robust and feature-rich experience for users and administrators. In this section, we will explore the current limitations of the system and propose potential future enhancements to address these gaps.

### 8.1. Current Limitations

The chat application, as it stands, performs well with essential features such as user authentication and real-time messaging. However, there are a few limitations that restrict its functionality:

- **Lack of Multimedia Message Support (Images/Videos):**
  Currently, the system only supports text-based messages. It does not allow users to send or receive multimedia messages, such as images or videos. As a result, users are limited to text-only interactions, which might not fully meet the communication needs of some users who prefer to share rich media (photos, videos, etc.). Multimedia support could add a significant improvement to the chat system, providing a richer user experience.
    - **Impact:**
      This limitation restricts the diversity of communication types within the chat application, especially for users who want to share visual content. This also reduces the overall engagement and functionality of the platform.
- **Basic Admin Functionalities:**
  The current version of the chat application offers very basic administrative controls, such as user registration and login. However, it lacks more advanced admin features such as user management (e.g., banning users, managing permissions), chat moderation (e.g., filtering inappropriate content), and analytics. Without these functionalities, administrators have limited control over the system's user interactions and data.
    - **Impact:**
      The lack of advanced admin features can hinder the management of a larger user base and create challenges in maintaining the security and integrity of the chat platform. Additionally, administrators may face difficulties in ensuring a positive user experience without robust moderation tools.

### 8.2. Planned Enhancements

To address the above limitations and improve the overall user experience, several future enhancements are planned for the chat application. These enhancements will provide users with more functionality and give administrators better control over the system. Some of the key planned enhancements are as follows:

- **Support for File Sharing (Images, Documents, etc.):**
  One of the most important planned enhancements is the integration of multimedia messaging. This will include the ability for users to send and receive images, videos, and documents. Users will be able to upload and download files within the chat interface, significantly improving the overall experience by allowing them to share various types of content.

- **Implementation Details**:
  This enhancement will require the integration of file upload functionality in the client-side interface (Java Swing) and a backend that handles file storage, such as saving images or videos in a cloud storage solution or the local server. The database schema will also need to be modified to support storing file metadata (e.g., file name, size, type) alongside chat messages.

- **Impact**:
  Supporting file sharing will make the chat application more versatile and engaging. Users will be able to have more dynamic conversations, and the platform will better support business or personal communications where file exchange is necessary.

- **Advanced Admin Functionalities**:
  The addition of more sophisticated admin features will help improve user management, platform security, and content moderation. The following admin functionalities are planned:

  - **User Management**: Admins will have the ability to manage user accounts, including banning or suspending problematic users, assigning roles (e.g., regular user, moderator, admin), and enabling or disabling certain features based on user roles.

  - **Chat Moderation**: Admins will be able to monitor chats, block inappropriate content, and filter messages based on predefined keywords or patterns. This feature is essential for maintaining a safe and positive environment for all users.

  - **Analytics and Reporting**: Admins will have access to analytics dashboards to monitor user activity, message volume, and other relevant metrics. This will allow admins to identify trends, manage server load, and ensure the platform is operating smoothly.

  - **Implementation Details**:
    To support these enhancements, a separate admin interface will be added to the front-end, and additional routes or endpoints will be implemented in the Node.js server to manage these features. The MongoDB database will also need to store user roles, bans, and chat content moderation rules.

  - **Impact**:
    With these enhancements, administrators will have greater control over the platform, ensuring better user experience, security, and smoother management of the chat system. Enhanced admin functionalities will also allow for more flexibility in customizing user roles and permissions.

## 9. Conclusion

The chat application developed as part of this project provides a comprehensive solution for real-time communication, user authentication, and secure data management. The application effectively implements several core features, including real-time messaging, user login, and message persistence, leveraging modern technologies like Node.js, MongoDB, and Java Swing. This section summarizes the achievements of the project, evaluates its success in meeting its objectives, and outlines the key takeaways.

### 9.1. Key Achievements

- **User Authentication**:
  One of the main objectives of the project was to implement a secure login system for users. This was successfully achieved by integrating user authentication, where users can sign up, log in, and securely store their credentials. Passwords are hashed using bcrypt before being stored in MongoDB, ensuring that sensitive information is protected.

- **Real-Time Communication**:
  The application supports real-time communication using WebSockets. Once a user logs in, they are able to send and receive messages instantly, without needing to refresh the page or make repeated requests to the server. This is accomplished through the use of Node.js and the `socket.io` library, which facilitates bidirectional communication between clients and the server.

- **Data Persistence and Storage**:
  MongoDB, a NoSQL database, is used to store user details and chat messages. MongoDB's flexibility and scalability make it a suitable choice for this project, especially as the system needs to handle increasing volumes of messages and user interactions. The messages are stored with relevant metadata, such as sender, receiver, and timestamp, ensuring that all communication is logged and can be retrieved when needed.

### 9.2. System Scalability and Efficiency

The application has been designed to be both scalable and efficient. Node.js's event-driven, non-blocking I/O model is well-suited for handling a large number of concurrent connections, making the system capable of supporting multiple users communicating in real-time without significant performance degradation. Additionally, MongoDB's scalability ensures that the application can handle the growing storage demands of chat messages and user data as the user base expands.

- **Scalability**:
  As the number of users increases, the application can scale horizontally by adding more server instances. MongoDB also supports horizontal scaling (sharding), allowing the database to grow efficiently by distributing data across multiple servers.

- **Efficiency**:
  The application has been optimized for performance, particularly in terms of real-time messaging. The use of WebSockets ensures that messages are delivered with minimal latency, providing users with a seamless experience.

### 9.3. Success in Meeting Project Objectives

The project successfully met its core objectives:

- **Real-Time Messaging**:
  The real-time messaging feature has been implemented using WebSockets, allowing for instant communication between users.

- **User Authentication**:
  The login and registration functionality, with secure password hashing, has been fully implemented, ensuring that user credentials are handled safely.

- **Secure Data Storage**:
  MongoDB was integrated into the application to handle the storage of user data and chat messages securely. This choice of database supports scalability and fast data retrieval.

### 9.4. Key Takeaways

This project highlights the importance of using modern technologies for building scalable and efficient web applications. By combining Node.js, MongoDB, and Java Swing, the chat application is able to provide a responsive and secure platform for users to communicate in real-time.

The use of WebSockets enables real-time communication without the need for frequent page refreshes, providing an experience similar to popular messaging apps. The MongoDB database ensures that data is securely stored and can be efficiently retrieved when needed, while bcrypt guarantees the protection of sensitive user information.

Although the project meets its basic requirements, it also presents several areas for future enhancement, such as supporting multimedia messages and providing more advanced admin features. These improvements will make the system more versatile and appealing to a wider user base.

## Appendices

The appendices contain additional technical details that provide insight into the system's design and troubleshooting process:

- **Appendix A: MongoDB Schema Design**
  This appendix will include the design of the MongoDB schema, showcasing how data is structured in the database. It will detail the collections used (e.g., Users and Messages), the fields within each collection, and any relationships between data entities. This section will provide a deeper understanding of how the database supports the core functionalities of the chat application.

- **Appendix B: Error Logs and Troubleshooting Tips**
  This appendix will include common error logs encountered during the development of the chat application, along with troubleshooting steps and solutions. It will cover issues such as connection problems, database errors, and WebSocket issues, helping developers to quickly resolve any problems that may arise during deployment or operation.

## 10. References

[1] Documentation: Official Java documentation on socket programming: https://docs.oracle.com/ javase/tutorial/networking/ sockets/.

[2] Java File I/O: Understanding file input/output in Java: https:// docs.oracle.com/javase/tutorial/essential/io/.

[3] Java Programming Language: Java SE 8 Platform documentation: https://docs.oracle.com/javase/8/docs/api/.

[4] Networking with Java: Java Networking and Sockets tutorial: https://www.geeksforgeeks.org/socket-programming-in-java/

[5]Zala, Nidhi, Jinan Fiaidhi, and Vinita Agrawal. "ChatterBox-A Real Time Chat Application." *Authorea Preprints* (2023).