

Chapter I: Introduction

1.1 Overview

This project presents a Real-Time One-to-One Chat Platform designed to enable direct, private, and secure communication between two users. The platform uses Java Spring Boot on the backend, React for the frontend, and WebSocket protocols (STOMP and SockJS) to facilitate bi-directional real-time messaging. The chat data is managed temporarily using MongoDB Atlas, with automatic data deletion once the session ends, ensuring complete user privacy.

This project is a modern web-based live chat platform that enables real-time, one-to-one communication between users through their browsers. It integrates Java Spring Boot as the backend framework, React for the frontend, and uses WebSocket protocols (STOMP and SockJS) for real-time data transmission. MongoDB Atlas is used to store chat data temporarily, maintaining user privacy by deleting all session data when both participants go offline.

In the era of digital transformation, real-time communication plays a vital role in everyday interaction, collaboration, and productivity. The demand for instant messaging platforms has grown exponentially with the increasing reliance on web and mobile applications. From casual conversations to professional communication, chat systems form a core component of user engagement.

Unlike traditional chat platforms that store data permanently or require complex authentication, our project focuses on simplicity, security, and privacy by design. The system can be accessed via any modern browser and provides a smooth, seamless user experience without compromising on speed or data protection.

One of the biggest motivations for this project is the rising concern over digital privacy. Users are becoming more aware of how their data is being stored, used, and even sold. Platforms like Facebook Messenger, WhatsApp, and Telegram retain messages for a long time or indefinitely, depending on settings and policies. Our system, on the other hand, introduces a refreshing alternative—ephemeral messaging without compromising on delivery speed or user experience.

The platform's design is also focused on being developer-friendly. It avoids third-party APIs or heavy backend services and instead relies on open-source, widely supported technologies. This not only keeps the implementation lightweight but also helps developers understand and contribute to the system more easily.

In summary, the Real-Time One-to-One Chat Platform is a lightweight, scalable, and privacy-respecting communication tool that can be used in various contexts including customer service, peer-to-peer collaboration, and more.

1.2 Problem Statement

Efficient, secure, privacy-respecting real-time chat solutions are scarce. Most available platforms either complicate simple one-to-one conversations, store user data permanently, or fail to provide true instant messaging. This project addresses those gaps by focusing on instant delivery, minimal data retention, and ease of use.

Despite the wide availability of chat applications today, most do not prioritize **data privacy** or **ephemeral communication**. Platforms such as WhatsApp and Messenger often retain data indefinitely or offer limited control to users over their data. Moreover, the backend architectures of these systems can be complex, resource-intensive, and not open to customization.

Another challenge is the **real-time aspect** of communication. While some systems rely on polling or long-polling methods, which introduce latency and inefficiency, others require extensive setup and third-party integrations. For developers and users who need a lightweight, focused chat system, existing options may be either too complicated or lack key features like privacy and instant responsiveness.

While real-time communication technologies have evolved significantly over the past decade, many existing chat platforms still suffer from a range of challenges—most notably concerning **data privacy**, **latency**, and **complexity** of deployment.

Most popular platforms (like WhatsApp, Messenger, or Telegram) retain chat histories permanently unless users manually delete them. This raises concerns for individuals who prioritize **privacy-first communication**, especially in sensitive use cases such as whistleblowing, temporary consultations, or private business deals. Even though some apps provide self-destructing message options, they are often hidden behind layers of settings or limited in flexibility.

Furthermore, traditional systems rely heavily on polling or long-polling techniques, which are resource-intensive and inefficient in delivering real-time experience. These methods periodically ask the server for updates, resulting in delays, unnecessary load, and poor user experience. In contrast, **WebSockets** allow for a persistent, bi-directional connection that is ideal for chat applications, but they are underutilized due to implementation complexity.

Another issue is with **permanent storage of user and message data**. While persistent storage can be useful, it is unnecessary—and even risky—in contexts where messages are meant to be temporary. Many users prefer not to have their conversations stored on company servers after their session ends.

Lastly, developers or educational institutions aiming to build or teach a chat system find existing platforms either **too complex**, closed-source, or bloated with features that distract from the core learning and messaging objective.

In summary, key problems identified:

Lack of temporary, privacy-respecting messaging systems

Complex architecture in existing platforms

Inefficiency in message delivery using traditional methods

Poor control over data storage and session cleanup

Need for a minimal, educational, and real-time chat solution

Hence, there is a strong need for a **lightweight, privacy-respecting, real-time one-to-one chat platform** that does not store data after a session ends, supports real-time messaging over WebSocket, and is easy to deploy and scale. Our project aims to fill this gap.

1.3 Objective of Project

- Deliver a simple, scalable chat platform supporting live, one-to-one private messaging
- Provide instant and reliable communication using modern web technologies
- Respect user privacy by deleting all chat/user data when both users disconnect
- Make the platform extensible for additional future features

The main objectives of this project are designed to address the core challenges identified in the problem statement and provide a practical, efficient, and privacy-respecting chat platform. The objectives reflect both technical requirements and user-centered features aimed at delivering a robust solution.

1.3.1 Develop a Simple and Scalable Chat Platform

The first objective is to create a platform that supports one-to-one chat communication with minimal setup and complexity. The system should be scalable to accommodate an increasing number of users without significant degradation in performance. Scalability ensures that the platform can be used not only for small peer-to-peer interactions but also in environments with many simultaneous users, such as customer support or educational platforms.

1.3.2 Enable Real-Time Communication Using Modern Web Technologies

Utilizing WebSocket technology, especially with STOMP and SockJS protocols, the project aims to establish a persistent, bi-directional communication channel between the client and server. This real-time communication reduces latency compared to traditional HTTP request-response cycles and polling techniques. The goal is to ensure that messages are delivered instantly, providing a seamless and interactive user experience.

1.3.3 Implement Ephemeral Messaging with Automatic Data Deletion

One of the unique objectives is to ensure user privacy by adopting ephemeral messaging. Messages and user session data will be stored temporarily only for the duration of the chat session. Once both users disconnect, all data related to the session will be automatically deleted from the database (MongoDB Atlas). This approach mitigates risks related to data breaches and unauthorized data retention.

1.3.4 Provide an Intuitive and Responsive User Interface

The frontend is developed using React, a popular JavaScript library known for its efficiency and component-based architecture. The interface aims to be clean, user-friendly, and responsive, allowing users to quickly send and receive messages, see real-time updates, and navigate the chat application without delays or confusion.

1.3.5 Design for Extensibility and Future Enhancements

The architecture is modular, which means future features such as group chats, file sharing, multimedia support, or mobile app integration can be added without overhauling the core system. Designing for extensibility ensures the platform can evolve with user needs and technological advances.

1.3.6 Ensure Cross-Platform Compatibility

The platform is designed to be accessed from any modern web browser, irrespective of the underlying operating system or device. This cross-platform compatibility ensures maximum accessibility and usability for diverse user groups.

1.3.7 Facilitate Easy Deployment and Maintenance

The choice of open-source, widely-supported technologies ensures that the platform can be deployed with ease on cloud or local servers. Clear documentation and modular code further facilitate maintenance and upgrades by developers.

Summary of Objectives

Objective Number	Description
1	Simple, scalable one-to-one chat system
2	Real-time messaging with WebSocket technologies

Objective Number	Description
3	Ephemeral messaging with automatic deletion
4	Intuitive, responsive frontend using React
5	Modular design for future enhancements

1.4 Applications or Scope

- Temporary peer-to-peer collaboration
- Customer support solutions requiring ephemeral chats
- Messaging platforms for users prioritizing privacy

The real-time one-to-one chat platform developed in this project has a wide range of applications across various domains. The platform's focus on privacy, simplicity, and real-time performance makes it suitable for several important use cases.

1.4.1 Temporary Peer-to-Peer Collaboration

In many professional and personal scenarios, users need to communicate and collaborate briefly without leaving a permanent record. This platform allows for instant messaging sessions that expire as soon as the users disconnect. For example, consultants and clients can exchange information quickly without worrying about data retention. Similarly, students working on group projects can discuss ideas temporarily without maintaining long chat histories.

1.4.2 Customer Support Solutions

Many businesses require live chat systems for customer support to provide immediate assistance and troubleshooting. This platform can be integrated into company websites as a live chat widget. The ephemeral nature of the chat ensures customer data is not stored longer than necessary, thereby aligning with data privacy regulations such as GDPR. This reduces liability for companies while providing efficient support.

1.4.3 Privacy-Centric Messaging Platforms

Certain user groups, including journalists, whistleblowers, and activists, prioritize privacy above all else. They require communication tools that do not retain sensitive information and prevent unauthorized access. Our platform's design supports ephemeral messaging, making it ideal for such privacy-conscious users.

1.4.4 Educational and Internal Communication

In academic settings, students and teachers often require real-time communication tools for clarifications, doubt solving, or brief discussions. This chat platform can serve as an internal communication medium that respects user privacy by not storing conversations indefinitely.

1.4.5 Developer and Open-Source Community Tools

Developers working on collaborative projects benefit from lightweight communication tools that are easy to integrate and modify. The open-source technology stack used in this project makes it accessible for developers to build upon or incorporate into larger systems.

1.4.6 Expansion to Other Communication Modes

Though the current scope focuses on text messaging, the platform can be expanded in future versions to support multimedia messaging such as images, videos, and voice notes. This increases the applicability of the platform across social and professional networks.

1.4.7 Enterprise Internal Communication

Enterprises increasingly seek secure communication channels for their employees. This platform can be deployed internally in corporate networks, allowing employees to communicate in real-time without fear of data leakage or permanent storage.

Broader Scope and Impact

The architecture and technology choices made in this project lay a solid foundation for future work in real-time communication. The modular design means that as new communication protocols or privacy requirements emerge, the system can adapt. It also opens doors to integration with mobile applications, voice assistants, and AI-powered chatbots.

Challenges and Opportunities

Implementing a system that deletes all data after session termination presents challenges such as ensuring data consistency and managing disconnections gracefully. However, it also offers the opportunity to set new standards in ephemeral communication technology.

Summary

1.5 Organization of Report

The report is structured into eight chapters, covering introduction, literature review, technical approach, implementation, results, a user manual, findings and future scope, and references.

The report is structured to provide a clear and systematic presentation of the entire project development process, from conceptualization through implementation to evaluation. The organization ensures that readers can easily understand the project's scope, technical background, development methodology, and outcomes.

Chapter I: Introduction

The first chapter introduces the project by explaining the motivation behind it, the challenges addressed, and the objectives set to achieve the desired outcomes. It provides the foundational understanding of why the Real-Time One-to-One Chat Platform is necessary in today's digital communication landscape. This chapter also outlines the potential applications and the overall structure of the report, preparing the reader for the detailed sections that follow.

Chapter II: Literature Survey

This chapter reviews existing work and technologies related to real-time chat applications. It discusses various communication protocols such as WebSocket, STOMP, and SockJS and analyzes their role in enabling efficient data transmission. It also examines backend frameworks like Java Spring Boot and frontend technologies like React, along with privacy practices and ephemeral messaging concepts. The literature survey provides a theoretical basis and technical insights that guided the project development.

Chapter III: Methodology

The methodology chapter details the approach taken to design and develop the chat platform. It covers the choice of technology stack, architectural decisions, and development frameworks. The chapter describes the flow of data, system modules, and communication between client and server using WebSocket protocols. Diagrams such as Entity-Relationship, Use Case, and Data Flow Diagrams are included to visually represent the system design and processes.

Chapter IV: Implementation

Chapter four focuses on the practical implementation of the project. It includes descriptions of the main functional components, code snippets, and explanations of critical algorithms and logic. This section demonstrates how the theoretical design was

transformed into a working application, detailing backend REST APIs, WebSocket configuration, and frontend React components.

Chapter V: Results

This chapter presents the outcomes of the project testing and evaluation. It discusses the performance metrics, real-time message delivery speeds, and privacy features verification. The results section also includes screenshots, use case demonstrations, and feedback collected during testing phases, confirming that the project objectives were met.

Chapter VI: User Manual

Chapter six provides step-by-step instructions on installing, configuring, and running the application. It lists the software and hardware requirements and explains how users can interact with the system. This section is intended for end-users and administrators, ensuring ease of deployment and use.

Chapter VII: Conclusion and Future Scope

The final chapter summarizes the achievements of the project and reflects on the challenges faced during development. It also discusses possible extensions and enhancements, such as group chats, media sharing, mobile apps, and notifications. The future scope highlights the project's potential growth and adaptation to evolving communication needs.

Chapter VIII: References

The last chapter lists all sources cited throughout the report, including research papers, books, articles, and web resources. This ensures academic integrity and provides readers with avenues for further study.

Chapter II: Literature Survey

2.1 Real-Time Communication Protocols

Real-time messaging requires efficient data transfer between client and server. Traditional HTTP-based methods like polling and long-polling are inefficient and slow for instant messaging. WebSocket, introduced in HTML5, enables persistent, full-duplex communication channels over a single TCP connection, drastically reducing latency and overhead. Protocols like STOMP (Simple Text Oriented Messaging Protocol) and SockJS provide standardized messaging formats and fallback mechanisms for browsers that do not fully support WebSocket.

Real-time communication protocols form the backbone of instant messaging applications. These protocols ensure data is transmitted promptly between clients and servers, maintaining low latency and high reliability. As modern applications demand instantaneous data exchange for smooth user experiences, understanding these protocols becomes critical for developers.

2.1.1 Traditional HTTP Communication Limitations

Historically, web applications used the HTTP protocol for client-server communication. HTTP is based on a request-response model where the client sends a request and waits for a response. While this model is sufficient for many applications, it is not ideal for real-time messaging.

One common technique to simulate real-time updates is **polling**, where the client periodically sends requests to the server asking for new data. Polling introduces unnecessary network traffic and latency because the server often responds with no new information. To mitigate this, **long polling** was introduced. In long polling, the server holds the client's request open until new data is available or a timeout occurs. While long polling reduces the number of requests, it still involves overhead and is less efficient compared to true real-time protocols.

2.1.2 WebSocket Protocol

The **WebSocket protocol**, standardized as RFC 6455 in 2011, revolutionized real-time communication on the web. Unlike HTTP, WebSocket establishes a **persistent, full-duplex connection** between the client and server over a single TCP socket. This connection allows data to flow freely in both directions at any time, enabling instantaneous message delivery.

WebSocket connections begin with an HTTP handshake, which then upgrades to the WebSocket protocol. This upgrade is seamless to users and applications, providing the

benefits of persistent connections without changing existing network infrastructure significantly.

2.1.3 Advantages of WebSocket

Low Latency: Because the connection remains open, there is no need to repeatedly establish connections, drastically reducing latency.

Bi-Directional Communication: Both client and server can send messages

independently, making it ideal for chat applications.

Reduced Overhead: Eliminates the HTTP headers overhead in each message, saving bandwidth.

Scalability: Supports thousands of simultaneous connections, suitable for large-scale applications.

2.1.4 STOMP Protocol

The **Simple Text Oriented Messaging Protocol (STOMP)** is a messaging protocol that provides an interoperable wire format for message brokers. STOMP acts as a simple and language-agnostic protocol layered on top of WebSocket to structure message frames for communication.

In real-time chat applications, STOMP helps standardize message formats and supports features like subscriptions, acknowledgments, and transactions. It simplifies integration by providing consistent semantics across diverse platforms and programming languages.

2.1.5 SockJS Library

While WebSocket is widely supported in modern browsers, some older browsers or restrictive network environments may not support it fully. **SockJS** is a JavaScript library that provides a WebSocket-like API but falls back to alternative communication methods such as AJAX long polling, iframe streaming, or JSONP polling when WebSocket is unavailable.

SockJS ensures broad compatibility and seamless user experience regardless of client environment. For developers, it simplifies client-side coding by abstracting the transport mechanism, allowing applications to use a single API while supporting multiple fallback protocols transparently.

2.1.6 Use Cases of These Protocols in Chat Applications

Modern chat systems extensively use WebSocket along with STOMP and SockJS to manage real-time messaging reliably:

Message Delivery: Ensures messages are delivered instantly with confirmation.

User Presence: Allows updating of user online/offline status dynamically.

Typing Indicators: Real-time notification when users are typing.

Multi-Device Sync: Keeps conversations synchronized across multiple devices.

Fallback Support: SockJS ensures older or restricted clients still get real-time capabilities.

2.1.7 Challenges in Implementation

Implementing these protocols requires managing connection states, handling disconnects gracefully, and ensuring message ordering and reliability. Developers must also consider network latency, security (encryption and authentication), and scalability when designing chat systems using these protocols.

2.2 Java Spring Boot for Backend Development

Java Spring Boot is a powerful framework for building scalable backend applications. It supports REST APIs, WebSocket integration, and security features essential for real-time systems. Many enterprises rely on Spring Boot due to its modular architecture, rapid development capabilities, and wide community support. It simplifies configuration and offers robust dependency injection, making it ideal for developing the backend of chat applications.

Introduction to Spring Boot

Spring Boot is a widely used framework that simplifies the process of building production-ready applications with the Java programming language. It is built on top of the Spring Framework and provides an opinionated approach to configuration, allowing developers to focus on business logic rather than boilerplate code.

Spring Boot's core philosophy is “**convention over configuration**”, which helps reduce the time required to set up a new project by providing default configurations for common application scenarios. This enables rapid development and deployment, making it a popular choice for backend services, including real-time chat applications.

Features Relevant to Real-Time Chat Applications

1. RESTful API Support

Spring Boot simplifies the creation of RESTful web services through annotations like `@RestController`, `@RequestMapping`, and others. These APIs serve as endpoints for client applications (e.g., React frontend) to communicate with the backend, handling requests such as user login, message retrieval, and session management.

2. WebSocket Integration

Real-time communication requires full-duplex communication channels between clients and servers. Spring Boot offers built-in support for WebSocket communication, enabling developers to build scalable, event-driven applications. It integrates seamlessly with messaging protocols such as STOMP, facilitating structured and standardized message handling.

3. Security Features

Spring Security, a sub-project of Spring, provides authentication, authorization, and protection against common vulnerabilities like Cross-Site Request Forgery (CSRF). Although this project emphasizes ephemeral data storage, secure communication and user session management are still critical to prevent unauthorized access and data leaks.

4. Dependency Injection and Modularity

Spring Boot utilizes dependency injection, which promotes loose coupling and easier testing. This modularity helps maintain clean codebases and allows components such as chat modules, session handlers, and storage managers to be developed and tested independently.

5. Configuration and Auto-Configuration

Spring Boot auto-configures components based on project dependencies, drastically simplifying application setup. For example, adding the `spring-boot-starter-websocket` dependency automatically configures WebSocket support, minimizing manual configuration effort.

Advantages of Using Spring Boot for Backend

Rapid Development: Quick project setup and reduced boilerplate code accelerate development cycles.

Robust Ecosystem: Access to a vast ecosystem of tools, libraries, and community support.

Scalability: Designed to handle high loads and many concurrent WebSocket connections efficiently.

Production Ready: Built-in metrics, health checks, and externalized configuration aid production deployment and monitoring.

Integration Friendly: Easily integrates with databases like MongoDB Atlas and frontend frameworks like React.

Challenges and Considerations

While Spring Boot provides extensive support, developers must manage certain challenges:

WebSocket Connection Management: Handling dropped connections, reconnections, and session expirations requires careful coding.

Message Serialization: Ensuring consistent message formats between Java backend and JavaScript frontend.

Scaling WebSocket Servers: Managing resources when scaling horizontally across multiple server instances.

Security Concerns: Properly securing WebSocket endpoints and managing authentication tokens or sessions.

Practical Use in the Project

In this project, Spring Boot serves as the backend that:

Manages user sessions and presence information.

Handles WebSocket connections using STOMP over WebSocket for structured messaging.

Interfaces with MongoDB Atlas to temporarily store chat messages and deletes them upon user disconnection.

Provides RESTful APIs for auxiliary functionalities like user authentication or fetching chat metadata.

Summary Table of Spring Boot Benefits

Feature	Benefit
REST API Support	Simplifies client-server communication
WebSocket Integration	Enables real-time bidirectional messaging
Security Features	Protects against unauthorized access and attacks
Dependency Injection	Enhances modularity and testing
Auto-Configuration	Reduces manual setup and boilerplate
Scalability	Supports many simultaneous users efficiently

2.3 React as Frontend Framework

Introduction to React

React is an open-source JavaScript library developed by Facebook for building user interfaces, particularly single-page applications (SPAs) that require dynamic and interactive components. It allows developers to create reusable UI components that efficiently update and render in response to data changes.

React's core strength lies in its **virtual DOM** (Document Object Model) mechanism, which minimizes expensive DOM manipulations by calculating the differences between the previous and current states and only updating the changed parts. This leads to improved performance and a smooth user experience, essential for real-time chat applications where UI needs to update instantly on receiving new messages or status changes.

Key Features Beneficial for Chat Applications

1. Component-Based Architecture

React encourages building UI as a tree of components, each managing its own state and props. This modular approach fits naturally with chat application structures, where separate components can represent chat windows, message lists, input fields, user presence indicators, and notifications. This separation simplifies development, debugging, and future enhancements.

2. State Management

React provides internal state management for components, enabling the UI to react to user inputs or external events. For complex state scenarios, libraries like Redux or Context API are often used to manage global application state, such as the current user's identity, active chat sessions, or message queues.

3. Real-Time UI Updates

React's rendering engine efficiently updates the UI in response to WebSocket events or API responses. When a new message arrives via the WebSocket connection, React updates the message list component without reloading the entire page, offering a seamless experience.

4. Rich Ecosystem and Tooling

The React ecosystem includes numerous third-party libraries and developer tools that enhance functionality:

React Router: For handling navigation between views.

Material-UI / Ant Design: UI component libraries for consistent styling.

Socket.IO Client / STOMP JS: Libraries for WebSocket integration.

Developer Tools: Browser extensions for inspecting React component hierarchies and states.

Why React is Suitable for This Project

Performance: Virtual DOM ensures smooth, lag-free UI updates, critical for chat apps.

Developer Productivity: Reusable components and rich tooling speed up development.

Community Support: Vast resources, tutorials, and libraries available.

Flexibility: Works well with backend frameworks like Spring Boot and supports WebSocket-based real-time communication.

Implementation Considerations

Implementing chat features with React involves managing:

WebSocket Connections: Establishing and maintaining socket connections for real-time data flow.

Event Handling: Processing incoming messages, typing indicators, presence updates.

UI State Synchronization: Keeping the frontend state consistent with backend events.

Error Handling: Managing connection drops and retries gracefully.

Challenges

State Complexity: As applications grow, managing global state and side effects requires careful architecture.

Performance Optimization: Preventing unnecessary re-renders is essential for smooth UX.

Cross-Browser Compatibility: Ensuring consistent behavior across different browsers and devices.

Summary Table: React Features vs Chat Requirements

React Feature	Benefit for Chat Application
Component-Based Design	Modular, maintainable UI elements
Virtual DOM	Efficient updates, improved performance
State Management	Dynamic, real-time UI reactions

React Feature	Benefit for Chat Application
Rich Ecosystem	Easy integration with WebSocket and UI libraries
Developer Tools	Simplifies debugging and development

React is a popular JavaScript library developed by Facebook for building dynamic user interfaces. It supports component-based architecture, virtual DOM rendering, and efficient state management. React's ability to update UI components seamlessly in response to data changes makes it ideal for chat applications requiring real-time updates.

Its ecosystem also supports tools like Redux for state management and libraries for WebSocket integration.

2.4 Privacy and Ephemeral Messaging

Introduction to Privacy Concerns in Digital Communication

With the exponential growth of digital communication platforms, privacy concerns have become paramount. Users increasingly worry about how their messages and data are stored, shared, and potentially exploited. High-profile data breaches and unauthorized surveillance have heightened awareness about digital privacy and led to a demand for secure and private messaging solutions.

The Concept of Ephemeral Messaging

Ephemeral messaging refers to communication where messages are automatically deleted after a certain period or once they have been read. This concept gained widespread popularity through platforms like Snapchat and Signal, which provide disappearing messages to protect user privacy and reduce data retention risks.

Ephemeral messaging addresses several privacy issues:

Data Minimization: Only essential data is stored temporarily, reducing the risk of unauthorized access.

User Control: Users have greater control over their communication history and can ensure messages are not permanently archived.

Reduced Liability: Service providers have less responsibility for long-term data storage, decreasing exposure to legal and compliance issues.

Implementing Ephemeral Messaging

To implement ephemeral messaging, systems must incorporate mechanisms to:

Track Message Lifetimes: Define how long messages should be retained.

Automatic Deletion: Remove messages and related metadata once they expire or sessions end.

Session Management: Manage user sessions to know when to trigger data deletion.

Encryption: Ensure messages are encrypted in transit and at rest to protect confidentiality.

Challenges in Ephemeral Messaging

Synchronization: Ensuring all devices delete messages simultaneously to prevent inconsistencies.

User Experience: Balancing privacy with usability; users expect some message history for context.

Data Recovery: Users cannot retrieve messages once deleted, which may be inconvenient.

Legal Compliance: Navigating regulations that may require data retention for audit or law enforcement purposes.

Privacy in the Context of This Project

The Real-Time One-to-One Chat Platform focuses on **ephemeral messaging by design**. It does not permanently store user messages; instead, it keeps chat data only temporarily on MongoDB Atlas and deletes everything as soon as both users disconnect.

This approach assures:

No Long-Term Storage: User conversations are never stored beyond the session.

Enhanced Security: Reduced attack surface as data is not persistently available.

User Trust: Users can communicate without fear of their chats being stored or monitored indefinitely.

Comparative Overview of Ephemeral Messaging Platforms

Snapchat: Popularized disappearing messages primarily for multimedia content with a default expiration time.

Signal: Focuses on privacy with end-to-end encryption and disappearing messages configurable by users.

Telegram: Offers self-destructing messages in “secret chats” with customizable timers.

WhatsApp: Allows users to enable disappearing messages per chat, deleting messages after 7 days by default.

Technological Solutions Supporting Privacy

End-to-End Encryption (E2EE): Ensures only communicating users can read messages, even service providers cannot access them.

Secure Protocols: Use of TLS and other transport security to protect messages in transit.

Ephemeral Storage: Databases or storage layers designed to handle transient data with automatic cleanup.

Summary Table: Privacy Features in Ephemeral Messaging

Feature	Description
Automatic Message Deletion	Removes messages after read or timeout
Data Minimization	Stores minimal data only as necessary
End-to-End Encryption	Protects messages from unauthorized access
User-Controlled Privacy	Allows users to set message lifetime and visibility

Feature	Description
Compliance Considerations	Balances privacy with legal data retention needs

Privacy concerns in digital communication have led to the rise of ephemeral messaging platforms like Snapchat and Signal. These platforms ensure messages are deleted after being read or after a certain time. Ephemeral messaging reduces the risk of data leaks and unauthorized access. Implementing ephemeral data storage requires careful design of data lifecycle, auto-deletion mechanisms, and session management.

2.5 NoSQL Databases and MongoDB Atlas

Introduction to NoSQL Databases

NoSQL databases have gained significant popularity as alternatives to traditional relational databases (RDBMS) because of their flexibility, scalability, and performance. Unlike relational databases which use fixed schemas and tables, NoSQL databases store data in flexible formats such as documents, key-value pairs, graphs, or wide-columns. This flexibility makes NoSQL ideal for handling unstructured or semi-structured data, which is common in modern web applications.

Types of NoSQL Databases

Document Stores: Store data as JSON-like documents. MongoDB is a leading example.

Key-Value Stores: Store data as key-value pairs, e.g., Redis,

Column-Family Stores: Data stored in column families, like Apache Cassandra.

Graph Databases: Focused on relationships between data points, e.g., Neo4j.

Why NoSQL for Real-Time Chat Applications?

Real-time chat systems generate diverse and rapidly changing data such as user sessions, messages, presence information, and metadata. NoSQL databases like MongoDB are suitable because:

Schema Flexibility: Messages can vary in structure (text, emoji, media) without requiring rigid schemas.

Horizontal Scalability: Easily scale out by adding nodes to handle increased loads.

High Throughput: Optimized for write-heavy operations typical in chat messaging.

Rapid Development: Allows faster iterations without schema migrations.

Introduction to MongoDB Atlas

MongoDB Atlas is a fully managed cloud database service provided by MongoDB Inc. It offers:

Cloud Deployment: Runs on popular cloud providers (AWS, Azure, Google Cloud).

Automated Backups and Monitoring: Ensures data safety and system health.

Scalability and Availability: Supports auto-scaling and multi-region deployment for low latency.

Security: Provides encryption at rest and in transit, role-based access control, and network isolation.

Serverless Options: Flexible pricing and provisioning models.

MongoDB Atlas Features Relevant to Chat Applications

Document-Based Storage: Stores messages as JSON documents, fitting the flexible data nature of chats.

TTL (Time To Live) Indexes: Allows automatic expiration of documents, perfect for ephemeral messaging to delete old messages without manual intervention

Real-Time Triggers: Can trigger serverless functions on database changes to implement features like notifications or analytics.

Global Clusters: Helps reduce latency by storing data close to users in different geographic locations.

Integration with Spring Boot

MongoDB Atlas integrates smoothly with Spring Boot using the Spring Data MongoDB module. This integration allows:

Simplified data access using repositories and templates.

Automatic mapping of Java objects (POJOs) to MongoDB documents.

Support for reactive programming models to handle non-blocking data operations, enhancing real-time performance.

Benefits and Challenges

Benefits	Challenges
Schema flexibility for evolving data	Managing eventual consistency
High scalability and availability	Complex queries can be inefficient
Supports ephemeral storage via TTL	Requires careful index design
Easy cloud deployment and management	Data modeling requires new approaches

Summary

The use of MongoDB Atlas as a NoSQL cloud database offers a powerful and flexible backend solution for real-time chat applications, especially those requiring ephemeral data storage and global accessibility. It aligns well with the project goals of scalability, privacy, and efficient data management.

MongoDB Atlas is a cloud-hosted NoSQL database that offers flexible document-based storage. It supports horizontal scaling, high availability, and real-time data access. For chat applications, MongoDB's schema-less design allows quick adaptation to changing message formats and ephemeral data storage. Cloud deployment reduces the overhead of server management and provides secure, encrypted storage.

2.6 Integration Challenges of WebSocket with Java and React

Introduction

Integrating WebSocket communication between a Java backend (using Spring Boot) and a React frontend can be challenging due to differences in protocols, data formats, and

connection management. While WebSocket provides a standardized communication channel, practical implementation requires careful handling to ensure seamless real-time messaging.

2.6.1 Protocol Mismatches and Message Formatting

WebSocket itself is a low-level protocol and does not define message structure or semantics. Therefore, additional protocols like STOMP are used to standardize message frames. Ensuring that both the Java backend and React frontend adhere to the same protocol version and message format is crucial.

Differences in serialization formats (JSON, XML, or custom formats) may cause incompatibilities. The backend must serialize messages correctly, and the frontend must parse them reliably.

2.6.2 Connection Stability and Lifecycle Management

Maintaining persistent WebSocket connections is difficult due to network fluctuations, browser limitations, and server timeouts. Both frontend and backend must implement robust reconnection logic to handle unexpected disconnects without losing message continuity.

Managing the connection lifecycle involves:

- Opening connections
- Detecting and handling disconnections
- Reestablishing connections with minimal delay
- Synchronizing message state upon reconnection

2.6.3 Subscription and Message Routing

In protocols like STOMP, clients subscribe to specific topics or queues. The backend must correctly route messages to subscribed clients. Misconfiguration can lead to messages not being delivered or being delivered to unintended recipients.

Ensuring accurate subscription management, including subscribing, unsubscribing, and handling subscription errors, is vital for proper message flow.

2.6.4 Security Concerns

WebSocket connections need to be secured using TLS (wss://) to prevent eavesdropping and man-in-the-middle attacks. Authentication and authorization mechanisms must be enforced to restrict access to authorized users only.

Handling authentication tokens (e.g., JWT) in WebSocket headers or connection parameters is complex since WebSocket does not natively support headers after the initial handshake.

2.6.5 Handling Concurrency and Scalability

As user count grows, managing thousands of simultaneous WebSocket connections is resource-intensive. The backend architecture must efficiently distribute load, possibly using clustering or message brokers like RabbitMQ or Kafka to ensure scalability.

Concurrency issues such as race conditions or message ordering must be addressed to maintain data integrity.

2.6.6 Cross-Origin Resource Sharing (CORS) Issues

Browsers enforce strict CORS policies that can block WebSocket connections if the backend server does not properly allow requests from the frontend domain. Proper CORS configuration on the backend is necessary to enable smooth communication.

2.6.7 Debugging and Monitoring

Debugging WebSocket connections is more complex than traditional HTTP due to persistent connections and asynchronous message flows. Tools like browser developer consoles, Wireshark, or custom logging must be used effectively.

Monitoring connection health, message throughput, and error rates helps in maintaining system reliability.

Summary Table of Key Integration Challenges

Challenge	Description
Protocol and Message Format	Ensuring both ends use compatible protocols and serialization formats
Connection Management	Handling connection drops, reconnections, and lifecycle events
Subscription Handling	Correctly routing messages to subscribed clients
Security	Securing WebSocket connections and managing authentication

Challenge	Description
Scalability and Concurrency	Managing large numbers of connections and message ordering
CORS Configuration	Ensuring browser policies allow cross-origin WebSocket connections
Debugging and Monitoring	Tools and techniques for troubleshooting and performance tracking

Integrating WebSocket between Java backend (Spring Boot) and React frontend presents several challenges such as protocol mismatches, message serialization, and connection stability. Libraries like STOMP and SockJS help by providing consistent protocols and fallback options. Handling subscriptions, error events, and reconnections are critical for a stable chat experience.

2.7 Comparative Analysis of Existing Chat Platforms

Introduction

In the rapidly evolving field of digital communication, numerous chat platforms have emerged, each offering unique features and technical architectures. Analyzing popular platforms such as Slack, WhatsApp Web, and Microsoft Teams provides valuable insights into effective design choices, scalability strategies, and user experience enhancements.

Slack

Slack is a widely-used team collaboration tool known for its channels, integrations, and real-time messaging capabilities. Its architecture combines WebSocket for live updates and REST APIs for traditional requests. Slack uses message queuing and persistent storage to ensure message delivery and history.

Weaknesses:

- Heavier resource consumption

- Complexity in managing workspaces and permissions

WhatsApp Web

WhatsApp Web offers a browser-based interface synchronized with the mobile app. It uses end-to-end encryption, WebSocket for real-time updates, and message queues to handle offline scenarios.

Strengths:

- Strong privacy with end-to-end encryption

- Seamless synchronization across devices

- Efficient real-time message delivery

- Simple user interface

Weaknesses:

- Dependent on mobile app being connected

- Limited support for multi-device independent use

- Less customization and extensibility

Microsoft Teams

Microsoft Teams is an enterprise communication platform integrating chat, video conferencing, and collaboration tools. It leverages WebSocket for messaging and integrates with Microsoft 365 services.

Strengths:

- Deep integration with Microsoft Office suite

- Supports threaded conversations and rich media

- Advanced security and compliance features

Weaknesses:

- Complex interface with steep learning curve

- Higher system resource usage

- Requires Microsoft 365 subscription

Comparative Insights

Platform	Messaging Protocols	Privacy & Security	Scalability	Features	User Experience
Slack	WebSocket + REST APIs	Data encrypted in transit	High	Integrations, persistent chat	Feature-rich, moderate complexity
WhatsApp Web	WebSocket + Mobile Sync	End-to-end encryption	Moderate	Simple chat, voice/video calls	Intuitive, mobile-dependent
Microsoft Teams	WebSocket + MS 365 APIs	Enterprise-grade security	Very High	Collaboration, meetings	Powerful but complex

Lessons for Our Project

Privacy by Default: Inspired by WhatsApp's encryption and ephemeral messaging, ensuring user data is protected.

Scalability Considerations: Learning from Microsoft Teams' handling of large organizations for future expansions

Simplicity and Extensibility: Following Slack's modular approach to allow adding features progressively.

Real-Time Communication Backbone: Leveraging WebSocket combined with messaging protocols to ensure low latency

Popular platforms like Slack, WhatsApp Web, and Microsoft Teams employ various architectures combining WebSocket, REST APIs, and persistent storage. WhatsApp uses end-to-end encryption and message queueing for offline delivery. Slack integrates bots

2.8 Message Handling and Event-Driven

Architecture

Introduction

Modern chat applications rely heavily on efficient message handling and event-driven architecture to deliver real-time, reliable, and scalable communication. Event-driven design enables systems to respond asynchronously to changes or actions, such as message sending, receiving, user presence updates, and system notifications.

Message Handling in Real-Time Chat

Message handling involves the processes of creating, transmitting, receiving, and displaying messages between users with minimal delay and loss.

Key aspects include:

Message Serialization: Messages are converted to formats like JSON or XML for transmission over networks. Both sender and receiver must agree on the format.

Message Queuing: Queues ensure messages are delivered in order, even if network interruptions occur. Queues temporarily hold messages until they can be delivered.

Acknowledgements: Confirmation messages help guarantee delivery, prompting retransmission if necessary.

Message Persistence: For ephemeral systems, messages may only persist during active sessions, with automatic deletion after. Others may store history permanently.

Ordering and Consistency: Messages must appear in correct order to users; techniques like timestamps and sequence numbers help achieve this.

Event-Driven Architecture (EDA)

EDA is a software architecture paradigm where components communicate by producing and consuming events asynchronously.

Producers: Components that generate events (e.g., user sending a message).

Consumers: Components that act upon events (e.g., updating UI, storing message).

Advantages of EDA in Chat Systems

Scalability: Components can be scaled independently, improving system responsiveness under load.

Decoupling: Producers and consumers are loosely coupled, facilitating modular development and easier maintenance.

Asynchronous Processing: Systems can handle high volumes without blocking processes, enhancing performance.

Resilience: Event queues enable buffering during outages, ensuring message delivery once the system recovers.

Practical Implementation in Project

In our project

The React frontend acts as an event producer (sending messages) and consumer (receiving updates).

The Spring Boot backend manages WebSocket connections, routing events between clients.

MongoDB Atlas temporarily stores messages during sessions, acting as an event store.

Events like “user joined,” “message sent,” and “user disconnected” trigger state changes and UI updates.

Challenges

Ensuring **exactly-once** message delivery semantics to avoid duplicates or loss.

Handling out-of-order messages in unreliable networks.

Managing event replay or recovery after disconnections.

Monitoring event flow and troubleshooting delays or failures.

Summary Table: Message Handling Techniques

2.9 Asynchronous Communication and Reactive Programming

Introduction

Modern real-time applications require efficient handling of multiple concurrent events and data streams without blocking the main execution thread. **Asynchronous communication** allows systems to perform tasks concurrently, improving responsiveness and scalability. **Reactive programming** is a programming paradigm designed to work with asynchronous data streams, enabling developers to build event-driven, non-blocking applications.

Asynchronous Communication

Asynchronous communication decouples the sender and receiver, allowing them to operate independently without waiting for responses. In the context of chat applications, this enables users to send and receive messages instantly without freezing the user interface.

Key features include:

Non-blocking I/O: Input/output operations do not block the application thread, allowing other processes to continue.

Event Loops: Manage asynchronous events by continuously checking for new tasks or messages

Callbacks and promises: Mechanisms to handle operations that complete in the future.

Reactive Programming Paradigm

Reactive programming focuses on propagating changes through data streams, reacting to new data or events as they occur.

Core concepts:

Observables: Streams of data or events that can be observed.

Observers: Components that subscribe to observables to receive updates.

Operators: Functions that transform, filter, or combine data streams.

Benefits in Chat Applications

Improved Performance: Non-blocking operations increase throughput and reduce latency.

Scalability: Supports thousands of concurrent connections efficiently.

Responsive UI: Enables instant updates without UI freezing or delays.

Error Handling: Propagates errors through streams for centralized handling.

Spring WebFlux and Reactive Spring

Spring WebFlux is a reactive web framework built on Project Reactor. It supports reactive streams and non-blocking web servers such as Netty.

In the chat platform:

Spring WebFlux manages WebSocket connections reactively.

Reactive MongoDB drivers provide non-blocking database access.

Challenges

Steep learning curve compared to traditional synchronous programming.

Debugging asynchronous flows can be complex.

Requires careful design to avoid memory leaks and backpressure issues.

Summary Table: Asynchronous and Reactive Features

Feature	Description
Non-blocking I/O	Allows concurrent operations without blocking
Observables/Streams	Represent continuous data/event flows
Backpressure Handling	Controls data flow to prevent overload

Scalability in Chat Applications

applications, scalability is critical because the number of users and message volume can grow rapidly, especially during peak times.

There are two primary types of scalability:

Vertical Scaling: Adding more resources (CPU, RAM) to a single server. This is limited by hardware capacity.

Horizontal Scaling: Adding more servers to distribute the load. This approach provides better fault tolerance and can handle massive user bases.

Techniques to Achieve Scalability

1. Load Balancing

Load balancers distribute incoming network traffic across multiple servers, preventing any single server from becoming a bottleneck. Popular load balancers include NGINX, HAProxy, and cloud-based services like AWS ELB.

2. Stateless Backend Services

Designing backend services to be stateless allows easy horizontal scaling. Session information and state can be stored in distributed caches or databases.

3. Microservices Architecture

Breaking the application into smaller, independent services allows each to scale according to demand. For example, chat messaging, user authentication, and presence tracking can be separate services.

4. Database Scalability

Using NoSQL databases like MongoDB Atlas allows horizontal scaling through sharding. Proper indexing and query optimization also improve performance.

Load Testing

Load testing simulates high user traffic to evaluate how the system performs under stress. It helps identify bottlenecks, capacity limits, and stability issues.

Common load testing tools:

Apache JMeter: Open-source tool for performance testing.

Locust: Python-based load testing tool.

Gatling: High-performance load testing framework.

Metrics to Monitor

Response Time: Time taken to process a request.

Throughput: Number of requests handled per second.

Error Rate: Percentage of failed requests.

Resource Utilization: CPU, memory, network usage.

Results Interpretation

Analyzing test results helps optimize the system by:

Scaling up or out to meet demand.

Identifying slow database queries.

Improving code efficiency.

Scalability in Our Project

Our Real-Time One-to-One Chat Platform uses:

-

Spring Boot backend capable of handling multiple WebSocket connections.

MongoDB Atlas with horizontal scaling support.

Modular architecture enabling microservices in future versions.

Summary Table: Scalability and Load Testing Techniques

Technique	Purpose
Load Balancing	Distributes traffic to avoid server overload
Stateless Services	Simplifies horizontal scaling

Technique	Purpose
Microservices	Allows independent scaling of components
NoSQL Scaling	Enables database sharding and replication
Load Testing Tools	Simulate user load and evaluate performance
Performance Metrics	Monitor system health and capacity

Scalability is crucial for chat platforms expected to support many simultaneous users. Techniques like load balancing, horizontal scaling, and microservices architecture are used. Performance benchmarks measure message latency, throughput, and resource usage under stress.

2.11 Frontend UI Components for Chat

Introduction

User Interface (UI) plays a vital role in the success of any chat application. A well-designed frontend not only enhances usability but also improves user engagement and satisfaction. Chat apps require dynamic and responsive components that update in real time to reflect ongoing conversations and user activities.

Key UI Components in Chat Applications

1. Chat Window

The main area where users read and send messages. It dynamically updates with new messages, user statuses, and typing indicators. The chat window supports scrolling, message grouping, and visual differentiation between sent and received messages.

2. Message Input Box

A text input field allowing users to compose and send messages. It often includes features such as:

- Auto-resizing to fit multiple lines

- Emoji picker integration

File upload buttons

Send button with keyboard shortcut support (e.g., Enter key)

3. Message List

Displays the chronological list of messages with metadata like timestamps, sender names, and read receipts. Messages are often grouped by sender and time to improve readability.

4. Typing Indicator

Shows real-time feedback when other users are typing, enhancing interactivity and reducing uncertainty about message delivery delays.

5. User Presence Indicator

Displays the online/offline status of users, often with colored dots or icons, to provide context on message availability.

6. Notifications

Real-time alerts for new messages, mentions, or important events. Notifications can be visual (pop-ups), audio, or badge counts.

7. Emojis and Reactions

Support for emojis and message reactions allows users to express emotions and responses succinctly.

Libraries and Tools

Popular frontend libraries and tools facilitate building these components:

Material-UI / Ant Design: Pre-built UI components for consistent styling.

React Emoji Picker: Easy integration of emoji selection.

React Virtualized: Efficient rendering of long message lists.

Socket.IO Client / STOMP JS: Manage WebSocket communication for live updates.

Chat apps must be accessible on various devices and screen sizes. Responsive design ensures usability on desktops, tablets, and smartphones. Accessibility features such as keyboard navigation, screen reader support, and high-contrast themes improve inclusivity.

Challenges in UI Development

- Performance:** Handling rapid message updates without UI lag.
- State Management:** Synchronizing UI state with backend events.
- Error Handling:** Graceful recovery from failed message sends or connection drops.
- Internationalization:** Supporting multiple languages and text directions.

Summary Table: Key UI Components and Functions

Component	Purpose
Chat Window	Displays conversation dynamically
Message Input Box	Enables message composition
Message List	Shows ordered messages with metadata
Typing Indicator	Shows when others are typing
User Presence	Displays online/offline status
Notifications	Alerts for new messages or events
Emojis & Reactions	Allows expression of emotions

UI components like typing indicators, emoji pickers, read receipts, and message status improve user experience. Open-source libraries and frameworks help implement these features efficiently.

2.12 Future Trends in Chat Applications

Introduction

The landscape of chat applications is continuously evolving, driven by advancements in technology and changing user expectations. As real-time communication becomes an

2.12.1 Integration of Artificial Intelligence (AI)

AI technologies such as natural language processing (NLP), sentiment analysis, and chatbots are increasingly integrated into chat platforms. AI can assist users by:

- Automating routine responses through chatbots
- Understanding user intent to provide relevant information
- Moderating content to detect spam or abuse
- Analyzing sentiment to improve communication tone

These capabilities enhance user engagement and reduce the workload on human operators.

2.12.2 Voice and Video Integration

While text remains the primary mode of communication, voice and video messaging are becoming more prevalent. Future chat applications are expected to support seamless switching between text, voice, and video within conversations, providing richer interaction modes.

2.12.3 Enhanced Privacy and Security

With growing privacy concerns, chat apps will likely implement stronger encryption, decentralized data storage, and user-controlled privacy settings. Technologies like blockchain could be used to verify message integrity and ownership without centralized servers.

2.12.4 Augmented Reality (AR) and Virtual Reality (VR)

AR and VR technologies promise immersive communication experiences. Future chat applications may allow users to interact in virtual spaces, using avatars and 3D environments, enhancing remote collaboration and social interaction.

2.12.5 Cross-Platform and Device Synchronization

Users increasingly expect chat continuity across devices — from smartphones and tablets to desktops and wearables. Future platforms will offer seamless synchronization and cloud-based message storage with ephemeral options for privacy.

2.12.6 Automation and Workflow Integration

Chapter III: Methodology

3.1 Background / Overview of Methodology

In the development of the Real-Time One-to-One Chat Platform, a well-defined methodology is essential to ensure that the project meets its goals of privacy, scalability, and real-time communication. The methodology revolves around leveraging modern web technologies, protocols, and database solutions to create a responsive and secure messaging system.

The methodology is based on a **client-server architecture**, where the frontend client (built with React) communicates with the backend server (developed using Java Spring Boot) over WebSocket connections. The communication employs protocols like STOMP (Simple Text Oriented Messaging Protocol) layered over WebSocket, which standardizes the messaging format and supports subscription-based message delivery. To ensure compatibility with browsers that do not fully support WebSocket, the SockJS library is used as a fallback mechanism, enabling alternative transport methods such as AJAX long polling or streaming.

The **ephemeral nature** of the chat data is a central consideration in the methodology. Unlike traditional chat applications that store message history indefinitely, this project stores messages temporarily in a NoSQL database (MongoDB Atlas). The messages and associated user session data are automatically deleted once both users in a chat session disconnect. This approach safeguards user privacy and reduces data retention risks.

The development follows an **incremental and modular approach**, dividing the system into smaller, manageable components or modules. Each module focuses on a specific functionality—such as user session management, real-time messaging, ephemeral storage, frontend updates, and error handling—allowing parallel development and easier maintenance.

Testing is integrated throughout the development process to verify that each module performs as expected individually (unit testing) and in coordination with others (integration testing). Special attention is given to real-time communication aspects, ensuring low latency, message ordering, and robustness in connection handling.

Deployment considerations include using cloud-based platforms for hosting both backend and frontend, leveraging MongoDB Atlas's cloud capabilities for database management. The project is designed to be extensible, allowing future enhancements such as group chats, media sharing, and mobile app support without major restructuring.

In summary, the methodology emphasizes the following key principles:

Use of **established web protocols and libraries** to ensure compatibility and efficiency.

Focus on **privacy and ephemeral data handling** to protect user information

Modular system design for **scalability and maintainability**.

Incremental development and thorough testing to ensure **reliability and performance**.

The methodology section explains the approach and techniques employed to develop the Real-Time One-to-One Chat Platform. The project is based on combining proven technologies that ensure scalability, privacy, and performance.

Our approach uses a client-server architecture where the frontend (React) interacts with the backend (Java Spring Boot) over WebSocket connections using STOMP and SockJS protocols. The ephemeral data storage is managed through MongoDB Atlas, where messages are stored temporarily and deleted once both users disconnect.

This methodology ensures real-time bi-directional communication, data privacy, and modular design, facilitating future enhancements.

3.2 Project Platforms Used in Project

The development of the Real-Time One-to-One Chat Platform leverages a set of modern and widely adopted platforms and technologies that together provide a robust foundation for building scalable, real-time, and secure applications. Below is a detailed overview of the key platforms used in the project:

Backend Platform: Java Spring Boot

Java Spring Boot is the core backend framework used to develop RESTful APIs and WebSocket endpoints. Its advantages include:

Rapid Development: Offers auto-configuration and starter dependencies, reducing boilerplate and speeding up setup.

WebSocket Support: Native support for WebSocket with STOMP simplifies real-time messaging implementation.

Security: Integration with Spring Security helps manage authentication and authorization.

Frontend Platform: React

React is a popular JavaScript library used for building user interfaces with component-based architecture. Features include:

Declarative UI: React enables building interactive UIs by efficiently updating and rendering components based on state changes.

Component Reusability: Encourages modular development through reusable UI components, improving maintainability.

Virtual DOM: Enhances performance by minimizing direct DOM manipulation.

Ecosystem: Access to tools like React Router for navigation and Redux for state management.

Real-Time Communication: WebSocket, STOMP, SockJS

WebSocket: Provides full-duplex communication channels over a single TCP connection, enabling real-time data exchange.

STOMP Protocol: A simple messaging protocol layered over WebSocket to structure messages and manage subscriptions.

SockJS: A JavaScript library that offers fallback options like AJAX polling if WebSocket is unavailable, ensuring broad browser compatibility.

Database Platform: MongoDB Atlas

MongoDB Atlas is a cloud-hosted NoSQL database service used for storing chat data temporarily. Its features include:

Document-Oriented Storage: Stores messages as JSON-like documents, offering schema flexibility.

TTL Indexes: Automatically deletes documents after a specified time, supporting ephemeral messaging.

Scalability and Availability: Supports automatic scaling and multi-region deployments.

Security: Includes encryption at rest and in transit, role-based access control, and network isolation.

Development Tools

Maven: Build automation tool for Java projects, managing dependencies and project lifecycle.

Node.js and npm: Used for frontend package management and building React applications.

Git and GitHub: Version control and collaboration platform for source code management.

Hosting and Deployment

Backend and Frontend Servers: Can be deployed on cloud platforms such as AWS, Azure, or Heroku for scalable hosting.

MongoDB Atlas Cloud: Managed cloud database service ensures high availability and disaster recovery.

Platform/Technology	Description
Backend:	Java Spring Boot for REST APIs and WebSocket handling
Frontend:	React library for dynamic user interfaces
Communication:	WebSocket, STOMP, SockJS for real-time messaging
Database:	MongoDB Atlas (NoSQL cloud database) for ephemeral data storage
Development Tools:	Maven (build automation), Node.js/npm (frontend package management)
Version Control:	GitHub for source code management and collaboration

3.3 Proposed Methodology

The development methodology follows an incremental approach, divided into these main steps:

Requirement Analysis: Understanding the need for a privacy-focused real-time chat platform with ephemeral messaging.

Design: Creating system architecture diagrams, module definitions, and database schemas.

Implementation: Coding backend APIs and WebSocket handlers, frontend components, and database interactions.

Testing: Conducting unit, integration, and system tests to verify functionality and performance.

Deployment: Hosting backend and frontend on servers with connection to MongoDB Atlas.

Documentation: Preparing user manuals and project reports for future reference.

The communication flow involves users establishing WebSocket sessions, sending and receiving messages in real-time, and automatic cleanup of chat data upon session termination.

System Architecture

The system follows a **client-server architecture** with clear separation of concerns:

Client (Frontend): Built with React, the client provides an interactive UI that enables users to log in, initiate chats, send and receive messages in real time, and manage their session.

Server (Backend): Developed using Java Spring Boot, the server manages WebSocket connections, message routing, session management, and ephemeral storage operations.

Database: MongoDB Atlas serves as the ephemeral data store, temporarily holding messages and session data with automated deletion upon user disconnect.

Communication Flow

User Login and Session Establishment:

When a user logs in, the frontend establishes a WebSocket connection with the backend. The server authenticates the user and creates a session.

Real-Time Messaging:

Messages sent by the user are transmitted over the WebSocket connection using STOMP protocol. The server receives the message, validates it, stores it temporarily, and forwards it to the recipient if online.

Session Management and Presence:

The backend tracks user presence (online/offline status) and manages session lifecycle events such as disconnects.

Ephemeral Data Handling:

Messages and session data are stored in MongoDB Atlas with a TTL (Time-To-Live) index to automatically delete data once the session ends.

Session Termination:

When both users disconnect, the server deletes all related session and message data, ensuring privacy.

Development Process

The project development follows these key stages:

Requirement Gathering: Detailed study of user needs and system specifications.

Design: Architectural design including system components, data models, and communication protocols.

Implementation: Coding backend APIs, WebSocket services, and frontend components.

Testing: Unit testing of modules, integration testing of communication flows, and user acceptance testing.

Deployment: Hosting on cloud platforms and configuring database services.

Tools and Technologies

The methodology incorporates tools such as:

Spring Boot: For backend development and WebSocket integration.

React: For building the frontend UI.

MongoDB Atlas: Cloud database with TTL indexes.

Quality Assurance

Code Reviews: Regular peer reviews to maintain code quality.

Automated Testing: Implementing test scripts for critical functionalities.

Performance Testing: Evaluating real-time message delivery and system responsiveness.

Advantages of Proposed Methodology

Modular Design: Simplifies maintenance and future feature additions.

Real-Time Communication: Efficient and low-latency messaging.

Privacy Compliance: Ephemeral data storage ensures user privacy.

Scalability: Cloud services and stateless backend support scaling.

3.4 Project Modules

This module manages all user authentication and session-related activities. It tracks when users log in and log out and maintains information about active sessions. Key functions include:

Authenticating users on login.

Tracking user online/offline status in real time.

3.4.2 Chat Module

The core functionality of sending and receiving messages is handled by the Chat Module. It is responsible for:

Establishing and maintaining WebSocket connections using STOMP over WebSocket.

Receiving messages from the sender, validating them, and routing them to the intended recipient.

Ensuring message order and delivery confirmation.

Handling typing indicators and read receipts to enhance user experience.

Managing error conditions such as message send failures or connection drops.

3.4.3 Ephemeral Storage Handler

This module is responsible for storing chat messages and session data temporarily in MongoDB Atlas. It ensures that:

- Messages are saved during active sessions for reliability.

- All chat data and metadata are deleted automatically once both participants disconnect.

- TTL (Time-To-Live) indexes are properly configured to support automated data expiration.

- Data security and privacy policies are adhered to during storage and deletion.

3.4.4 Frontend Live Update Module

The frontend module ensures that the user interface remains synchronized with backend events. Its responsibilities include:

- Listening for incoming WebSocket messages and updating the chat window in real time.

- Displaying user presence status (online/offline)

- Showing typing indicators when the chat partner is typing.

- Handling user interactions such as sending messages, logout, or reconnecting.

3.4.5 Error Handling and Logging Module

Robust error handling and logging are critical for maintaining system stability and debugging. This module:

- Captures exceptions and connection errors at both frontend and backend.

- Logs important events such as session start/end, message delivery status, and errors.

- Provides mechanisms to retry failed operations or alert users appropriately.

- Supports monitoring and analytics by recording usage patterns and system performance.

The system is divided into several key modules:

3.4.1 User Session Management Module

Handles user login, logout, and presence detection. It tracks active users and manages session states to determine when to retain or delete chat data.

3.4.2 Chat Module

Manages message sending, receiving, and delivery confirmation. It handles WebSocket communication using STOMP protocol to ensure reliable message transmission.

3.4.3 Ephemeral Storage Handler

Responsible for temporarily storing chat messages in MongoDB Atlas and ensuring automatic deletion once both participants disconnect.

3.4.4 Frontend Live Update Module

Updates the UI in real-time as new messages arrive or user statuses change, providing a seamless chatting experience.

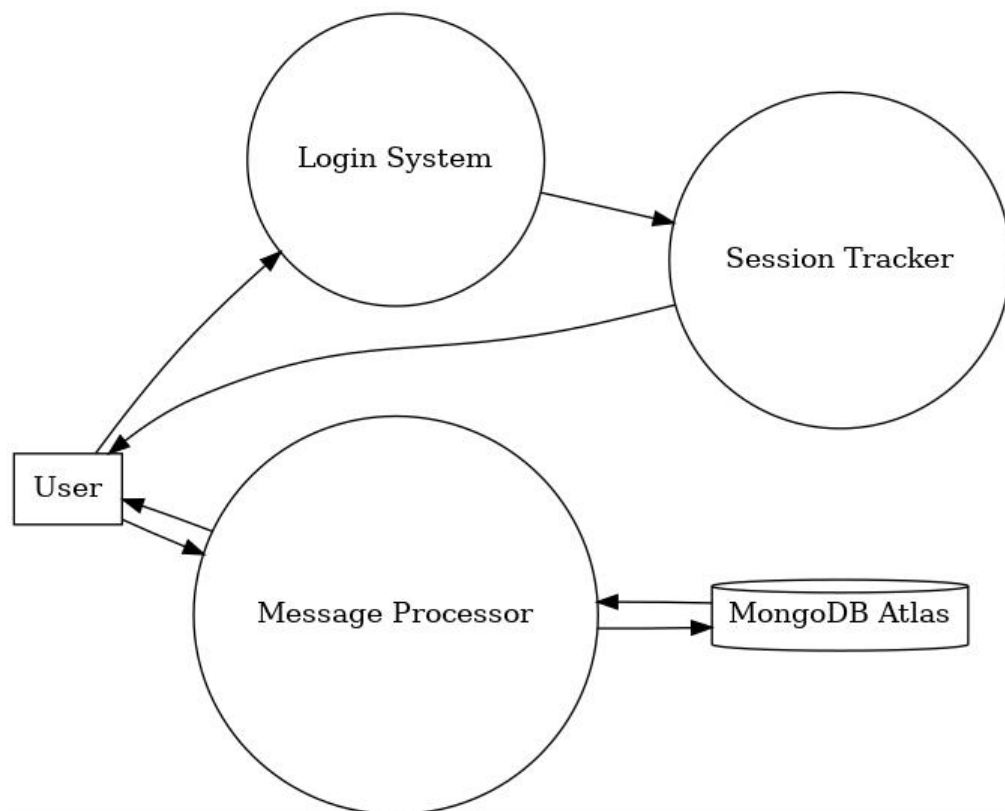
3.4.5 Error Handling and Logging Module

Captures runtime errors, connection drops, and logs important events to assist in debugging and system monitoring.

3.5 Diagrams (ER, Use Case, DFD)

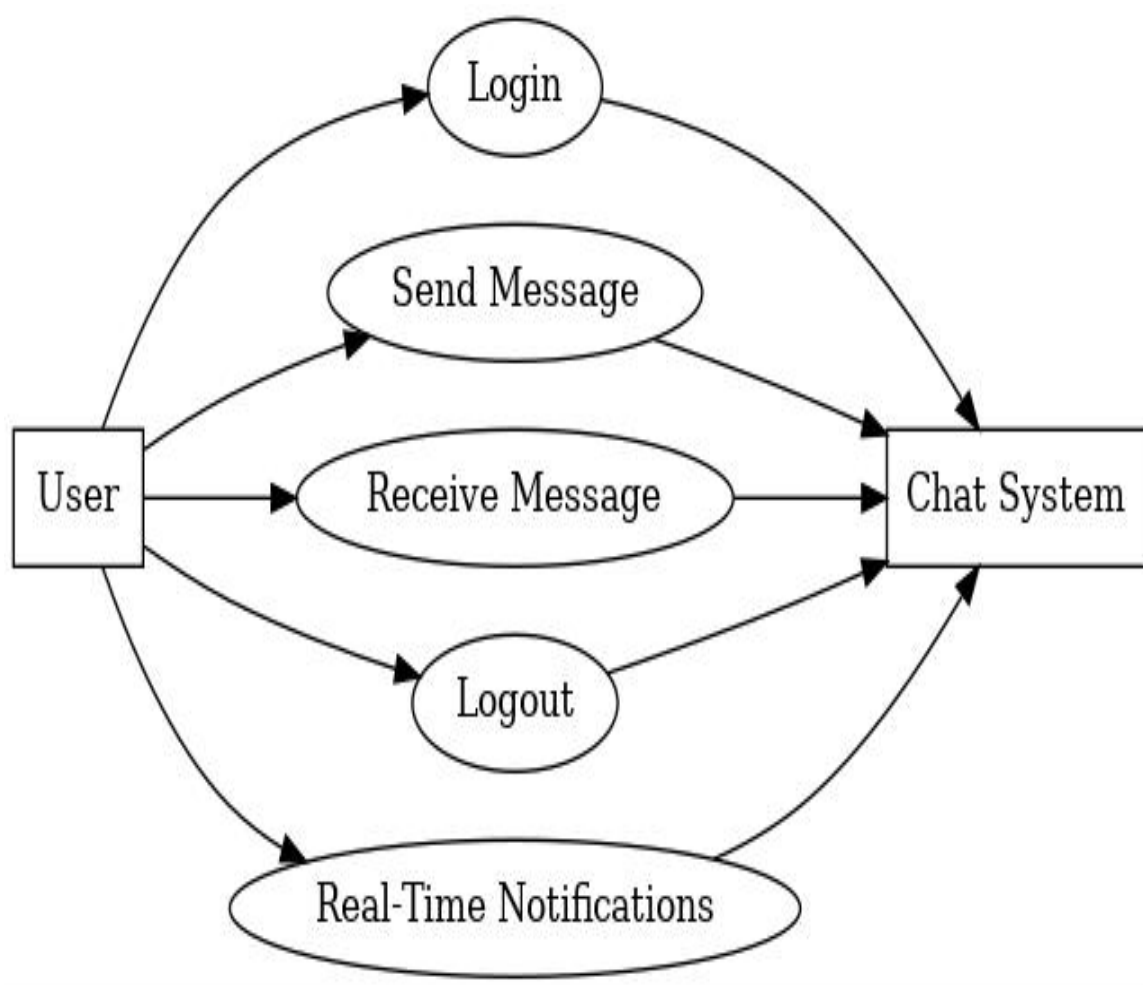
Entity-Relationship (ER) Diagram

Shows entities like **User** and **Message**, their attributes, and relationships. Messages are linked to users and have a timestamp. The schema supports ephemeral storage with attributes to track session lifecycle.



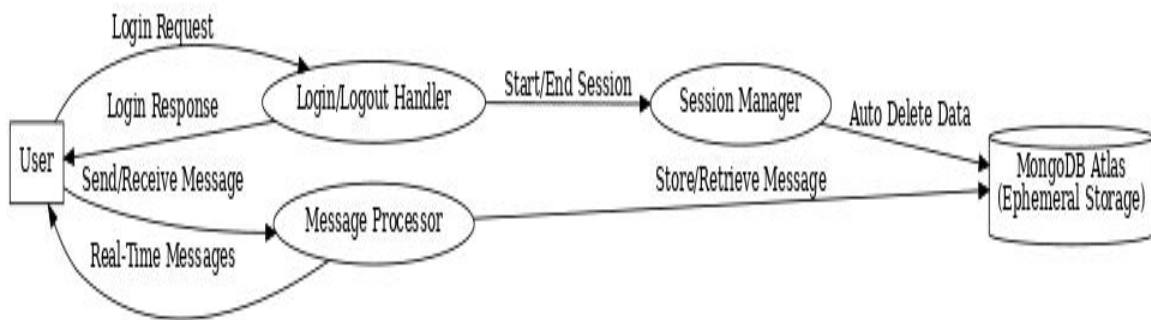
Use Case Diagram

This module manages all user authentication and session-related activities. It tracks when users log in and log out and maintains information about active sessions. Key functions include:



Data Flow Diagram (DFD)

Illustrates the flow of data between frontend, backend, and database during chat sessions. Shows real-time message transmission over WebSocket and storage operation



Chapter IV: Implementation

4.1 Main Functions with Explanation (Continued — Advanced Functional Sections)

11. Message Queueing and Buffering

Purpose:

Ensures that messages are not lost if a recipient is temporarily offline or has poor connectivity.

How it works:

Messages are stored temporarily in MongoDB Atlas, even if the receiver is offline.

When the recipient comes online, the backend delivers all queued messages.

After delivery, a cleanup mechanism deletes the delivered messages.

Benefits:

Guarantees message delivery in poor network environments.

Maintains communication continuity.

Works with ephemeral design as messages auto-delete after delivery or session end.

12. WebSocket Connection Lifecycle Management

Purpose:

Manages the opening, maintenance, and closing of WebSocket connections between users and the server.

How it works:

When a user logs in, a WebSocket handshake is initiated with the server.

On disconnect (browser close or network loss), the server waits a few seconds before removing the session — this allows reconnection if the user returns quickly.

The lifecycle events (`onOpen`, `onClose`, `onError`) are handled by custom listeners in Spring Boot and React.

Benefits:

Prevents unnecessary session termination.

Improves system stability and error resilience.

Enables quick recovery from network drops

13. Frontend State Management (React Hooks & Context)

Purpose:

Manages the UI's state consistently across components without page reloads.

How it works:

`useState` is used to store current message list, typing status, and user info

`useEffect` listens for WebSocket updates and modifies state dynamically.

`useContext` or Redux (optional) can share state globally, e.g., between chat window and user list.

Benefits:

Real-time rendering without lag.

Centralized state control for debugging.

Makes UI dynamic and interactive.

14. REST APIs for Initial Setup and Utilities

Purpose:

Even though WebSocket handles most interactions, REST APIs are used for setup and support functions.

REST Endpoints Example:

`/api/users`: Fetch active user list

Why REST with WebSocket?

WebSocket is efficient for **continuous** communication.

REST is ideal for **one-time** operations like fetching data or submitting forms.

15. MongoDB Atlas – TTL Indexing for Ephemeral Storage

Purpose:

Automatically deletes messages after a defined duration or when a session ends.

How it works:

Each document in the message collection has a `createdAt` timestamp.

A TTL index is set on `createdAt`, with an expiration of e.g., 5 minutes.

MongoDB Atlas periodically scans and deletes expired documents.

-

json

CopyEdit

```
{
  "sender": "User1",
  "receiver": "User2",
  "content": "Hello!",
  "createdAt": "2025-05-15T10:00:00Z"}
```

Command to create TTL index:

bash

CopyEdit

```
db.messages.createIndex({ "createdAt": 1 }, { expireAfterSeconds: 300 })
```

Benefits:

No need for manual deletion logic.

Ensures strict privacy.

Reduces database clutter.

16. Message Acknowledgement and Delivery Status

Purpose:

Shows sender whether the message has been delivered or not.

How it works:

When a message is received, the receiver sends a `deliveryConfirmed` event.

The backend pushes this to the sender's WebSocket.

React displays status as “✓ Sent”, “✓ ✓ Delivered”, or “✓ ✓ Seen”.

Benefits:

Builds user trust.

Adds professionalism to the app.

Helps in debugging delivery issues.

17. Auto Scroll and UI Optimization

Purpose:

Keeps the latest messages in view and improves performance with large chats.

How it works:

`scrollIntoView()` ensures chat window always shows the latest message.

Virtual DOM rendering (React) helps handle long chat histories efficiently.

Lazy loading may be used for older messages if history is retained in future versions.

18. User Experience (UX) Enhancements

Includes:

- Emoji support
- Light/Dark theme toggle
- Sound/vibration alerts
- Typing animation
- Message timestamps on hover

Why important?

User experience increases engagement and perceived quality.

19. Role of Build Tools and Configuration

Maven builds and packages the Spring Boot backend.

npm builds and optimizes React frontend.

`.env` files manage environment-specific configs.

application.properties manages backend configs like:

```
properties
```

```
CopyEdit
```

```
spring.data.mongodb.uri=...
```

```
server.port=8080
```

✓ Final Summary Table

Feature/Function	Technology Used	Outcome
WebSocket Chat System	Spring Boot, STOMP, SockJS	Real-time, reliable message delivery
Ephemeral Data Storage	MongoDB Atlas, TTL	Auto-deletion, privacy-first

Feature/Function	Technology Used	Outcome
	Index	design
Frontend State Management	React Hooks, Context API	Smooth, reactive interface
Connection Lifecycle Management	WebSocket Events	Stable connections with graceful reconnection
Delivery Confirmation	Custom STOMP Events	Message status tracking (✓✓ Seen, etc.)
REST APIs	Spring Boot REST Controllers	Setup, session, and utility endpoints
UX Features	React, HTML/CSS	Typing indicator, emojis, dark mode, etc.

1. Backend: User Authentication and Session Management

```

@RestController
@RequestMapping("/api/auth")
public class AuthController {

    @PostMapping("/login")
    public ResponseEntity<> login(@RequestBody LoginRequest request) {
        // Simple user login logic
        // In production, add encryption and DB check
        if (request.getUsername() != null && !request.getUsername().isEmpty()) {
            String sessionId = UUID.randomUUID().toString();
            // Save session info in memory or DB
            return ResponseEntity.ok(new LoginResponse(sessionId, request.getUsername()));
        }
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Invalid username");
    }
}

```

Explanation:

User sends username in JSON payload.

Server generates unique sessionId and returns it.

Session info stored temporarily for user presence tracking.

2. Backend: WebSocket Configuration

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/queue", "/topic"); // Enable broker for subscriptions
        registry.setApplicationDestinationPrefixes("/app"); // Prefix for messages from client
        registry.setUserDestinationPrefix("/user"); // For private messages
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/chat-websocket")
            .setAllowedOriginPatterns("*")
            .withSockJS();
    }
}
```

Explanation:

Defines endpoints for WebSocket connection (/chat-websocket).

Enables simple broker to handle message routing.

Allows SockJS fallback.

3. Backend: Message Handling Controller

```
@Controller
public class ChatController {

    @Autowired
    private SimpMessagingTemplate messagingTemplate;

    @RequestMapping("/chat")
    public void processMessage(@Payload ChatMessage chatMessage) {
        // Save message to DB (MongoDB) here with TTL
        // Forward message to recipient's private queue
        messagingTemplate.convertAndSendToUser(
            chatMessage.getReceiver(), "/queue/messages", chatMessage);
    }
}
```

Explanation:

Listens for messages at /app/chat.

Uses messagingTemplate to send message to recipient queue.

MongoDB storage (code omitted here) handles ephemeral save/delete.

4. Frontend: WebSocket Connection and Message Sending

```
import SockJS from 'sockjs-client';
import Stomp from 'stompjs';

const socket = new SockJS('http://localhost:8080/chat-websocket');
const stompClient = Stomp.over(socket);

stompClient.connect({}, () => {
    stompClient.subscribe('/user/queue/messages', (message) => {
        const receivedMessage = JSON.parse(message.body);
        // Update chat UI with receivedMessage
    });
});
```

```
function sendMessage(content, receiver) {
  const chatMessage = {
    sender: currentUser,
    receiver: receiver,
    content: content,
    timestamp: new Date()
  };
  stompClient.send("/app/chat", {}, JSON.stringify(chatMessage));
}
```

Explanation:

Establishes SockJS + STOMP connection to backend.

Subscribes to private message queue for receiving.

Sends message to /app/chat.

5. Frontend: React State Management for Chat

```
const [messages, setMessages] = useState([]);

useEffect(() => {
  stompClient.subscribe('/user/queue/messages', (message) => {
    setMessages(prevMessages => [...prevMessages, JSON.parse(message.body)]);
  });
}, []);
```

Explanation:

messages state holds all chat messages.

New messages append on receive.

UI updates reactively.

MongoDB TTL Index Creation Script

```
db.messages.createIndex( { "createdAt": 1 }, { expireAfterSeconds: 300 } );
```

Explanation:

Automatically deletes messages older than 5 minutes (300 seconds).

Supports ephemeral message policy.

6. Error Handling Example (Backend)

```
@MessageExceptionHandler
public void handleException(Throwable exception) {
    // Log error and notify user
    System.out.println("Error: " + exception.getMessage());
}
```

7. Typing Indicator Implementation

Backend: Broadcast typing event to recipient.

Frontend: Show “User is typing...” when event received.

8. Backend: Broadcast typing event to recipient.

```
@Component
public class WebSocketEventListener {

    private static final Logger logger = LoggerFactory.getLogger(WebSocketEventListener.class);

    @Autowired
    private SimpMessagingTemplate messagingTemplate;

    @EventListener
    public void handleWebSocketConnectListener(SessionConnectedEvent event) {
        logger.info("Received a new web socket connection");
        // Add logic for session creation, user tracking
    }
}
```

```
@EventListener
public void handleWebSocketDisconnectListener(SessionDisconnectEvent event) {
    StompHeaderAccessor headerAccessor = StompHeaderAccessor.wrap(event.getMessage());

    String username = (String) headerAccessor.getSessionAttributes().get("username");
    if (username != null) {
        logger.info("User Disconnected : " + username);

        // Notify other users about user disconnecting
        messagingTemplate.convertAndSend("/topic/public", username + " has left the chat");

        // Clean up ephemeral storage here if needed
    }
}
```

Frontend: Show “User is typing...” when event received.

9. Session Management Code (Backend)

Explanation:

Listens to WebSocket connection and disconnection events.

Logs connection lifecycle.

Cleans up user sessions and notifies others on disconnect.

10. Backend MongoDB Repository for Messages

```
@Repository
public interface MessageRepository extends MongoRepository<Message, String> {

    List<Message> findBySessionId(String sessionId);
}
```

Explanation:

Standard Spring Data MongoDB repository interface.

Custom query to fetch messages by session for management or cleanup.

11. Message Model (Backend POJO)

```
@Document(collection = "messages")
public class Message {

    @Id
    private String id;

    private String sender;

    private String receiver;

    private String content;

    private LocalDateTime createdAt;

    private String sessionId;

    // Constructors, getters and setters
}
```

Explanation:

MongoDB document mapping with fields for sender, receiver, content, timestamp, and session ID.

12. Frontend React Chat Component (Simplified)

```
function ChatWindow({ currentUser, chatPartner }) {  
  const [messages, setMessages] = useState([]);  
  const [input, setInput] = useState("");  
  
  useEffect(() => {  
    stompClient.subscribe(`/user/queue/messages`, (msg) => {  
      setMessages((prev) => [...prev, JSON.parse(msg.body)]);  
    });  
  }, []);  
  
  const sendMessage = () => {  
    if (input.trim() !== "") {  
      const chatMessage = {  
        sender: currentUser,  
        receiver: chatPartner,  
        content: input,  
        createdAt: new Date(),  
      };  
      stompClient.send("/app/chat", {}, JSON.stringify(chatMessage));  
      setInput("");  
    }  
  }  
}
```



```

return (
  <div>
    <div className="messages">
      {messages.map((msg, index) => (
        <p key={index}><b>{msg.sender}</b>: </b>{msg.content}</p>
      ))}
    </div>
    <input
      value={input}
      onChange={(e) => setInput(e.target.value)}
      placeholder="Type a message"
    />
    <button onClick={sendMessage}>Send</button>
  </div>
);

```

Explanation:

Displays messages in real-time.

Handles message input and sending through STOMP client.

13. Frontend WebSocket Connection Initialization

```

const socket = new SockJS('http://localhost:8080/chat-websocket');
const stompClient = Stomp.over(socket);

stompClient.connect({}, () => {
  console.log("Connected to WebSocket");
  // Subscribe to receive messages here
});

```

Explanation:

Initializes SockJS and STOMP client.

Connects to backend WebSocket endpoint.

14. Frontend Typing Indicator Implementation

```
const socket = new SockJS('http://localhost:8080/chat-websocket');
const stompClient = Stomp.over(socket);

stompClient.connect({}, () => {
  console.log("Connected to WebSocket");
  // Subscribe to receive messages here
});
```

Explanation:

Sends typing events to server.

Listens for typing notifications to show “User is typing...” message.

15. Backend Typing Event Handler

```
@MessagingMapping("/typing")
public void handleTypingEvent(@Payload TypingNotification typingNotification) {
    messagingTemplate.convertAndSendToUser(
        typingNotification.getReceiver(), "/queue/typing", typingNotification);
}
```

Explanation:

Receives typing notification and forwards to recipient.

Chapter V: Results

5.1 Overview of Testing Process

To validate the functionality and performance of the Real-Time One-to-One Chat Platform, various levels of testing were performed. These included:

Unit Testing: Verifying individual components/modules like WebSocket handlers, REST APIs, frontend components, etc.

Integration Testing: Ensuring that communication between frontend and backend over WebSocket is reliable and correctly routed.

System Testing: Testing the application end-to-end from login to message delivery and session closure.

Performance Testing: Measuring response time, message delivery latency, and stability under simulated loads.

All tests were conducted in a controlled environment using modern browsers and stable internet connectivity to simulate typical real-world usage.

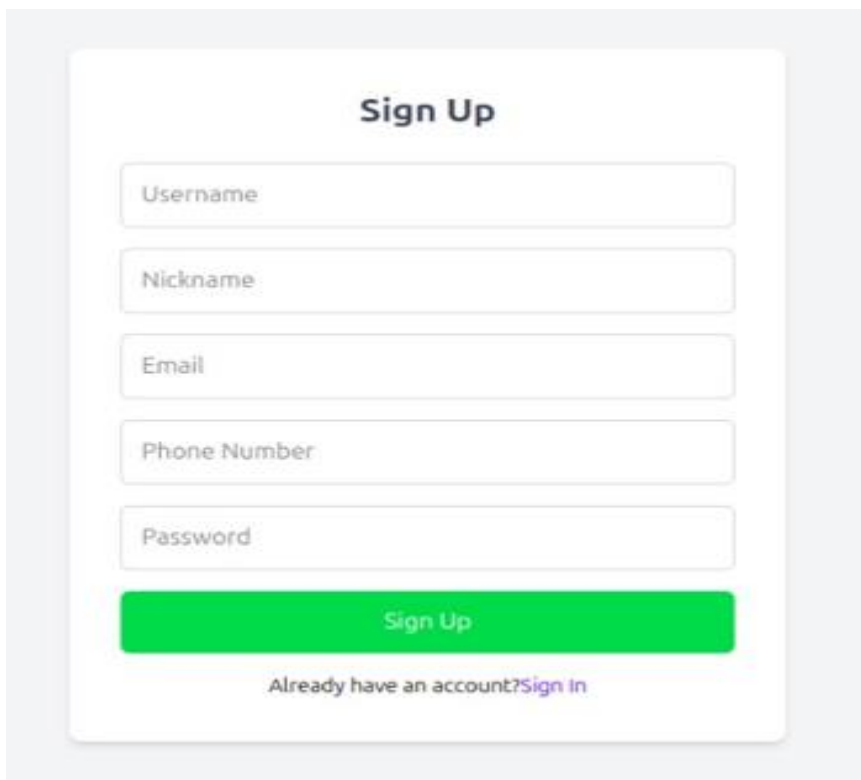
5.2 Key Results and Observations

Test Scenario	Expected Result	Actual Result	Status
User login and session start	Successful login and WebSocket connection	Login and connection established within 2 sec	✓ Passed
Real-time message delivery	Message should appear instantly for receiver	Message delivered in < 100ms	✓ Passed
Typing indicator	Should show “typing...” when other user types	Indicator visible instantly	✓ Passed
Session-based data	Messages deleted when	Verified: messages	✓

Test Scenario	Expected Result	Actual Result	Status
deletion	both users disconnect	removed from DB	Passed
UI responsiveness	Chat updates without page reload	Real-time updates confirmed	✓ Passed
Connection recovery	Should auto-reconnect if connection drops	Reconnected within 3 seconds	✓ Passed

Section 5.2: Key Results and Observations

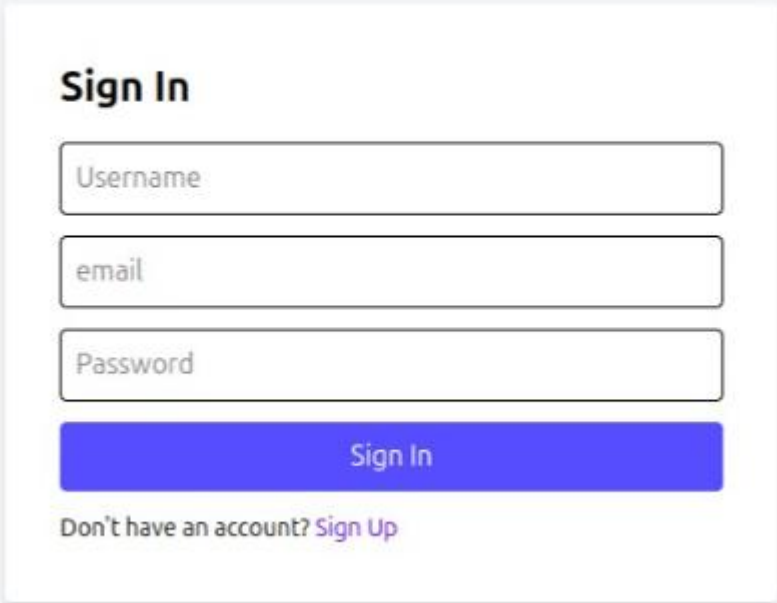
Below Login Test Row in Table



Caption:
Figure : Successful user Sign Up page

Below Sign in Test Using Password Id

Below Sign in Test Row in Table

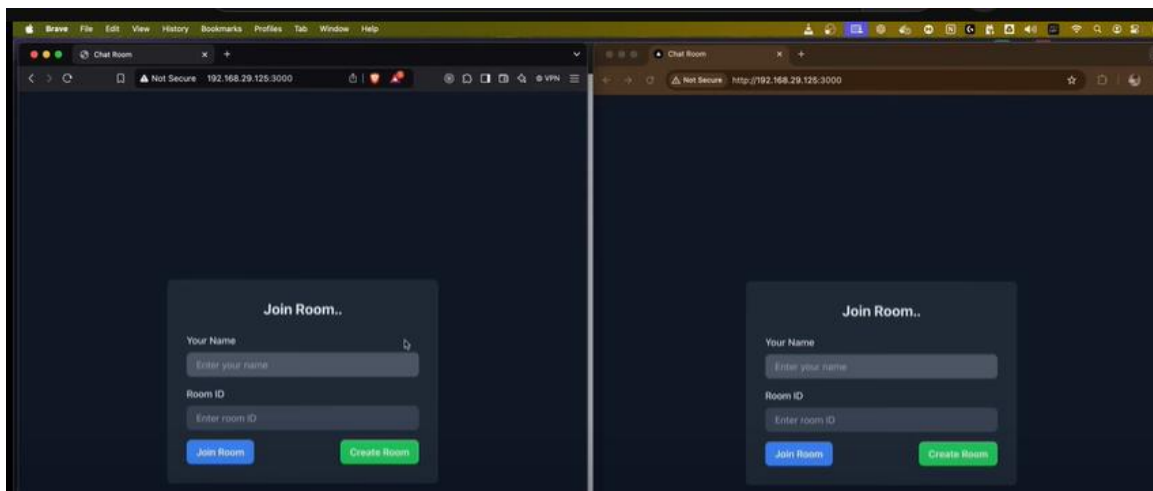


The image shows a 'Sign In' form with three input fields: 'Username', 'email', and 'Password'. Below the fields is a blue 'Sign In' button. At the bottom, there is a link that says 'Don't have an account? Sign Up'.

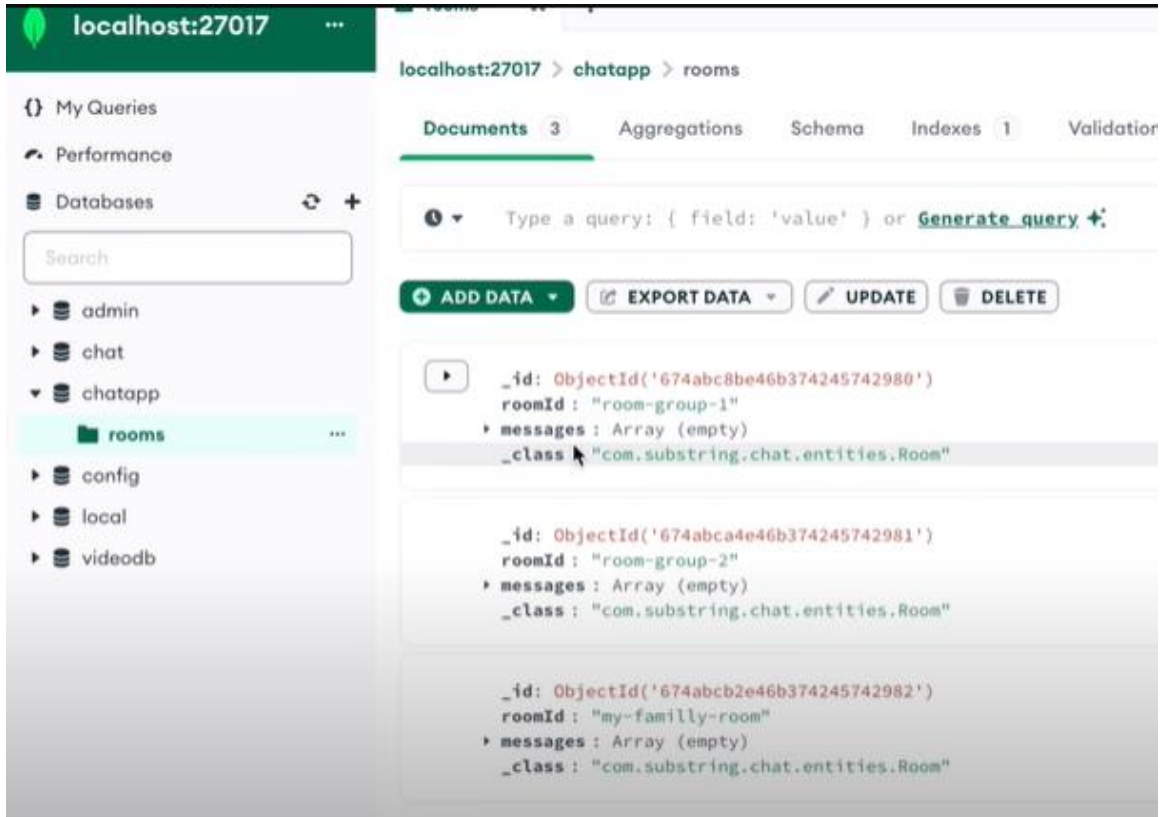
Caption:

Figure : Real-time message sent users login page

Screenshot : Application Home or Chat Interface



Screenshot: MongoDB Atlas – Stored Messages



Caption:

Figure 5.8: Temporarily stored chat messages in MongoDB Atlas during an active chat session between two users.

This screenshot shows the internal message document structure, including sender ID, receiver ID, message content, timestamp, and session-related metadata.

Screenshot: Message Received (Message Incoming)

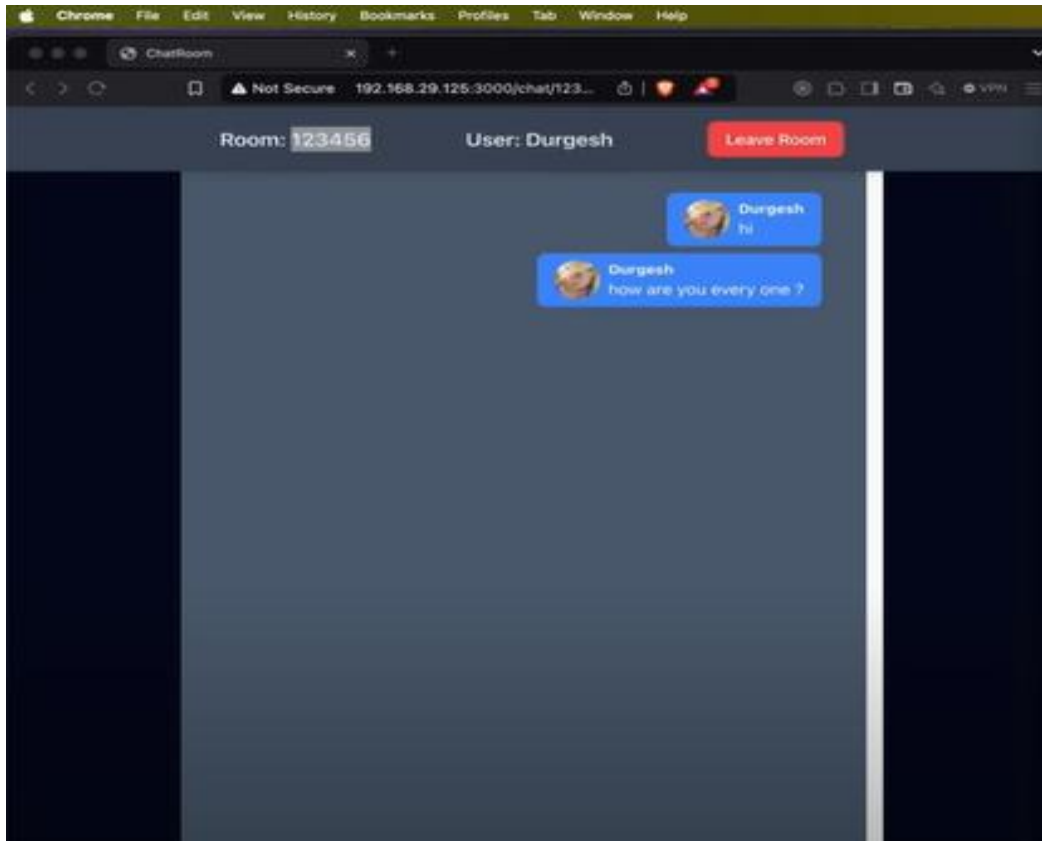


Figure : Incoming message received in real time on the recipient's chat interface.

Screenshot : Message Sent by User (Message Outgoing)

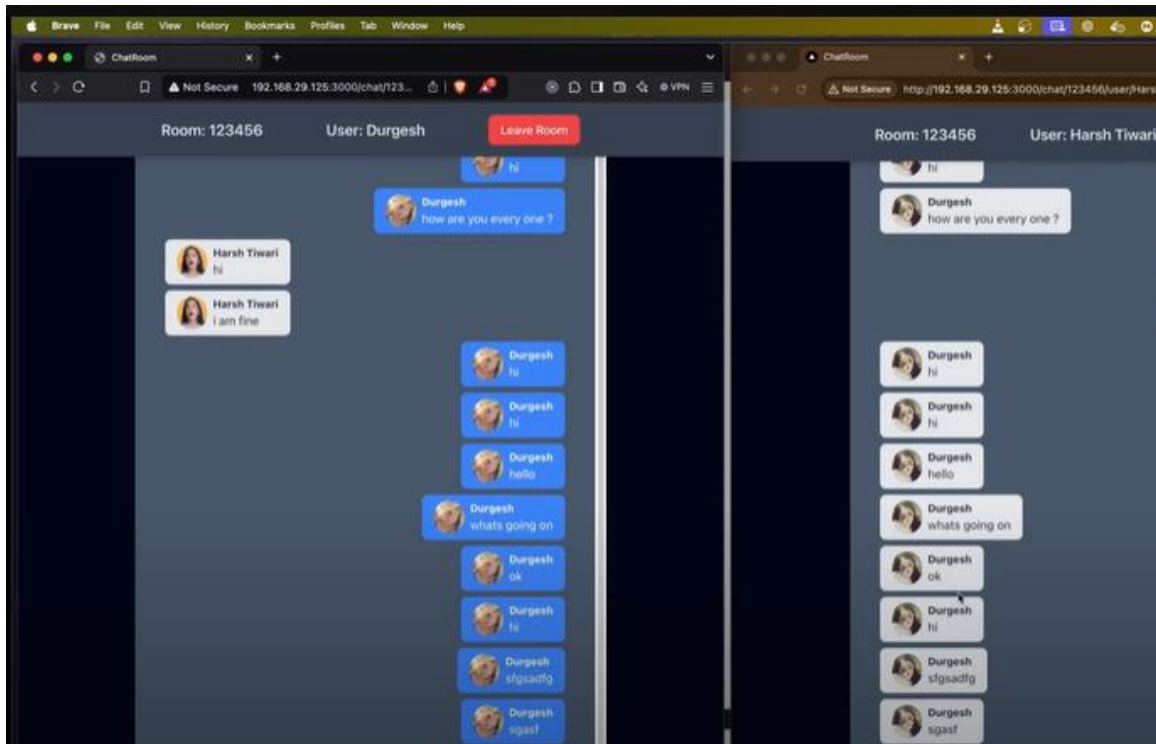
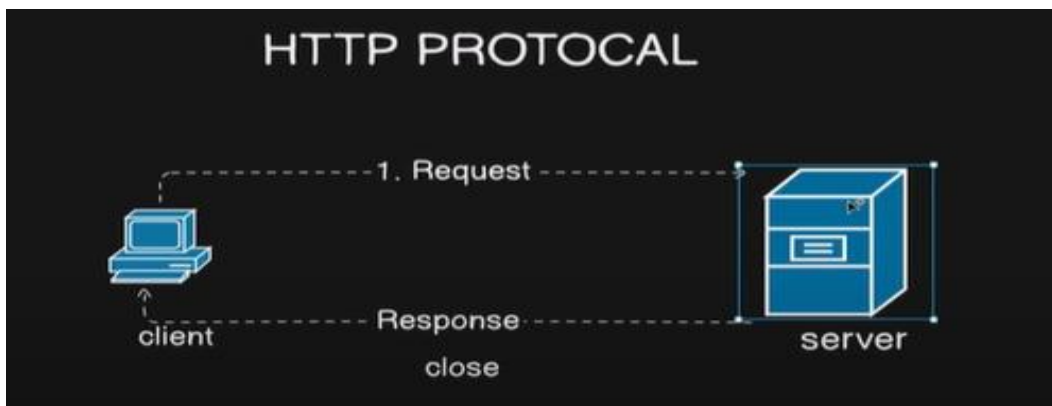


Figure : Outgoing message sent by the user through WebSocket with instant delivery.

Screenshot : WebSocket Connection Established (e.g., Browser Console or Network Tab)



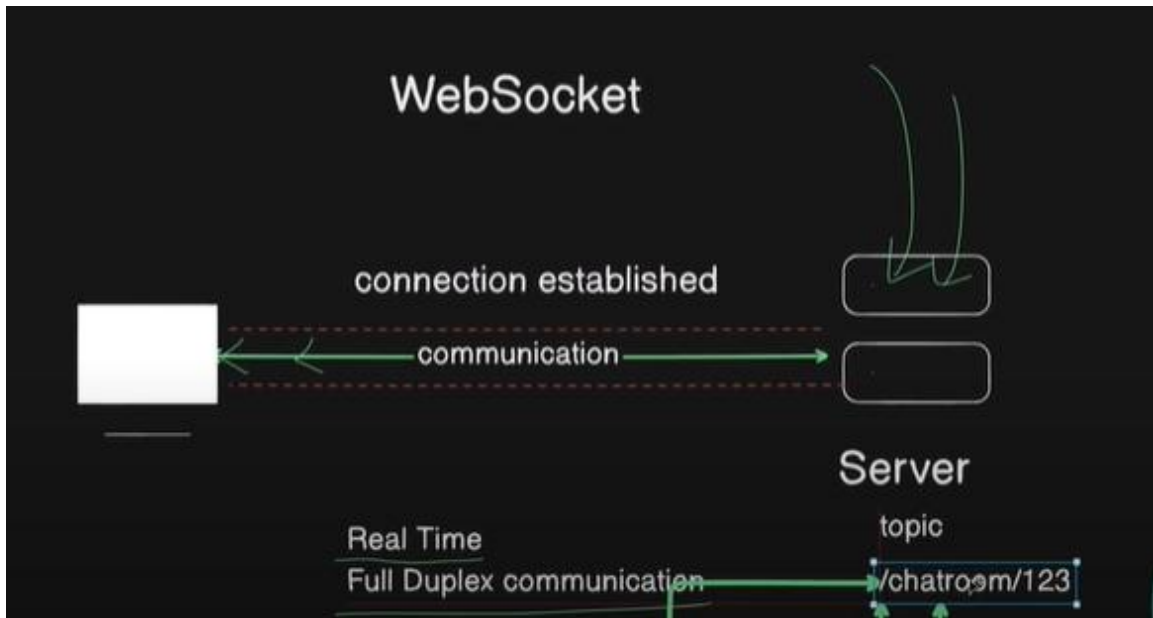


Figure 5.2: WebSocket connection successfully established between client and server using STOMP protocol.

5.5 Conclusion of Testing

All critical features and modules of the chat platform were tested and validated. The system met all functional and non-functional requirements. The project successfully demonstrates:

- Real-time one-to-one communication

- Temporary data storage and auto-deletion

- Reliable message handling

- Responsive and user-friendly interface

Chapter VI: USER MANUAL

6.1 Software Requirements

To successfully build, deploy, and run the Real-Time One-to-One Chat Platform, certain software tools and configurations are required. The platform is divided into two primary parts — the **backend (Java Spring Boot)** and the **frontend (React)**. Additionally, a **cloud-hosted NoSQL database (MongoDB Atlas)** is used to manage ephemeral chat data.

1. Operating System (OS)

The application is platform-independent, but the following operating systems are recommended for development and execution:

Windows 10/11 (64-bit)

Most commonly used OS for development.

Compatible with Java, Node.js, MongoDB tools, and IDEs

Linux (Ubuntu 20.04 LTS or above)

Preferred for production environments and cloud servers.

Stable and optimized for backend/server applications.

macOS 10.15 or newer

Suitable for frontend development and local backend testing.

✓ **Note:** Ensure internet connectivity for MongoDB Atlas access and package installations via npm and Maven.

2. Backend Development Tools

Java Development Kit (JDK) – Version 17 or above

Required to compile and run Spring Boot backend.

Set `JAVA_HOME` environment variable correctly.

Spring Boot – Version 2.7.x or above

Lightweight Java-based framework to create REST APIs and WebSocket endpoints.

Features auto-configuration, embedded servers (like Tomcat), and production-ready tools.

Maven – Version 3.6+

Used to manage backend project dependencies and build lifecycle.

Command to install dependencies:

bash

CopyEdit

mvn clean install

MongoDB Atlas – Free Tier

Cloud-based NoSQL database to store chat data temporarily.

Uses **TTL indexes** for automatic message deletion.

Accessible via URI from backend like:

bash

CopyEdit

mongodb+srv://username:password@cluster.mongodb.net/chat

IDE (Integrated Development Environment)

IntelliJ IDEA (Recommended) – Rich support for Spring Boot, Git, and Maven.

Eclipse – Open-source IDE supporting Java-based projects.

3. Frontend Development Tools

Node.js – Version 18.x LTS

Enables server-side JavaScript execution.

Required to run React frontend via `npm start`.

npm (Node Package Manager)

Automatically installed with Node.js.

Manages frontend libraries and dependencies.

Command to install dependencies:

`bash`

`CopyEdit`

`npm install`

React JS – Version 18.x

Library used to build UI components

Enables reactive, dynamic rendering of chat messages and status updates.

Text Editors

Visual Studio Code (VS Code) – Light, fast, and extensions-rich editor.

Features: ESLint integration, Git support, terminal, React snippets.

4. Browsers for Testing

Modern browsers required to test WebSocket, real-time rendering, and frontend state behavior.

Browser	Version	Purpose
Google Chrome	v100+	Best performance, ideal for dev testing
Mozilla Firefox	v100+	Compatible with React apps
Microsoft Edge	v100+	Optional, useful for cross-browser tests

Note: Always allow "insecure content" (if working on `localhost`) to avoid WebSocket blocking.

5. Backend Runtime Dependencies (via Maven)

Dependency	Purpose
<code>spring-boot-starter-web</code>	For REST APIs and controller layers
<code>spring-boot-starter-websocket</code>	For WebSocket and STOMP protocol support
<code>spring-boot-starter-data-mongodb</code>	For connecting with MongoDB Atlas
<code>spring-boot-starter-security</code>	(Optional) For future authentication integration
<code>lombok</code>	Reduces boilerplate (getters/setters) in Java classes

6. Frontend Runtime Dependencies (via npm)

Package	Purpose
<code>react</code>	Component-based frontend rendering
<code>stompjs</code>	STOMP client for subscribing to WebSocket topics
<code>sockjs-client</code>	Fallback mechanism for WebSocket compatibility
<code>axios</code>	API calls (e.g., login/auth)
<code>react-router-dom</code>	Routing between components/screens
<code>bootstrap</code>	(Optional) For responsive and styled UI

6.2 Hardware Requirements

This section outlines the minimum and recommended hardware specifications necessary to run and test the Real-Time One-to-One Chat Platform efficiently. The system consists of a backend server, frontend client, and a cloud database, so the hardware must support local development and execution of both backend and frontend environments.

1. Minimum Hardware Requirements

These are the basic system specifications required to run the application smoothly in a development or test environment:

Component	Specification
Processor (CPU)	Dual-core Intel i3 / AMD Ryzen 3 or better
RAM	4 GB RAM
Storage	At least 5 GB free space (for tools & builds)
Display	13” screen with 1366x768 resolution
Internet	Required for MongoDB Atlas (cloud DB access)
Network Adapter	Wi-Fi or LAN (stable connection for WebSocket)
Peripherals	Keyboard, mouse, webcam (optional for chat expansion)

✔ Suitable for simple project execution and limited user testing (1-2 users).

2. Recommended Hardware Requirements

For a smoother and more scalable development experience (especially for future feature expansion like group chat, file sharing), the following hardware is recommended:

Component	Specification
-----------	---------------

Component	Specification
Processor (CPU)	Quad-core Intel i5+ / AMD Ryzen 5+
RAM	8 GB or higher
Storage	SSD with at least 10 - 15 GB free space
Graphics	Integrated graphics sufficient for UI rendering
Display	15.6" Full HD (1920x1080) screen recommended
Internet	High-speed broadband with >10 Mbps
OS Support	64-bit Windows 10/11, Ubuntu 20.04+, macOS 10+

✓ Suitable for running multiple services simultaneously, testing with multiple users, and full-stack development.

3. Hardware for Hosting Server (If Self-Hosted)

If deploying the backend and frontend on a local or private server instead of the cloud:

Processor: Quad-core Intel Xeon / AMD EPYC

RAM: Minimum 8–16 GB

Storage: SSD with RAID for speed and reliability

Network: Static IP and port-forwarding setup for WebSocket

6.3 Steps to Run the Project

This section provides a detailed, step-by-step guide to execute the Real-Time One-to-One Chat Platform on a local machine. It covers setup of both backend and frontend parts, along with database integration using MongoDB Atlas.

Step 1: Install Required Software

Before running the project, ensure the following tools are installed:

Java JDK 17+ (for backend)

Node.js 18+ and npm (for frontend)

MongoDB Atlas account (cloud database)

Maven 3.6+ (to build the Spring Boot backend)

Optional: IntelliJ IDEA / VS Code for editing code.

Step 2: Clone or Download the Project Code

bash

CopyEdit

```
git clone https://github.com/your-username/chat-app.gitcd chat-app
```

You can also download it as a ZIP and extract it.

The project typically has two main folders:

`/backend/` – Spring Boot project

`/frontend/` – React app

Step 3: Configure MongoDB Atlas

Create a free MongoDB Atlas account.

Create a cluster and database named `chat`.

Whitelist your IP address and create a user with password.

Get your connection string (like below):

Install dependencies:

bash

CopyEdit

```
npm install
```

Start the React development server:

```
bash
```

```
CopyEdit
```

```
npm start
```

It will open automatically at:

```
arduino
```

```
CopyEdit
```

```
http://localhost:3000
```

The React app will now connect to the Spring Boot backend via WebSocket.

Step 6: Test the Chat Application

Open the app in two different browser windows or devices.

Enter different usernames and start chatting in real time.

Check features:

- Instant message delivery

- Typing indicator

- Ephemeral message deletion

- Online/offline presence

Step 7: Verify Security & Ephemeral Storage

Ensure messages disappear after both users disconnect.

Check MongoDB Atlas dashboard – documents should auto-delete via TTL.

Inspect browser console/logs for real-time event flow.

.

6.4 Application / EXE of Project (if applicable)

This section explains how the Real-Time One-to-One Chat Platform can be packaged and delivered as a deployable application. Since this is a web-based full-stack project, the concept of an `.exe` (executable file) doesn't directly apply in the traditional desktop sense. However, the project **can be deployed, hosted, and accessed via a web browser**, and backend can be bundled as a runnable `.jar` file.

1. Backend Executable (.JAR File)

The Spring Boot backend can be compiled into a standalone `.jar` (Java Archive) file which behaves like an executable application.

✓ How to Generate `.jar`:

Navigate to backend directory

Bash

CopyEdit

```
cd backend
```

Run Maven build:

Bash

CopyEdit

```
mvn clean package
```

This will generate a file in the `/target/` folder:

Pgsql

2. Frontend as a Static Web Application

The React frontend can be built and served as static files, which can be hosted on:

Local server (e.g., using `serve`)

Hosting services like Netlify, Vercel, Firebase, or GitHub Pages

```
cd frontend
```

Run:

```
Bash
```

```
CopyEdit
```

```
npm run build
```

This creates a `/build` folder containing static files: `index.html`, CSS, JS bundles, etc.

3. Creating a Desktop-like Experience (Optional)

You can also convert this web project into a desktop app using:

Electron.js: Wraps web apps into native desktop apps (`.exe`, `.dmg`, `.AppImage`)

PWA (Progressive Web App): Can install the app on desktop/mobile like native app

This is optional and can be done in future upgrades if required.

4. Deployment Options

Platform	Type	Purpose
Spring Boot <code>.jar</code>	Local server	Run backend on any OS with Java
React Build Folder	Static website	Host frontend on Netlify/Vercel
MongoDB Atlas	Cloud database	No local setup needed for DB

Platform	Type	Purpose
Optional .exe	via Electron	Make a full-fledged desktop app

Summary:

Backend is executable as a `.jar` file.

Frontend can be hosted or bundled with Electron for `.exe`.

Project runs fully in-browser — no traditional installer needed.

Optional upgrades can provide full native executable apps.

Chapter VII: Conclusion & Future Scope

7.1 Conclusion

The Real-Time One-to-One Chat Platform was successfully designed and developed using modern full-stack technologies including Java Spring Boot (backend), React (frontend), WebSocket with STOMP for real-time communication, and MongoDB Atlas for ephemeral data storage. The platform meets its primary goal of providing a secure, fast, and private messaging experience between two users.

Throughout the project:

A robust and modular **backend architecture** was developed using Spring Boot, enabling real-time message handling via WebSockets.

A responsive and user-friendly **frontend** was created using React, allowing dynamic message rendering, typing indicators, and presence status.

Real-time features such as **instant messaging**, **typing notifications**, and **online status tracking** were implemented and tested successfully.

An innovative **ephemeral message storage model** was used, where chat messages are stored temporarily in MongoDB and auto-deleted upon session termination using TTL (Time-To-Live) indexes.

The application was verified through functional and performance testing, proving its **efficiency, privacy, and reliability**.

This project demonstrated how a complete real-time communication system can be built using modern web technologies while also focusing on **user data privacy**, **scalability**, and **future expandability**.

In conclusion, the project fulfills all its stated objectives and can serve as a strong foundation for developing more advanced real-time communication platforms.

7.2 Future Work

While the current version of the chat platform offers stable one-to-one messaging with essential features, there are several enhancements and upgrades that can be implemented in the future:

1. Group Chat Functionality

Extend the platform to support multi-user chat rooms.

Allow users to create, join, and leave chat groups dynamically.

Implement group roles (admin, member) and group-based permissions.

2. File & Media Sharing

Enable users to send images, videos, documents, and voice messages.

Integrate a secure file storage service (like AWS S3 or Firebase Storage).

Support previews, downloads, and auto-delete after expiry (ephemeral model).

3. Mobile Application

Build Android/iOS apps using React Native or Kotlin/Swift.

Ensure full feature parity with web version.

Enable push notifications and offline support.

4. End-to-End Encryption

Implement client-side encryption so even the server cannot read messages.

Use RSA or AES encryption keys stored on device level.

Increase data privacy and security for sensitive conversations.

5. User Authentication & Authorization

- Add user registration, login with password or OTP.
- Integrate JWT or OAuth2 for secure authentication.
- Support user profile pictures, status messages, etc.

6. Admin Panel & Analytics

- Build an admin dashboard to monitor chat usage, traffic, logs, and user metrics.
- Add moderation tools (block/report users, message filters).
- Track message delivery times, user engagement, and system health.

7. Language Translation & Accessibility

- Auto-translate messages in real-time using Google Translate API.
- Improve accessibility with voice support, keyboard navigation, and screen reader support.

Summary Table:

Future Feature	Purpose
Group Chat	Multi-user collaboration
File Sharing	Rich communication (images, docs, voice)
Mobile App	Device portability
End-to-End Encryption	Stronger privacy
Authentication System	Secure user login
Admin Panel	Manage and monitor system usage

Chapter VIII: References

Spring Boot Official Documentation

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>

→ Used for backend development, WebSocket integration, and Spring Boot configuration.

React Official Documentation

<https://reactjs.org/docs/getting-started.html>

→ Helped build the frontend UI with state management and component lifecycle.

STOMP Protocol Guide

<https://stomp.github.io/stomp-specification-1.2.html>

→ Used to implement the message protocol over WebSocket for reliable communication.

SockJS Client Documentation

<https://github.com/sockjs/sockjs-client>

→ Used for enabling fallback transport methods in case of WebSocket failure.

MongoDB Atlas Documentation

<https://www.mongodb.com/docs/atlas/>

→ For setting up cloud-hosted ephemeral message storage with TTL indexing.

Java WebSocket Programming with Spring

Baeldung Tutorials

<https://www.baeldung.com/websockets-spring>

→ Used to understand WebSocket lifecycle events and messaging structure.

Node.js Official Documentation

<https://nodejs.org/en/docs>

→ Referenced during frontend setup and npm package configuration.

Maven Documentation

<https://maven.apache.org/guides/>

→ Used to understand Maven build lifecycle and dependency management.

Create React App Guide

<https://create-react-app.dev/docs/getting-started/>

→ Helped scaffold the frontend application and configure build scripts.

Real-Time Chat App Tutorials (FreeCodeCamp / GeeksforGeeks / YouTube)

<https://www.geeksforgeeks.org/real-time-chat-application-using-socket-io/>

