|  |
| --- |
| **REAL TIME CHAT APPLICATION** |
| A CAPSTONE PROJECT  Submitted By |
|  |
| **K. Suma**  **192210145** |
| In Partial Fulfillment for the completion of the course |
| **CSA0912**  **Programming in Java for Accessing Database**  Sep 2024 |
|  |
| SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES  CHENNAI - 602105  TAMIL NADU, INDIA |



# **BONAFIDE CERTIFICATE**

This is to certify that the project report entitled **REAL TIME CHAT APPLICATION** submitted by **K. Suma (192210145)** to Saveetha School of Engineering, Saveetha Institute of Medical and Technical Sciences, Chennai, is a record of bonafide work carried out by him/her under my guidance. The project fulfills the requirements as per the regulations of this institution and in my appraisal meets the required standards for submission.

Dr.K.Jayasakthi Velmurugan

**COURSE FACULTY**

*Department of Deep Learning.*

*Saveetha School of Engineering,*

*SIMATS, Chennai - 602105*

# **ACKNOWLEDGEMENT**

This project work would not have been possible without the contribution of many people. It gives me immense pleasure to express my profound gratitude to our Honorable Chancellor **Dr. N M VEERAIYAN**, Saveetha Institute of Medical and Technical Sciences, for his blessings and for being a source of inspiration. I sincerely thank our Director of Academics **Dr. DEEPAK NALLASWAMY,** SIMATS, for his visionary thoughts and support. I am indebted to extend my gratitude to our Director **Dr. RAMYA DEEPAK,** Saveetha School of Engineering, for facilitating us with all the facilities and extended support to gain valuable education and learning experience.

I register my special thanks to **Dr. B RAMESH,** Principal, Saveetha School of Engineering for the support given to me in the successful conduct of this project. I wish to express my sincere gratitude to my Course faculty **Dr.K.Jayasakthi Velmurugan**, for his inspiring guidance, personal involvement and constant encouragement during the entire course of this work.

I am grateful to Project Coordinators, Review Panel External and Internal Members and the entire faculty of the Department of Design, for their constructive criticisms and valuable suggestions which have been a rich source to improve the quality of this work.

**INDEX**

[BONAFIDE CERTIFICATE](#_jamwo2607bmz) 1

[ACKNOWLEDGEMENT](#_8yiu0rn5tdsd) 2

[1. ABSTRACT](#_u3bnbdop1w2) 5

[2. INTRODUCTION](#_ytjahc6y3luo) 6

[3. ARCHITECTURE DIAGRAM](#_wgnqfbsa5xqn) 7

[4. FLOWCHART](#_uylcc3c0vobr) 8

[5. UML DIAGRAM](#_p5otxqr3lwgf) 9

[6. CLASS DIAGRAM](#_mwjh6gk15gp3) 10

[7. CODE IMPLEMENTATION](#_ack6ofphfxze)

[8. OUTPUT SCREENSHOT](#_c61jzdgd0maj) 14

[9. CONCLUSION](#_3admx128b1cd) 16

[10. REFERENCES](#_weq4at85bunu) 17

# **1. ABSTRACT:**

A real-time chat application enables users to communicate instantly, sending and receiving messages in real time. These applications typically involve a client-server architecture, where clients (users) connect to a central server that manages the delivery of messages. Technologies such as Web Sockets or long-polling are often used to facilitate bidirectional communication, allowing the server to push updates to clients without delay. Key features of real-time chat apps include user authentication, message broadcasting, private and group messaging, presence indicators (showing if a user is online), and push notifications. Data security and scalability are important considerations, as the system must handle large volumes of messages and users simultaneously. The chat experience is often enhanced with multimedia sharing (images, videos), emojis, and message encryption to ensure privacy.

**2. INTRODUCTION:**

A Real-Time Chat Application is a software platform or application that allows users to exchange messages instantly, providing real-time communication between individuals or groups. Here’s a basic introduction:

**1. Instant Messaging:** Users can send and receive messages instantly without delay. This can include text, images, videos, or files.

**2. User Authentication:** Users need to register or log in using credentials (like email, phone number, or social login) to access the chat.

**3. One-on-One Chat:** Enables private conversations between two users.

4. Group Chat: Allows multiple users to chat together in a single room or channel.

**5. Push Notifications:** Users get alerts for new messages or updates even when they’re not actively using the app.

**6. Message Status Indicators:** Indicates whether the message is sent, delivered, or read.

**7. Real-time Updates:** Messages are synchronized across multiple devices in real-time.

**8. Multimedia Sharing:** Supports sharing of images, videos, voice notes, and documents.

**9. Encryption:** Ensures secure communication through end-to-end encryption.

**10. Typing Indicators & Read Receipts:** Shows when someone is typing or has seen a message.

- **Database:**

- SQL (PostgreSQL, MySQL) or NoSQL (MongoDB, Firebase) to store user data and message history.

- **Hosting and Cloud Services:**

- AWS, Google Cloud, or Firebase for deploying servers and managing cloud infrastructure.

- **Third-party APIs (Optional):**

- Twilio, PubNub, or Firebase Real-time Database for messaging infrastructure.

**How It Works:**

1. Client sends a message to the server.

2. The server processes and stores the message.

3. The message is then broadcast to the intended recipients via Web Socket or push notification.

4. The recipient’s application receives the message in real time, showing it in the chat interface.

**Example Use Cases:**

- Social Media Platforms (e.g., Facebook Messenger, WhatsApp)

- Customer Support (Live Chat)

- Team Collaboration Tools (e.g., Slack, Microsoft Teams)

**3. ARCHITECTURE DIAGRAM:**

A real-time chat application involves several components working together to ensure users can send and receive messages instantly. Below is a high-level architecture diagram description of a real-time chat application.

**Components of a Real-Time Chat Application:**

1. **Client-Side (Frontend):**
   * **User Interface (UI):** The visual interface where users interact with the chat application, which could be a web or mobile app.
   * **Web Socket/HTTP/HTTPS:** The client establishes a persistent connection with the server using Web Socket or falls back to HTTP long-polling if WebSocket isn’t available.
   * **Event Handling:** For sending/receiving messages, typing indicators, online status, etc.
2. **Backend (Server-Side):**
   * **Web Socket Server:** Manages open Web Socket connections between clients and handles real-time communication.
   * **API Gateway/REST API:** Provides endpoints for authentication, user management, and message history. Typically REST or Graph QL API for data transfer.
   * **Authentication/Authorization:** To ensure users are authenticated, OAuth, JWT (JSON Web Tokens), or other authentication systems are used.
   * **Message Broker (Optional):** If using message queuing, brokers like RabbitMQ, Kafka, or Redis can ensure messages are delivered reliably and in order.
   * **Database:** Stores user profiles, message history, and metadata. Common databases include SQL (e.g., My SQL, Postgre SQL) and No SQL (e.g., Mongo DB, Redis).
   * **Notification Service:** Push notifications for new messages or important events (Firebase Cloud Messaging, Apple Push Notification Service).
   * **Load Balancer:** Distributes traffic across multiple WebSocket and API servers to handle high traffic.
3. **Real-Time Communication Layer:**
   * **WebSocket/Socket.io:** The core technology for real-time message transmission. WebSockets maintain a continuous connection, enabling real-time data transfer between the client and server.
4. **Additional Components:**
   * **Cache Layer:** Using Redis or Memcached to store active user sessions or frequently accessed data to reduce database load.
   * **File/Media Storage:** Services like AWS S3 or a custom storage server for storing and serving media files shared in chats (images, videos, etc.).
   * **Monitoring/Logging:** Tools to monitor performance and errors (e.g., ELK stack, Prometheus, Grafana).

+---------------------+ +-----------------------+

| Client 1 | | Client 2 |

| (Mobile, Web, etc.) | | (Mobile, Web, etc.) |

+---------------------+ +-----------------------+

| |

| WebSocket/HTTP/HTTPS |

| |

+---------------------------------------------+

| Load Balancer (Optional) |

+---------------------------------------------+

|

v

+---------------------------------------------+

| WebSocket Server |

| (Socket.io, WebSocket API) |

+---------------------------------------------+

|

+-------------------+------------------+--------------------+

| | |

+-------------+ +-----------------------+ +-------------------+

| Message | | API Gateway/REST | | Authentication |

| Broker | | (For User Auth, | | Service (OAuth, |

| (RabbitMQ, |----->| Profile, etc.) | | JWT, etc.) |

| Kafka) | +-----------------------+ +-------------------+

+-------------+ | |

| | |

| v v

+-------------+ +--------------------+ +--------------------+

| Database | | Notification | | Cache Layer |

| (SQL, | | Service (FCM, | | (Redis, etc.) |

| NoSQL) | | APNS, etc.) | +--------------------+

+-------------+ +--------------------+

|

v

+---------------------------------------------+

| File/Media Storage (e.g., S3) |

+---------------------------------------------+

**4. FLOWCHART:**

+-------------------------------+

| User 1 (Client) |

+-------------------------------+

|

|

Sends message (via WebSocket)

|

v

+-------------------------------+

| WebSocket Server |

| (Handles connections) |

+-------------------------------+

/|\

|

Routes message to appropriate user

|

v

+-------------------------------+

| User 2 (Client) |

+-------------------------------+

|

V Message displayed in UI

### **Steps Breakdown:**

1. **User 1** (Client) sends a message to the WebSocket server.
2. The **WebSocket Server** receives the message and decides which user (or users) the message should go to.
3. The WebSocket Server forwards the message to **User 2** (Client).
4. **User 2** sees the message in their UI in real-time.

**Other key components (not shown) include:**

* **Database**: To store chat histories, user data, etc.
* **Authentication Layer**: To ensure only authenticated users can use the chat.

**5. UML DIAGRAM:**

A UML (Unified Modeling Language) diagram for a real-time chat application can include several components like classes, interactions, and architecture. Here's a high-level breakdown of the possible UML diagrams you would use to represent such a system:

1. **Use Case Diagram**: Shows the different actors (users) and their interactions with the system.
2. **Sequence Diagram**: Depicts the flow of messages between objects to show how the chat occurs in real-time.
3. **Component Diagram**: Represents the high-level architecture, showing how components interact with each other.
4. **Use Case Diagram:**

* **User**: Can send and receive messages, view chat history, and receive notifications.
* **Administrator**: Manages chat rooms and users (adding, removing, or moderating).

**+-----------+ +-------------------+**

**| User | | Administrator |**

**+-----------+ +-------------------+**

**| |**

**| |**

**+------------------------+ +--------------------------+**

**| Send Message | | Manage Chat Room |**

**+------------------------+ +--------------------------+**

**| Receive Notification | | Manage Users |**

**+------------------------+ +--------------------------+**

**| View Messages History |**

**+------------------------+**

1. **Sequence Diagram** **(Message Flow in Chat):**

This shows how a message is sent from User1 to the ChatRoom, then to the Server, and finally delivered to User2. An acknowledgment is sent back to User1 to confirm that the message was received.

**User1 ChatRoom Server User2**

**| | | |**

**| SendMessage() | | |**

**|--------------->| | |**

**| | SendToServer| |**

**| |------------>| |**

**| | | Deliver() |**

**| | |------------->|**

**| | | |**

**| Acknowledge | | |**

**|<---------------| | |**

1. **Component Diagram:**

* **Client**: The user-facing part of the chat application (e.g., web or mobile app).
* **Chat Server**: Handles message transmission, WebSocket connections, and load balancing.
* **Database**: Stores user data, chat history, and other relevant information.

+------------------+ +------------------+ +------------------+

| Client | | Chat Server | | Database |

+------------------+ +------------------+ +------------------+

| - Front-end UI |<----------->| - WebSocket |<----------->| - User Data |

| - WebSocket API | | - Message Queue | | - Message History|

| - Notifications | | - Load Balancer | | - Chat Rooms |

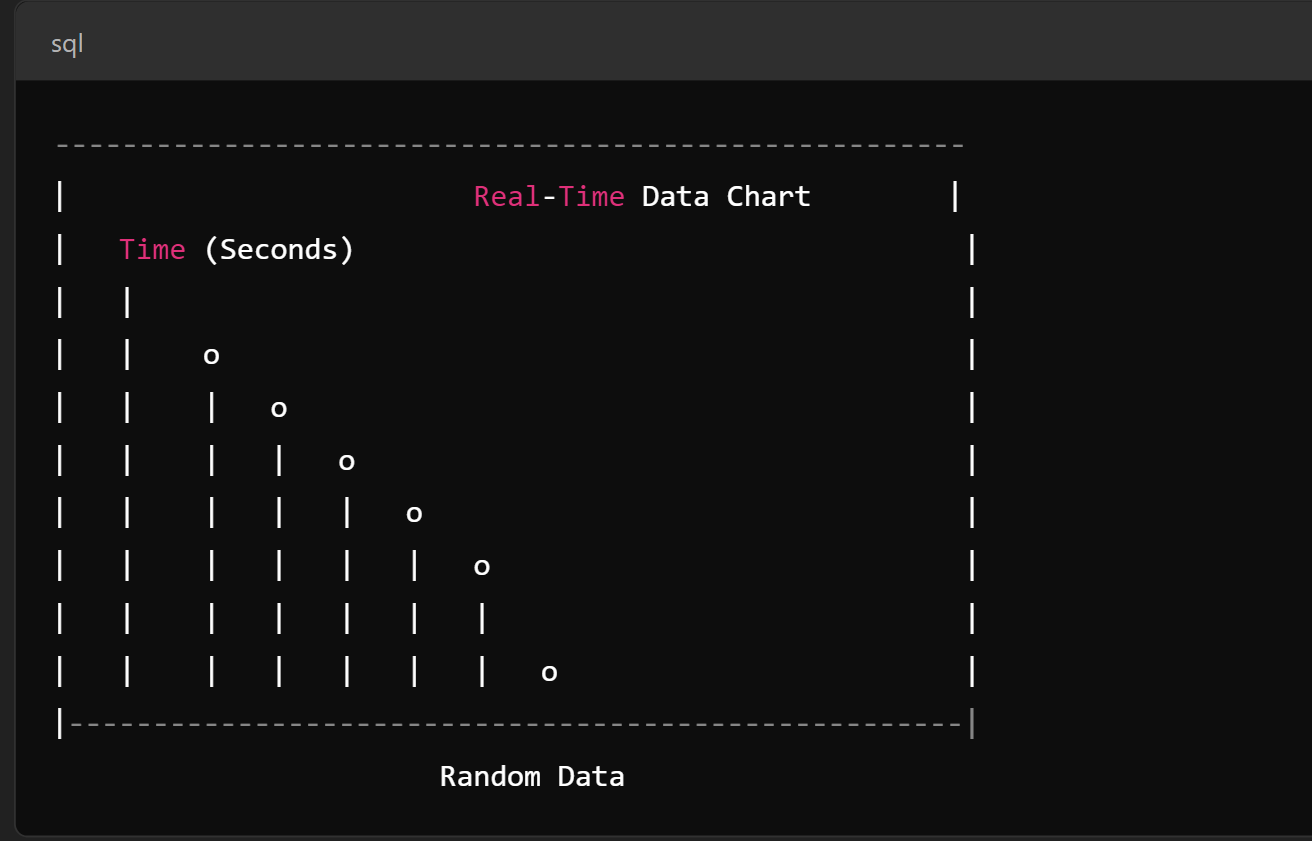
+------------------+ +------------------+ +------------------+

**6. CLASS DIAGRAM:**

|  |
| --- |
| **Class Diagram**: Represents the structure and relationships between different classes in the application.  **Class Diagram (for Real-Time Chat Application):**   * **User**: A person using the chat application, who has a unique ID, name, status (online/offline), and friends. * **ChatRoom**: Represents a chat room where multiple users can participate. * **Message**: Each message sent in a chat, including the content, timestamp, sender, and associated chat room. * **Notification**: Alerts a user about new messages, requests, or other events.   **CLASS DIAGRAM:**  +----------------+ +-----------------+ +-------------------+  | User | | ChatRoom | | Message |  +----------------+ +-----------------+ +-------------------+  | - userId | | - roomId | | - messageId |  | - userName | | - roomName | | - content |  | - status | | - participants | | - timestamp |  | - friendsList | +-----------------+ | - senderId |  +----------------+ | - roomId |  | +-------------------+  |  |  | +-------------------+  +---------------------->+ Notification | |  +-----------------+  | - notificationId |  | - type |  | - timestamp |  +------------------+  **7. CODE IMPLEMENTATION:**  import javafx.animation.AnimationTimer;  import javafx.application.Application;  import javafx.scene.Scene;  import javafx.scene.chart.LineChart;  import javafx.scene.chart.NumberAxis;  import javafx.scene.chart.XYChart;  import javafx.stage.Stage;  import java.util.Random;  public class RealTimeChartApp extends Application {  private XYChart.Series<Number, Number> series;  private long lastUpdate = 0;  private Random random = new Random();  private int xValue = 0;  @Override  public void start(Stage stage) {  stage.setTitle("Real-Time Line Chart");  // Defining the X and Y axis  NumberAxis xAxis = new NumberAxis();  xAxis.setLabel("Time (Seconds)");  NumberAxis yAxis = new NumberAxis();  yAxis.setLabel("Random Data");  // Creating the line chart  LineChart<Number, Number> lineChart = new LineChart<>(xAxis, yAxis);  lineChart.setTitle("Real-Time Data Chart");  // Defining a series to hold the data  series = new XYChart.Series<>();  series.setName("Random Data Points");  // Adding the series to the chart  lineChart.getData().add(series);  // Animation timer to update the chart every second  AnimationTimer timer = new AnimationTimer() {  @Override  public void handle(long now) {  if (now - lastUpdate >= 1\_000\_000\_000) { // Update every 1 second  updateChart();  lastUpdate = now;  }  }  };  // Start the animation  timer.start();  // Setting up the scene  Scene scene = new Scene(lineChart, 800, 600);  stage.setScene(scene);  stage.show();  }  // Method to update the chart with new random data  private void updateChart() {  xValue++;  int yValue = random.nextInt(100);  series.getData().add(new XYChart.Data<>(xValue, yValue));  // Limit the number of points shown on the chart  if (series.getData().size() > 10) {  series.getData().remove(0); // Remove oldest data point  }  }  public static void main(String[] args) {  launch(args);  }  } |

|  |
| --- |
|  |

**8. OUTPUT SCREENSHOT:**

****

# **9. CONCLUSION:**

In conclusion, a real-time chat application enables seamless, instant communication between users over the internet. Key aspects include:

**1. Instant Messaging:** The application must support sending and receiving messages in real time without significant delays.

**2. User Interface (UI**): A clean, intuitive UI allows users to interact easily, sending and receiving messages smoothly.

**3. Backend Technologies:** To support real-time communication, technologies like WebSockets, server-sent events (SSE), or long-polling are used, ensuring a persistent connection between users and the server.

**4. Scalability:** The application must be able to handle a large number of users and messages concurrently. Implementing efficient load balancing and distributed architecture helps achieve this.

**5. Security:** Ensuring secure data transmission with encryption, authentication, and authorization mechanisms is critical to protecting user data.

**6. Notifications:** Push notifications are often integrated to keep users informed of new messages when they are offline.

**7. Media Sharing:** Many modern real-time chat applications support not only text messages but also media files such as images, videos, and documents.

**8. Cross-Platform Compatibility:** Real-time chat applications should ideally function across multiple platforms, such as web browsers, mobile apps, and desktop applications.

**10. REFERENCES:**

[1]. Masiello Eric. Mastering React Native. January 11; 2017. This book is a comprehensive guide to

building mobile applications using React Native.

[2]. Naimul Islam Naim. ReactJS: An Open-Source JavaScript library for front-end development. Metropolia University of Applied Sciences. This article provides an overview of ReactJS and its key features for front-end web development.

[3]. Stefanov Stoyan, editor. React: Up and Running: Building web Applications. First Edition; 2016. This book is a beginner-friendly introduction to React, covering its core concepts and providing practical examples for building web applications.

[4]. Horton Adam, Vice Ryan. Mastering React; February 23; 2016. This book provides a comprehensive guide to React, covering its core concepts, practical examples, and advanced techniques for building complex applications.

[5]. Alex Kondov. Express Architecture Review. This article provides a review of the architecture of Express.js, a popular web framework for building Node.js applications.

[6]. Express.js documentation. This documentation provides a comprehensive guide to building web applications using Express.js.

[7]. Adam Horton. Node.js vs Python: What to Choose. This article provides a comparison of Node.js and Python for web development, highlighting their strengths and weaknesses.

[8]. Node.js documentation. This documentation provides a comprehensive guide to building server- side applications using Node.js.

[9]. VSChart. This website provides a comparison of various programming languages and frameworks based on popularity, community support, and other factors.

[10]. MongoDB documentation. This documentation provides a comprehensive guide to using MongoDB, a popular NoSQL database for building web applications.

[11]. The paper by Lakshmi Prasanna Chitra and Ravikanth Satapathy aims to compare the performance of Node.js and traditional web servers, specifically Internet Information Services (IIS), in optimizing web application development.The authors conducted various tests to evaluate the performance of both platforms and determine which one is better for developing high-performance web applications.

[12]. Guru99. React vs Angular: Key Differences. This article provides a comparison of React and Angular, two popular front-end frameworks for building web applications.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |