REAL-TIME CHAT APPLICATION

*A Project Based Learning Report Submitted in partial fulfilment of the requirements for the award of the degree*

*of*

**Bachelor of Technology**

**in The Department of**

**AOOP WITH 23CS2103E**

Submitted by

**2310030188 : Jashwanth Reddy**

**2310030163 : Abhinav Reddy**

**2310030421 : D. Guru charan**

Under the guidance of

**MR. Sashidhar Sir**

Department of Electronics and Communication Engineering

Koneru Lakshmaiah Education Foundation, Aziz Nagar

Aziz Nagar – 500075 (Optional)

NOV - 2023.

**What is a Real-Time Chat Application?**

A **Real-Time Chat Application** allows users to **send and receive messages instantly**, **without** needing to refresh the page or manually reload anything.

The keyword is **"Real-Time"** — as soon as **one user** sends a message, it **immediately appears** on **everyone else's screen** connected to the chat.

This happens through **live server connections** using technologies like **WebSockets**, **Socket.IO**, **Pusher**, **Firebase**, etc.

**Uses / Purposes of a Real-Time Chat Application**

✅ **Instant Communication**

* Quickly talk with friends, coworkers, customers, etc., without waiting.

✅ **Customer Support Chat**

* Websites use real-time chats to help customers immediately (like "Live Chat" on shopping websites).

✅ **Collaboration Tools**

* Apps like Slack and Microsoft Teams use real-time chat for fast team discussions.

✅ **Gaming**

* Multiplayer games use real-time chat for players to coordinate strategies.

✅ **Social Media**

* Platforms like Facebook Messenger and Instagram DMs enable real-time messaging.

✅ **Live Events / Streams**

* YouTube Live and Twitch chats provide real-time communication during events.

✅ **Online Learning**

* Students and teachers interact instantly during online classes.

✅ **Telemedicine**

* Doctors and patients can communicate immediately through chat.

**Why is Real-Time Important?**

* **Speed:** No waiting for page reloads.
* **Engagement:** Keeps users active and interested.
* **Collaboration:** Teams and communities can work or discuss faster.
* **Better User Experience:** Feels modern, smooth, and alive.

**Real-World Examples of Real-Time Chat Applications**

| **Application** | **Purpose** |
| --- | --- |
| WhatsApp | Personal communication |
| Slack | Team collaboration |
| Discord | Communities + gaming |
| Facebook Messenger | Social networking |
| Live Chat on Shopify | Customer support |

**Frontend**

**Frontend** is the part of a website, web application, or mobile app that the **user directly interacts with**.

It’s **everything you see** on the screen:

* The **buttons** you click
* The **forms** you fill out
* The **chat messages** you read
* The **animations**, **colors**, **fonts**, and **layouts**

Basically, **frontend** is about **design** + **functionality** that happens **on the user's device** (browser, phone, etc.).

The **frontend** is what users interact with. It handles:

* Displaying chat messages instantly
* Sending user messages
* Showing online users
* Notifications for new messages
* **Visual Layout**: How the page looks — placement of text, images, buttons.
* **User Interaction**: What happens when you click a button? Scroll a page? Type a message?
* **Responsiveness**: How the app adjusts when you use it on different devices (phone, tablet, desktop).
* **Animations/Effects**: Smooth transitions, loading animations, popups, modals.

**Frontend Components Example**:

* Login/Register Screen
* Chat Room UI (messages + input box)
* User List Sidebar (showing who’s online)
* Notifications for new messages

**Technologies Used in Frontend**

| **Technology** | **Purpose** | **Examples** |
| --- | --- | --- |
| **HTML** | Structure of the webpage | Headings, paragraphs, links |
| **CSS** | Styling (colors, layouts, fonts) | Buttons, backgrounds |
| **JavaScript** | Making things interactive | Sliders, forms, alerts |
| **Frontend Frameworks** | Build complex apps faster | React.js, Vue.js, Angular |
| **State Management** | Manage app data across pages/screens | Redux, Context API |
| **WebSockets** | Real-time updates (for chat apps) | Instant messaging |

**Code :**

package com.chatapp.gui;

import client.ChatClient; // Ensure this import matches your ChatClient package

import com.chatapp.db.Message; // If you use Message objects

import com.chatapp.db.MessageDAO; // If you use MessageDAO

import javafx.application.Application;

import javafx.application.Platform;

import javafx.collections.FXCollections;

import javafx.collections.ObservableList;

import javafx.geometry.Insets;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.layout.\*;

import javafx.stage.Stage;

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

import java.util.Arrays; // Import Arrays

import java.util.List;

import java.util.stream.Collectors;

public class ChatAppUI extends Application {

// --- UI Components ---

private TextArea chatArea;

private TextField inputField;

private Button sendButton;

private ListView<String> userListView;

private RadioButton privateRadioButton;

private RadioButton groupRadioButton;

private ToggleGroup messageTypeGroup;

private BorderPane root; // Main layout pane

// --- State & Networking ---

private String currentUsername; // Store the logged-in username

private ChatClient chatClient;

private MessageDAO messageDAO; // Optional: For loading/saving message history

// Observable list to hold user names for the ListView

private final ObservableList<String> userList = FXCollections.observableArrayList();

// --- Constructors ---

/\*\*

\* Primary constructor used by LoginRegisterApp.

\* @param username The username of the logged-in user.

\*/

public ChatAppUI(String username) {

if (username == null || username.trim().isEmpty()) {

throw new IllegalArgumentException("Username cannot be null or empty when creating ChatAppUI.");

}

this.currentUsername = username;

this.messageDAO = new MessageDAO(); // Initialize DAO (if needed)

System.out.println("ChatAppUI initialized for user: " + this.currentUsername);

}

/\*\*

\* Default constructor (Needed for some JavaFX scenarios, but not the main entry point).

\*/

public ChatAppUI() {

System.err.println("Warning: ChatAppUI created using default constructor. User context might be missing.");

}

// --- JavaFX Application Start Method ---

@Override

public void start(Stage stage) {

System.out.println("ChatAppUI start() method called.");

// 1. Check if user is logged in (essential)

if (this.currentUsername == null || this.currentUsername.trim().isEmpty()) {

System.err.println("ChatAppUI cannot start: Username not provided.");

showError("Fatal Error: User not logged in. Please restart via Login Screen.");

Platform.runLater(stage::close);

return;

}

stage.setTitle("ChatApp - Logged in as: " + this.currentUsername);

// 2. Initialize UI Components

initializeUIComponents();

// 3. Setup Layout

setupLayout();

// 4. Connect to Server & Start Listening (CRITICAL STEP)

if (!connectToServer()) {

Platform.runLater(stage::close);

return;

}

// 5. Load Initial Data (Optional History/Users)

loadInitialChatHistory();

// 6. Create Scene and Load CSS

Scene scene = new Scene(root, 750, 550); // Create scene BEFORE loading CSS

// --- START: Debugging CSS Loading ---

try {

System.out.println("--- CSS Loading Debug ---");

// Test loading login-styles.css (which worked before)

java.net.URL loginCssUrlRelative = getClass().getResource("login-styles.css");

System.out.println("Relative path ('login-styles.css'): " + loginCssUrlRelative);

java.net.URL loginCssUrlAbsolute = getClass().getClassLoader().getResource("com/chatapp/gui/login-styles.css");

System.out.println("Absolute path ('login-styles.css'): " + loginCssUrlAbsolute);

// Test loading

**Backend**

The **Backend** is the **behind-the-scenes brain** of any website or app.  
 It’s what happens **on the server**, not the screen.

If the **Frontend** is what the **user sees**,  
 then the **Backend** is what the **system does**.

It handles:

* Logic
* Databases
* Authentication
* Security
* Real-time events
* Communication with the frontend

Users don’t see it, but it powers everything.

The **backend** handles:

* Authenticating users
* Handling WebSocket connections
* Broadcasting messages
* Storing messages in a database
* Fetching old messages

**Technologies commonly used**:

| **Tool / Language** | **Role** | **Examples** |
| --- | --- | --- |
| **Node.js** | JavaScript runtime on server | Express.js, Socket.IO |
| **Python** | Backend language | Django, Flask |
| **PHP** | Server-side scripting | Laravel, WordPress |
| **Java** | Enterprise-level backend | Spring Boot |
| **Databases** | Stores app data | MongoDB, MySQL, PostgreSQL |
| **APIs (REST/GraphQL)** | Interface to frontend | /api/users, /messages |
| **WebSockets** | Real-time connection | For live chat, notifications |
| **Authentication** | Security features | JWT, OAuth2, bcrypt |

**Code :**

t thread pool terminated gracefully."); }

} catch (InterruptedException ie) { System.err.println("SERVER: Shutdown interrupted."); clientExecutor.shutdownNow(); Thread.currentThread().interrupt(); }

System.out.println(" SERVER: Shutdown complete.");

}

// --- Client Management ---

protected void addClientHandler(ClientHandler handler) {

if (handler != null && handler.getUsername() != null) { if (clientHandlers.add(handler)) { System.out.println("SERVER: Client handler added for: " + handler.getUsername() + ". Total clients: " + clientHandlers.size()); broadcastUserListUpdate(); } }

else { System.err.println("SERVER ERROR: Attempted to add invalid handler."); }

}

protected void removeClientHandler(ClientHandler handler) {

if (handler != null) { if (clientHandlers.remove(handler)) { String username = handler.getUsername(); System.out.println("SERVER: Client handler removed for: " + (username != null ? username : "[unknown]") + ". Total clients: " + clientHandlers.size()); broadcastUserListUpdate(); } }

}

protected synchronized boolean isUsernameTaken(String username) {

if (username == null || username.trim().isEmpty()) return true;

String trimmedUsername = username.trim();

return clientHandlers.stream().anyMatch(h -> trimmedUsername.equalsIgnoreCase(h.getUsername()));

}

protected String getUsernamesString() { return clientHandlers.stream().map(ClientHandler::getUsername).filter(Objects::nonNull).sorted(String.CASE\_INSENSITIVE\_ORDER).collect(Collectors.joining(",")); }

// --- Messaging ---

public void broadcast(String message, ClientHandler sender) {

String senderName = (sender != null && sender.getUsername() != null) ? sender.getUsername() : "[SERVER]";

System.out.println("SERVER: Entering broadcast method. Sender: " + senderName + ". Message: " + message); System.out.println("SERVER: Broadcasting to " + clientHandlers.size() + " total handlers (excluding sender if applicable).");

int count = 0;

for (ClientHandler client : clientHandlers) { if (client != sender) { String recipientName = (client != null && client.getUsername() != null) ? client.getUsername() : "[unknown]"; System.out.println("SERVER: Broadcasting message to handler: " + recipientName); client.sendMessage(message); count++; } }

System.out.println("SERVER: Finished broadcast loop. Sent to " + count + " recipient(s) for message from " + senderName);

}

public void sendPrivateMessage(String message, String recipientUsername, ClientHandler sender) {

String senderName = (sender != null && sender.getUsername() != null) ? sender.getUsername() : "[SERVER]";

System.out.println("SERVER: Entering sendPrivateMessage method. From " + senderName + " to '" + recipientUsername + "'");

Optional<ClientHandler> recipient = clientHandlers.stream().filter(client -> recipientUsername.equalsIgnoreCase(client.getUsername())).findFirst();

if (recipient.isPresent()) { ClientHandler recipientHandler = recipient.get(); System.out.println("SERVER: Found recipient handler for PM: " + recipientHandler.getUsername() + ". Sending message: " + message); recipientHandler.sendMessage(message); }

else { System.out.println("SERVER: Recipient '" + recipientUsername + "' not found for private message from " + senderName); if (sender != null) { sender.sendMessage("ERROR:Server:User '" + recipientUsername + "' is not online or does not exist."); } }

}

public void broadcastUserListUpdate() { String userListString = getUsernamesString(); String userListMessage = "UPDATE\_USERS:" + userListString + ":"; System.out.println("SERVER: Broadcasting user list update: " + userListMessage); clientHandlers.forEach(client -> client.sendMessage(userListMessage)); }

// --- FIX: Added public getter for the running state ---

/\*\*

\* Checks if the server is currently marked as running.

\* @return true if the server is running, false otherwise.

\*/

public boolean isRunning() {

return this.running; // Safely return the value of the private volatile field

}

// --- End Fix ---

// --- Main ---

public static void main(String[] args) {

int port = 1237; // Default port from your log

if (args.length > 0) { /\* ... port parsing ... \*/ }

ChatServer server = new ChatServer(port);

Runtime.getRuntime().addShutdownHook(new Thread(() -> { System.out.println("SERVER: Shutdown hook triggered."); server.shutdown(); }, "ServerShutdownHook"));

server.start();

}

} // End of ChatServer class

# **Database**

The **database** stores:

* User profiles (username, email, password hash)
* Chat messages (message text, sender, timestamp, room)

**Common choices**:

* MongoDB (document-based, flexible)
* PostgreSQL, MySQL (relational databases)
* Redis (for fast real-time data like sessions)

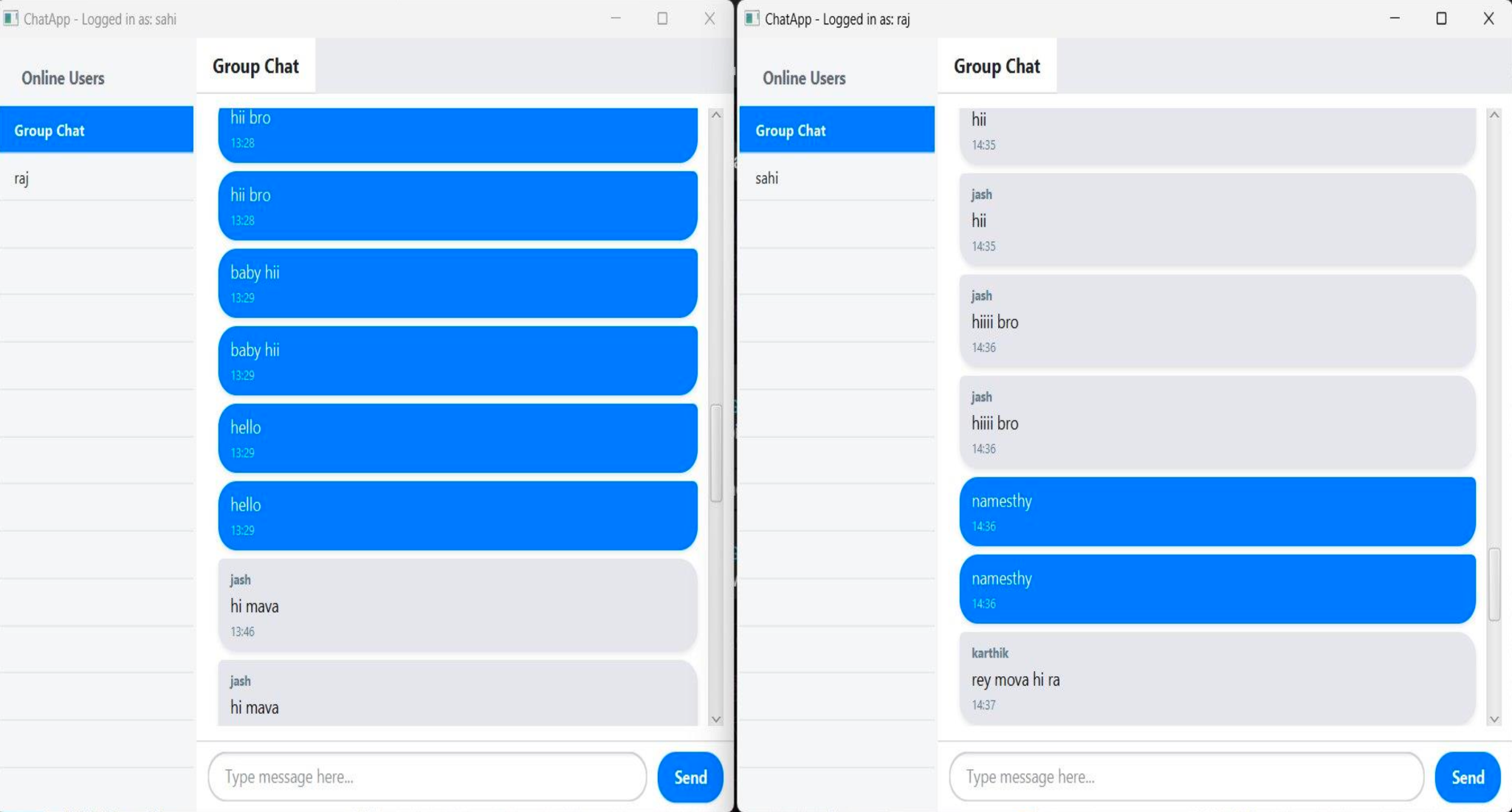
**MongoDB Schema Example**:

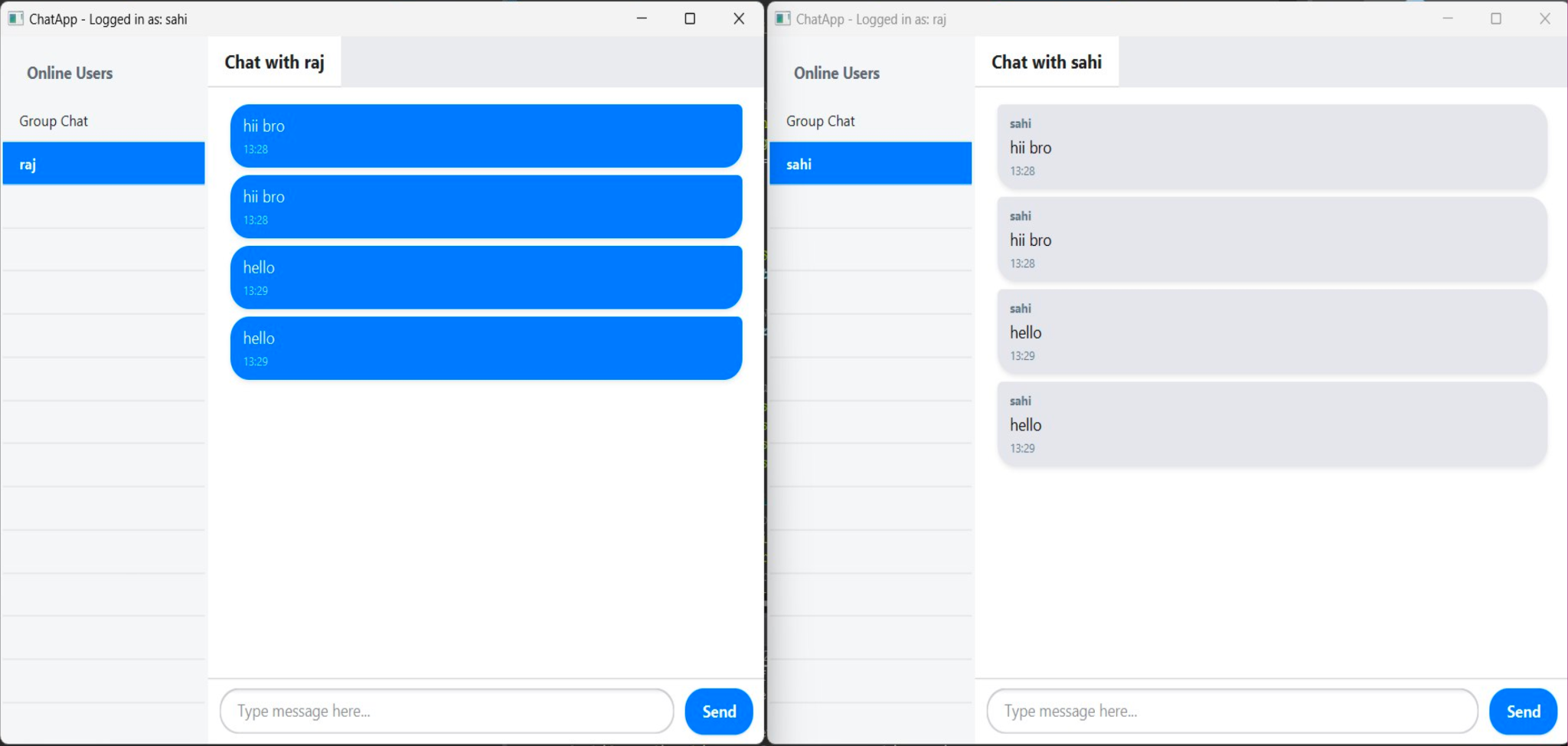
# **Full Example Flow**

**Scenario**: User A sends a message to a chat room.

1. **Frontend**:  
   * User A types "Hello everyone!" and clicks **Send**.
   * The frontend triggers a socket.emit('chat message', messageData).
2. **Backend**:  
   * Server listens with socket.on('chat message').
   * Server saves the message to MongoDB.
   * Server broadcasts the message to all connected clients.
3. **Frontend** (on all clients):  
   * All clients receive the event and update their chat screens instantly.
4. **Database**:  
   * Messages are stored securely so no data is lost.
   * New users can fetch old messages when they join.

**This how the login page looks and works:**

****

****

# **Code:**

package com.chatapp.gui;

import javafx.application.Application;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.stage.Stage;

import javafx.scene.control.Alert; // Import Alert

import java.io.IOException;

import java.net.URL;

/\*\*

\* Main application class. Sets up the initial stage and loads the

\* login/register screen from FXML.

\*/

public class LoginRegisterApp extends Application {

@Override

public void start(Stage primaryStage) {

try {

// --- Load the FXML file ---

// Define the path relative to the current class's package

String fxmlPath = "login-register.fxml";

URL fxmlUrl = getClass().getResource(fxmlPath);

// Check if the FXML file was found

if (fxmlUrl == null) {

System.err.println("CRITICAL ERROR: Cannot find FXML file at path: " + fxmlPath);

System.err.println("Ensure '" + fxmlPath + "' is in the same package as LoginRegisterApp.java " +

"or check your build configuration (e.g., Maven/Gradle resource handling).");

// Fallback attempt using ClassLoader (requires full path from classpath root)

String absolutePath = "com/chatapp/gui/" + fxmlPath;

System.err.println("Attempting fallback load via ClassLoader: " + absolutePath);

fxmlUrl = getClass().getClassLoader().getResource(absolutePath);

if (fxmlUrl == null) {

// Throw an exception if still not found

throw new IOException("FXML file '" + fxmlPath + "' not found via getResource() or ClassLoader().");

} else {

System.out.println("Loaded FXML via ClassLoader path successfully.");

}

} else {

System.out.println("Loading FXML using getResource() from: " + fxmlUrl);

}

// Load the FXML hierarchy

FXMLLoader loader = new FXMLLoader(fxmlUrl);

Parent root = loader.load(); // Loads the VBox defined in FXML

// --- Create the Scene ---

// The size might be determined by the FXML's prefWidth/prefHeight,

// or you can set it explicitly here if needed.

Scene scene = new Scene(root);

// --- Load the CSS ---

String cssPath = "login-styles.css";

URL cssUrl = getClass().getResource(cssPath); // Try relative path first

if (cssUrl == null) {

cssUrl = getClass().getClassLoader().getResource("com/chatapp/gui/" + cssPath); // Fallback

}

if (cssUrl != null) {

scene.getStylesheets().add(cssUrl.toExternalForm());

System.out.println("Login CSS loaded successfully from: " + cssUrl);

} else {

// Log a warning if CSS is missing, but don't stop the app

System.err.println("Warning: Could not load CSS file 'login-styles.css'. UI might lack styling.");

}

// --- Configure and Show the Stage ---

primaryStage.setTitle("ChatApp - Login / Register");

primaryStage.setScene(scene);

primaryStage.setResizable(false); // Login window is typically not resizable

primaryStage.show();

} catch (IOException e) {

// Handle errors loading FXML (critical)

System.err.println("Failed to load application UI (FXML): " + e.getMessage());

e.printStackTrace();

// Show a user-friendly error dialog

showErrorDialog("Application Load Error",

"Failed to load the application interface.",

"Could not load the main screen (login-register.fxml).\n" +

"Please check the application files and installation.\nError: " + e.getMessage());

} catch (Exception e) {

// Catch any other unexpected errors during startup

**The Successful JavaFX console output from Eclipse IDE, showing the startup logs for a Chat Application project.**

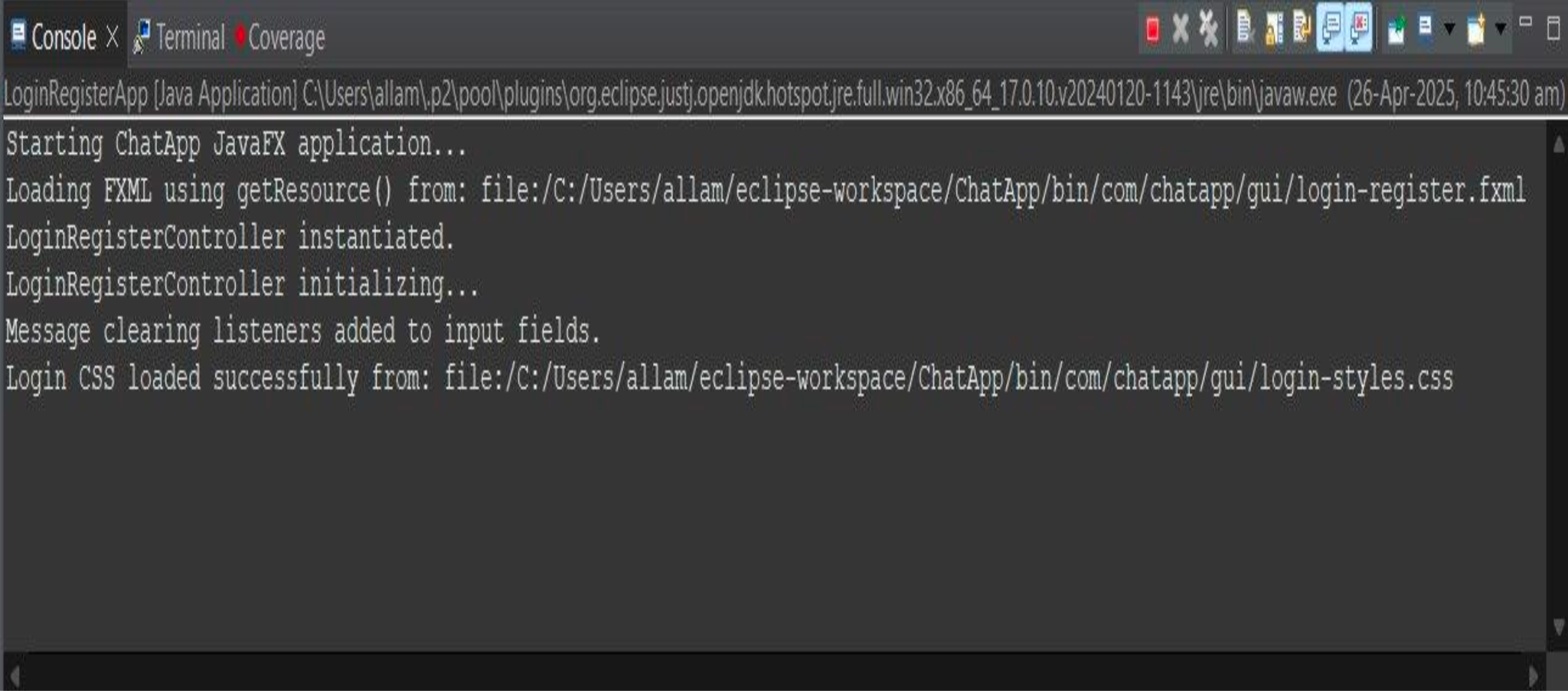
**steps:**

You are starting a JavaFX Chat Application.

It is loading a Login/Register screen designed in FXML.

The logic behind that screen is handled by LoginRegisterController.

The screen's styling (colors, fonts, etc.) is loaded through a CSS file.

****

**REFERENCES**

**React Documentation – *React – A JavaScript library for building user interfaces*. https://reactjs.org**

**Node.js Documentation – *Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine*.**

**https://nodejs.org**

**Express.js Guide – *Fast, unopinionated, minimalist web framework for Node.js*. https://expressjs.com**

**MongoDB Documentation – *The Developer Data Platform*.**

**https://www.mongodb.com/docs**