**Chat Relay – Design Document**

Software Design Specification

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Revision** | **Description** | **Author** |
| 3/16/2025 | 1.0 | Created Document | Talhah Shaik |
| 3/30/2025 | 1.1 | Add additional IT Admin Sequence Diagrams | Kenny Kottenstette |
| 3/31/2025 | 1.2 | Add MessageWrapper class, heavily refactor DBManager & Server classes | Kenny Kottenstette |
| 3/31/2025 | 1.3 | Added Sequence Diagrams for Client and GUI | Zoheb Sharif |
| 3/31/2025 | 1.4 | Added Descriptions for Users and IT Admins | Talhah Shaik |
| 4/2/2025 | 1.5 | Added information to Section 1 | Talhah Shaik |
| 4/3/2025 | 1.6 | Added GUI Class Description | Zoheb Sharif |
| 4/3/2025 | 1.7 | Added Class descriptions for Chat and Message | Talhah Shaik |
| 4/6/2025 | 1.8 | Added Prototype description | Talhah Shaik |
| 4/6/2025 | 1.9 | LucidChart UML Class Updates: Renamed MessageWrapper class to Packet. Created new and separate class ClientHandler. Heavily refactored Packet, Server, DBManager. Wrote out the descriptions for all aforementioned classes. | Kenny Kottenstette |
| 4/6/2025 | 2.0 | Updated ‘Client’ Class Description, updated ‘GUI’ LucidChart UML Class | Zoheb |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

Contents

[1. Overview 4](#_Toc194825492)

[1.1. Product Architecture 4](#_Toc194825493)

[1.2. Product Functionality/Features 4](#_Toc194825494)

[1.3. Constraints 4](#_Toc194825495)

[1.4. Assumptions and Dependencies 4](#_Toc194825496)

[1.5. References 4](#_Toc194825497)

[2. Use Cases Diagram - Descriptions 5](#_Toc194825498)

[3. UML Class Diagram - Description 14](#_Toc194825499)

[4. UML Sequence Diagram 20](#_Toc194825500)

[5. Paper Prototype – Interface & Data 24](#_Toc194825501)

[6. Prototype Description 25](#_Toc194825502)

# Overview

This document outlines the design for the Communications System – Chat Relay

## Product Architecture

* Project was created in Java using advanced operations to create an application where employees of a company can communicate over the internet synchronous and asynchronously.
* Multithreaded Client/Server Design pattern
* Java Swing for GUI on client side
* Locally created database to store information

## Product Functionality/Features

* Difference in GUI depending on what kind of user is logged in
  + IT Admin has access to all server items
  + Users have access to only their conversations
* Each message has a direct timestamp to the second
* Every message is saved using the DB Manger class
* Information sent over the internet is wrapped in the Message Wrapper class

## Constraints

* Restricted to only using Java Swing for GUI elements
* Database is created locally with the use of text files
* User creation is only doable from the IT Admin

## Assumptions and Dependencies

* Java Swing - GUI
* LocalDateTime – Message
* Scanner – DB Manager

## References

* Sequence Diagrams in folder ***/Phase 2 Documents/Sequence Diagrams***
* Class Diagram updated in folder /Phase 2 Documents
* Prototypes in folder /Phase 2 Documents

# Use Cases Diagram - Descriptions

**Use Case: 01**

**Use Case Name:** User Login

**Relevant Requirements:** Users must exist before logging in, users must have valid

credentials that are saved on system

**Primary Actor:** User, IT User

**Pre-Conditions:**

* Server is online
* User’s credentials are in data file
* Client has connection to server
* Client has user list and all their chat message data

**Post-Conditions:**

* The user is now logged into the system

**Basic Flow or Main Scenario:**

1. The user logs into the client
2. The user is prompted with a login screen
3. The user enters their login info
4. The system validates the login info
5. If valid, the user is given access as well as user list and all chat message data

**Extensions or Alternate Flows:**

* Invalid Credentials - If users provide invalid credentials, an error message will pop up and the user will have to provide new login details again.
* Server Outage - If the user is trying to login while the server is down, an error message will pop up saying that the server is down.

**Exceptions:**

* User has connectivity issues

**Related Use Cases:** N/A

**Use Case: 02**

**Use Case Name:** Sending Message

**Relevant Requirements:** User must be logged in, system ensures message delivery

**Primary Actor:** User

**Pre-Conditions:**

* User has successfully logged into the system
* User must have a recipient to send the message to
* System is online

**Post-Conditions:**

* The message has been sent successfully

**Basic Flow or Main Scenario:**

1. After login, user is in messaging interface
2. User selects a recipient or group chat
3. User types a message
4. User sends message
5. System sends message

**Extensions or Alternate Flows:**

Message Content - Messages can only be text, no images or video can be sent

**Exceptions:**

* Recipient does not exist

**Related Use Cases:** N/A

**Use Case: 03**

**Use Case Name:** Create Group Chat

**Relevant Requirements:** User must have logged in successfully, user must select recipients

**Primary Actor:** Users

**Pre-Conditions:**

* User must be successfully logged in

**Post-Conditions:**

* A new group chat is made with the new participants

**Basic Flow or Main Scenario:**

1. User has logged into the system
2. The user creates a group chat
3. The user selects the recipients to add to the group chat
4. The user gives the chatroom a name
5. System creates group chat with all added recipients
6. The user can now send messages in this new group chat

**Extensions or Alternate Flows:**

* None

**Exceptions:**

* To make a group chat, you must have at least two recipients. If two recipients don’t exist for the user to add, an error message is thrown
* If the chatroom’s name is already taken, an error message is thrown

**Related Use Cases:** N/A

**Use Case: 04**

**Use Case Name:** Creating User

**Relevant Requirements:** IT User must be logged in

**Primary Actor:** IT User

**Pre-Conditions:**

* IT User has successfully logged into the system

**Post-Conditions:**

* The new user has been successfully created

**Basic Flow or Main Scenario:**

1. After login, IT user navigates to admin interface
2. Admin enters users first name, last name and username
3. Admin receives a success message of the new user being created

**Extensions or Alternate Flows:**

* If User’s first name and last name are under 3 characters in length, just use the the 1 or 2 characters (Ex: Kenneth Ko → KenKo)
* Update Users - change name or details

**Exceptions:**

* If username already exists, IT User should add a numeric postfix (ex: KenKot2)

**Related Use Cases:** N/A

**Use Case: 05**

**Use Case Name:** Disabling/Re-enabling User

**Relevant Requirements:** IT User must be logged in

**Primary Actor:** IT User

**Pre-Conditions:**

* IT User has successfully logged into the system
* The User to be disabled/re-enabled exists
* The User to be disabled isn’t an IT User themselves

**Post-Conditions:**

* The target user is disabled/re-enabled and their login credentials are invalid for future system access

**Basic Flow or Main Scenario:**

1. After login, IT User navigates to admin interface
2. IT User selects target User and changes their account to disabled/re-enabled
3. Admin receives a success message of the new user being disabled/re-enabled

**Extensions or Alternate Flows:**

* If target User is an IT User, the option shouldn’t be listed in the UI

**Exceptions:**

**Related Use Cases:** N/A

**Use Case: 06**

**Use Case Name:** Receive Message

**Relevant Requirements:** User must be logged in, system must be online, received message has recipient and time stamp

**Primary Actor:** User

**Pre-Conditions:**

* User has logged into the system with verified credentials
* System is online

**Post-Conditions:**

* The message is received by the user
* The message will be recorded in the persistent log

**Basic Flow or Main Scenario:**

1. The user logs in
2. The system constantly checks if a message was sent to the user until the user logs out again
3. When a message is received, it will be shown in the chat log with the user and the recipient

**Extensions or Alternate Flows:**

* If the user is offline before being sent a message: The system sends a message with the recipient's name and the timestamp of the message

**Exceptions:**

* If the server fails to send the message, on which every user has the outgoing message, explain the error

**Related Use Cases:** User login, Sending Message

**Use Case: 07**

**Use Case Name:** Logout

**Relevant Requirements:** User must have successfully logged in with valid credentials

**Primary Actor:** User

**Pre-Conditions:**

* User must have successfully logged in with valid credentials

**Post-Conditions:**

* Client closes
* The user is logged out and the session is terminated

**Basic Flow or Main Scenario:**

1. Use selects the logout option from interface
2. The system asks for confirmation of logout
3. System terminates user session
4. User is logged out, is now in

**Extensions or Alternate Flows:**

* If the user cancels the logout confirmation, their login session continues

**Exceptions:**

* If the session termination fails, the system maintains the session and retries the termination

**Related Use Cases:** User Login

**Use Case: 08**

**Use Case Name:** Write group chat log to file

**Relevant Requirements:** User must have IT Admin credentials

**Primary Actor:** ITUser

**Pre-Conditions:**

* IT User must have successfully logged in with valid credentials
* IT User must be currently viewing the group chat to log

**Post-Conditions:**

* Text file of the chat is written to a .txt file on the client

**Basic Flow or Main Scenario:**

1. IT Admin views a group chat
2. IT Admin in the GUI clicks button to log the chat
3. The chat conversation is written to .txt on the client

**Extensions or Alternate Flows:**

* If the group chat

**Exceptions:**

* If file writing operation fails, the user is notified

**Related Use Cases:**

**Use Case: 09**

**Use Case Name:** Manage Group Chats

**Relevant Requirements:** Group chat must exist, users must be logged in.

**Primary Actor:** User

**Pre-Conditions:**

* Group chat must have already been created.
* Users must be logged in.
* Must have other user accounts to add to the group.

**Post-Conditions:**

* User will be added from group chat.
* User will be deleted.
* Group chat will be renamed.

**Basic Flow or Main Scenario:**

1. User selects the manage icon.
2. Select one of the available manage options.
3. User will be prompted to add/delete user, or rename chat.

**Extensions or Alternate Flows:**

* Add user to group chat
* Delete user from chat
* Rename chat

**Exceptions:** Group chat doesn’t exist.

**Related Use Cases:** Create Group Chat

# UML Class Diagram - Description

|  |
| --- |
| Abstract User |
| - id : String  - Used as a way to identify a user by a personal id, unique to each person  - firstName : String  - Storing the User’s First name  - lastName : String  - Storing the User’s Last name  - username : String  - Users usable login item, used in authentication  - password : String  - Users usable login item, used in authentication  - isDisabled: Boolean = false  - Current status of a User, disabled user is unusable but are still kept records  - isAdmin: Boolean = false  - Authorization of a user, Admin users have elevated properties.  - chats : Chat[]  - List of all chats the user is in, used to load specific chats |
| + AbstractUser(firstName : String, lastname: String, password: String)  Constructor – Base creation of a user  + getFirstName() : String  - Getter for personal first name, used in UI interface  + getLastName() : String  - Getter for personal last name, used in UI interface  + getUserName() : String  - Getter for Username  + getId() : String  - Getter for id, users cannot have overlapping IDs  + getChats() : Chat[]  - getter for list of Chats, Used in the UI interface  + toString() : String  - String return for a descriptor of the main parts of User  + createChat(users : User[]) : void  - Important function for the User to create a chat with one or more people including self  + addUserToChat(user : User) : void  - Important function for adding a user to a specific chat  + sendMessage()  - Important function for sending a message in a chat. |

|  |
| --- |
| IT Admin |
|  |
| + ITAdmin(firstName : String, lastname: String, passwordString, isAdmin Boolean = true)  - Constructor for creating an Admin  + createUser() : void  - User creation is controlled only by IT Admin  + disableUser(user : User) : void  - User status is controlled only by IT Admin  + enableUser(user: User) : void  - User status is controlled only by IT Admin  + writeChatLog(chat : Chat) |

|  |
| --- |
| User |
|  |
| + User(firstName: String, lastName: String, username: String, password : String)  - Constructor for creating a User |

|  |
| --- |
| GUI |
| Client: client   * GUI holds Client to link logic from GUI to backend   - JSplitPane: splitPane   * Adds split pane interactivity to GUI   - JPanel: leftPanel   * Creates a section on the left side of the GUI for elements to be added   - JPanel: rightPanel   * Creates a section on the right side of the GUI for elements to be added   - JLabel: nameLabel   * Adds a label to the top of a panel   - JScrollPane: privateChats   * Makes private chats scrollable for multiple chats to be visible   - JScrollPane: teamChats   * Adds scrolling functionality to team chats   - JLabel: chatInfo   * Adds a label to the top of a panel   - JPanel: messagesPanel   * A panel to hold messages   - JScrollPane: listOfMessages   * Makes the messages scrollable for when there’s multiple messages in a chat   - JPanel: inputPanel   * Creates a panel which will possess a means to enter text   - JTextField: textField   * Allows text to be inputted at the bottom of the chat   - JButton: sendButton   * A button that allows the sending of messages |
| + GUI(client : Client)   * Initializes user interface, binds client to GUI   + displayServerResponse() : void   * Updates GUI with server responses   - handleUIEvent() : void   * Handles user interactions like clicks and text input |

|  |
| --- |
| Message |
| - id : String  - Used to identify a message based on ID  - createdAt : LocalDateTime  - Current time message is created, saved by server and shown to users  - content : String  - Message content, saved by server and shown to users  - author : User  - The original message sender, saving purposes  - chat: Chat  - The current chat message was sent in |
| + Message(content : String, sender: User)  - Constructor for the message which takes the sender, and the content sent  + getId() : String  - Getter for ID  + getCreatedAt() : LocalDateTime  - Getter of the messages sent date and time  + getSender() : User  - Getter for the User that sent the message  + toString() : String  - String representation of the message to be saved in the server |

|  |
| --- |
| Chat |
| - id : String  - Used to identify a Chat based on ID  - messages : Message[]  - A collection of all the messages in the chat  - owner : User  - The chat creator  - roomName : String  - The chat name, groups are named, other person’s name if private  - chatters : User[]  - List of all the Users in the chat  - isPrivate : Boolean = false  - Privacy flag |
| + Chat(chatOwner : User, name : String)  - Constructor of a chat with the owner and the name of the chat  + addChatter()  - Adding a user into the chat  + removeChatter()  - Removing a user from the chat  + addMessage(msg : Message)  - sent messages are added to the chat  + changePrivacy(newState : boolean)  - change chat flag  + toString() : String  - String representation of chat, collection of messages, saved in the server database |

|  |
| --- |
| Packet <<Serializable>> |
| - count : int = 0  - A static variable counter that can be used to make unique identifiers for each packet  - id : String  - The unique identifier  - actionType : String  - Tells the purpose and ultimately how the Packet is supposed to be processed. It’s particularly relevant to then how to handle **actionArguments** An example would be: “LOGIN”  - timeCreated : LocalTime  - Time the packet is created, in particular to be useful for sorting data if needed  - senderId : String  - Denotes who created the packet. If client created it, it’ll the value will be their ID, if server created it – it will be “SERVER” |
| + Packet(actionType : String, actionArguments : String[], senderId : String)  - Constructor of a Packet which can be dynamically made to cover many functionalities  + getId()  - Returns instance variable **id**  + getTimeCreated() : LocalTime  - Returns instance variable **timeCreated**  + getSenderId() : String  - Returns instance variable **senderId**  + getActionType() : String  - Returns instance variable **actionType**  + getActionArguments() : String  - Returns instance variable **actionArguments** |

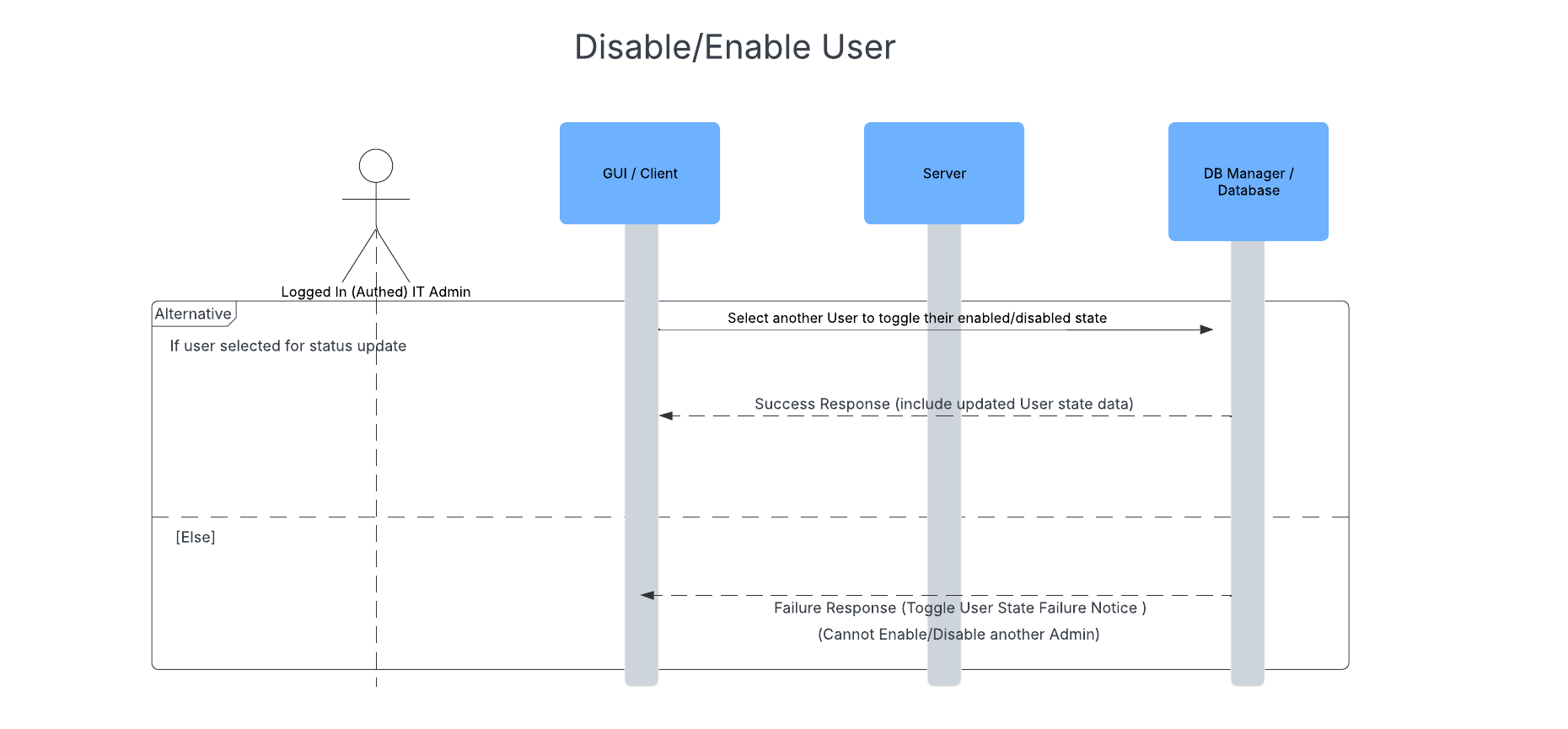
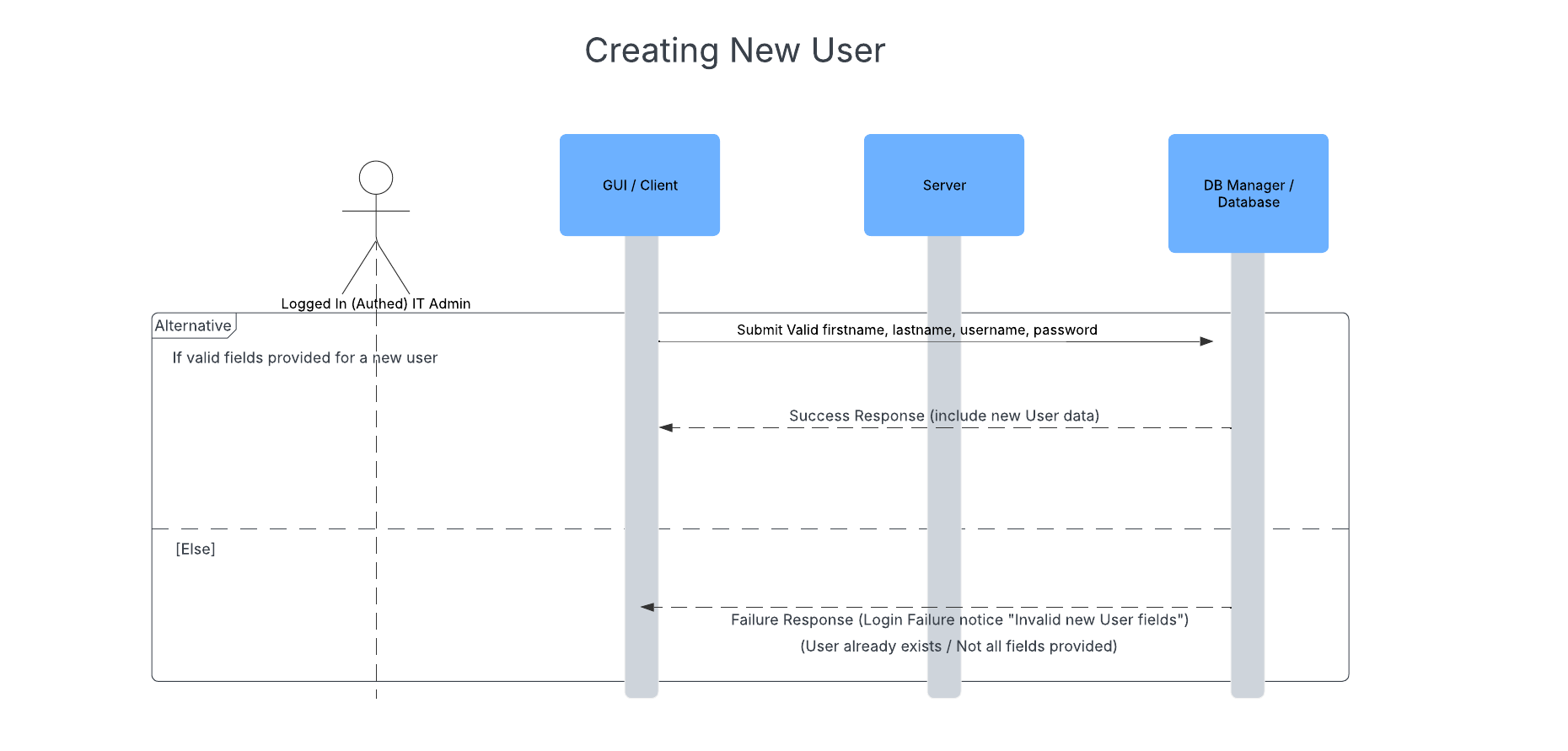
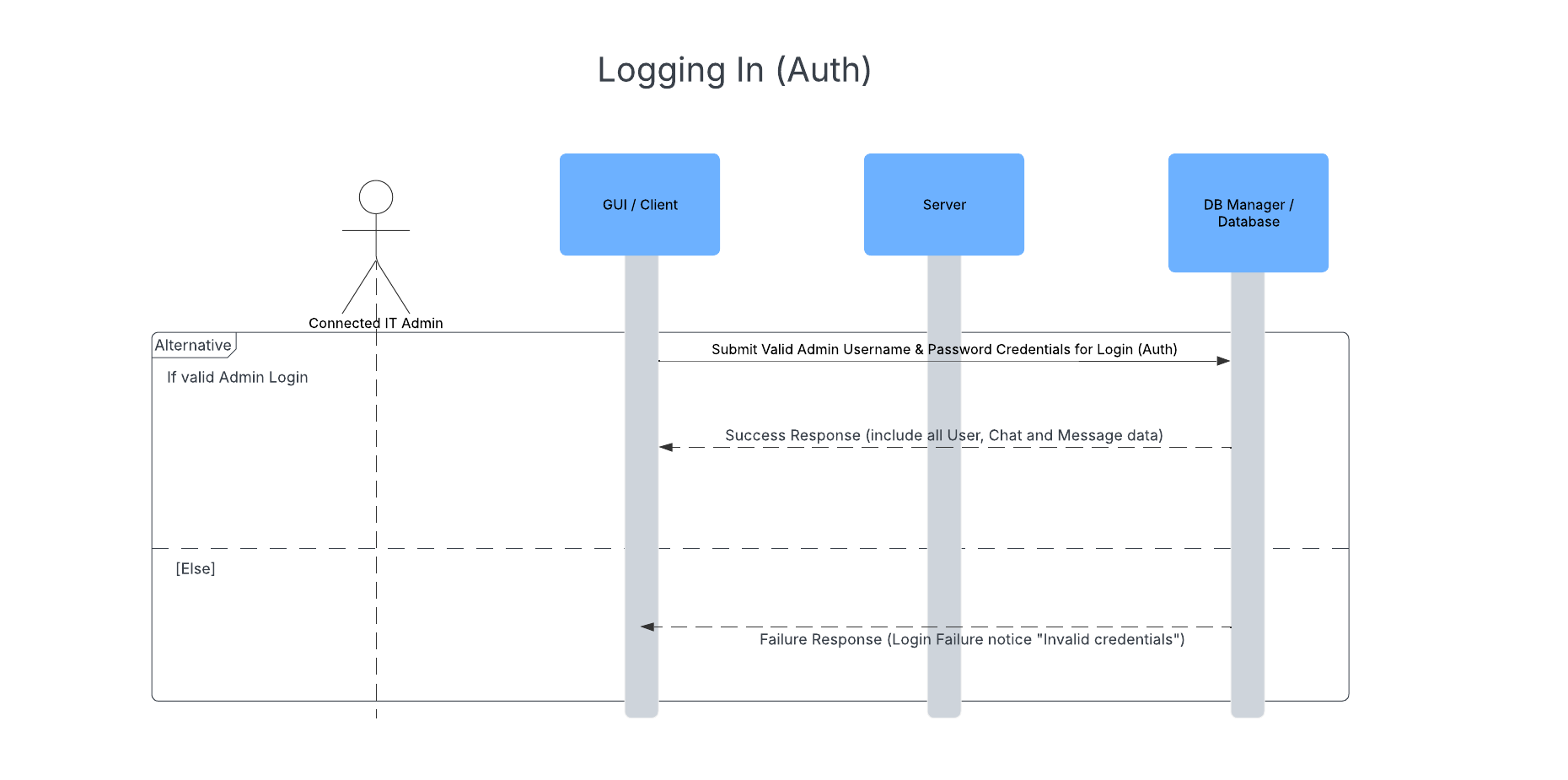
|  |
| --- |
| Server |
| - clients : Hashmap<userId : String, ClientHandler>  - All active clients with their userId as the key which allows for fast lookups  - dbManager : DBManager  - Instance of the DBManager  - port : int  - Port the server is listening on  - IP : String  - IP Addressed the server is using |
| + Server(port : int, IP : String)  - Constructor that creates the server with the designated port and IP address  + connect() : void  - Sets up and begins listening for clients  + disconnect() : void  - Stops active clients to shutdown the server  + receivePacket(clientId: String, packet: Packet) : void  - Handles the incoming packet appropriately depending on that packet’s information and instructions  + sendErrorMessage(userId: String, errorMessage: String) : void  - Sends error message to User  + sendSuccessMessage (userId: String, successMessage: String) : void  - Sends success message to User  + sendPacketToUsers(packet: Packet, userIds : String[]) : void  - Send a packet to one or more users |

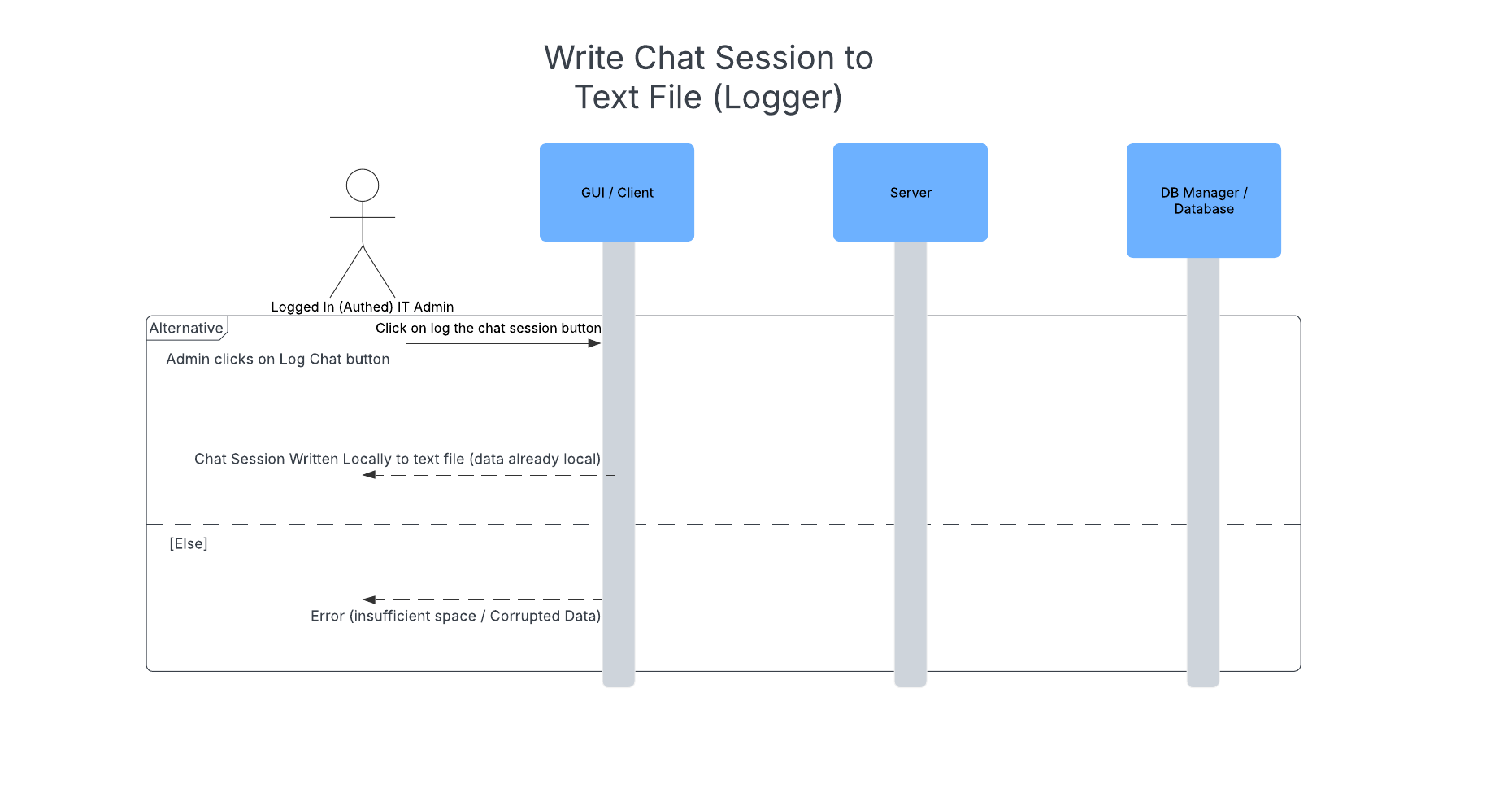
|  |
| --- |
| ClientHandler <<Runnable >> |
| - clientSocket : Socket  - Client’s socket  - userId : String  - associated user’s userId  - inputStream : ObjectInputStream  - Stream to receive information that client sent  - outputStream : ObjectOutputStream  - Stream to send information to the client |
| + ClientHandler(socket : Socket, server : Server)  - Constructor to create client handler with the server and socket  + start() : void  - Begins listening  + stop() : void  - Stops listening  + setUserId(userId : String) : void  - Sets instance variable userId  + getUserId() : String  - Gets instance variable userId  + getInputStream() : ObjectInputStream  - Returns instance variable inputStream  + getOutputStream() : ObjectInputStream  - Returns instance variable outputStream  + sendPacket(packet : Packet) : void  - Sends packet to the client using the output stream |

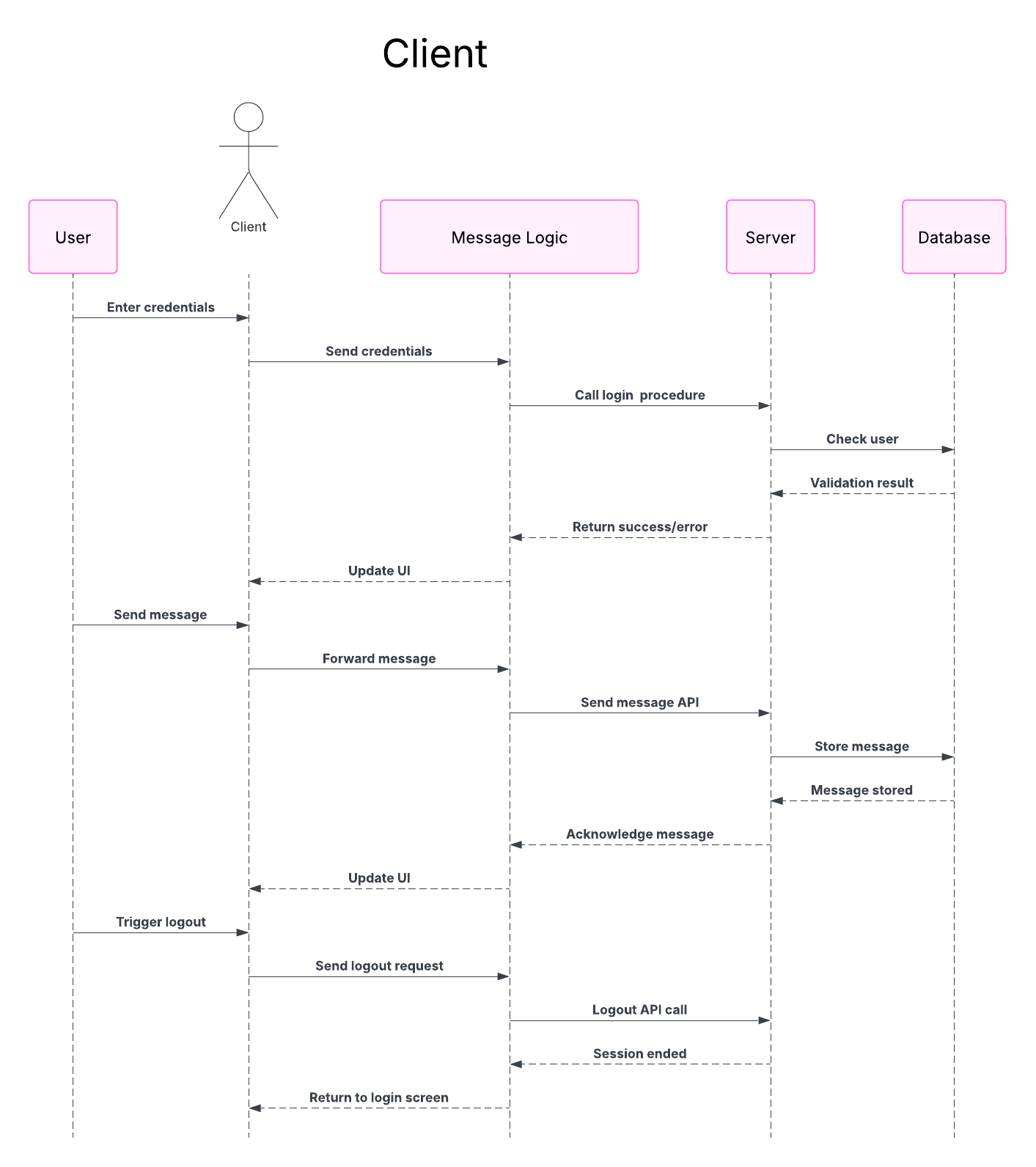
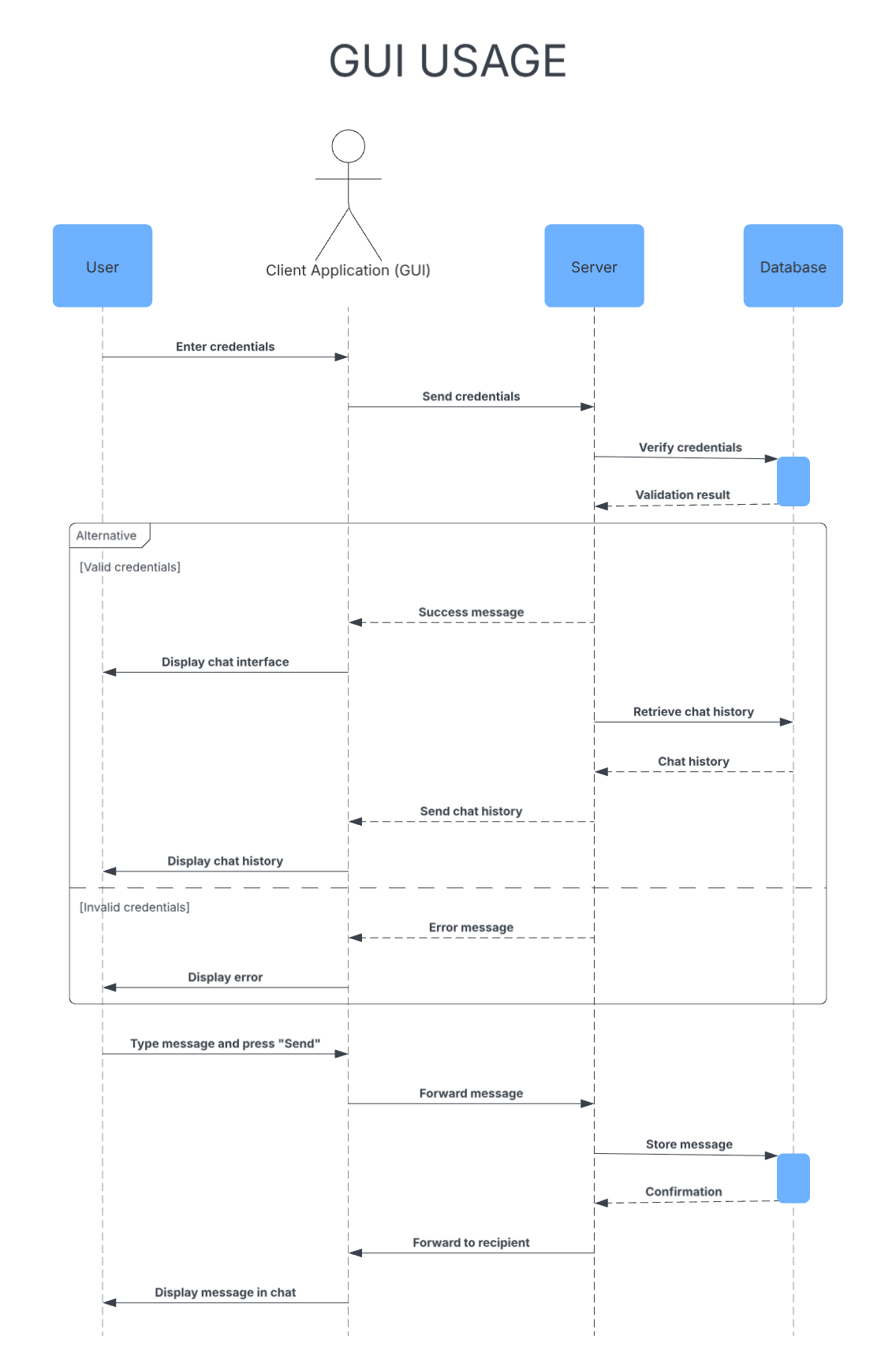
|  |
| --- |
| DBManager |
| - users : Hashmap<userId : String, User>  - All Users as a hashmap for fast lookups by using their userId  - chats : Hashmap<chatId : String, Chat>  - All Group Chats as a hashmap for fast lookups by using their chatId  - messages : Hashmap<messageId : String, Message>  - All Messages as a hashmap for fast lookups by using their messageId  - txtFilePath : String  - directory path that the users.txt, chats.txt and messages.txt reside (ex: “./files/”)  - userTxtFilename : String  - text filename for the persisted user information (ex: users.txt)  - chatTxtFilename : String  - text filename for the persisted Chat information (ex: chats.txt)  - messageTxtFilename : String  - text filename for the persisted message information (ex: messages.txt) |
| + DBManager(filepath: String, userTxtFilename : String, chatTxtFilename : String, MessageTxtFilename : String)  - Constructor that creates the DB Manager with the arguments that allow to set the path the text files exist and what their names are  + loadAllFiles() : void  - uses the User, Chat and Message text files to build their corresponding hashmaps in memory  + checkLoginCredentials(username : String, password : String) : User  - verifies the username and password as being valid by checking whats in the user.txt file  + sendUserAllFreshData() : void  - When User initially logs in they are sent all their associated chats, messages and the user list  + getUserById(userId: String) : User  - Get user by their userId  + getChatById(chatId: String) : Chat  - Get chat by the chatId  + getMessageById(messageId: String) : Message  - Get a message by its messageId  + fetchAllActiveUsers() : List<User>  - Get all the active users in the system  + fetchAllInactiveUsers() : List<User>  - Get all the de-activated users in the system – **(FOR ADMIN)**  + fetchAllChats() : List<Chat>  - Get all the chats in the system  + fetchAllMessages() : List<Message>  - Get all the messages in the system  + getChatsForUser(userId : String) : List<Chat>  - Get all chats that are associated with a particular user via their userId  + writeNewUser(user : User) : void  - Write the user to the DB **users.txt** file  + writeNewChat(chat : Chat) : void  - Write the chat to the DB **chats.txt** file  + writeNewMessage(message : Message) : void  - Write the message to the DB **messages.txt** file  - stringToUser(userString : String) : User  - Parse the string in the format of “username/password/id/firstName/lastName/isDisabled/isAdmin” from **users.txt** file to an actual User object  - stringToChat(chatString : String) : Chat  - Parse the string in the format of “id/owner/roomName/[userId1, userId2, userId3]/isPrivate” from **chats.txt** file to an actual Chat object  - stringToMessage(MessageString : String) : Message  - Parse the string in the format of “id/createdAt/content/authorId/chatId” from **messages.txt** file to an actual Message object |

|  |
| --- |
| Client |
| - socket: Socket   1. Represents a server socket used for the client-server communication.   - isConnected : Boolean   * Returns true or false depending if a connection is made or not   - targetIP : String  - A string that holds the target IP of the connection. - targetPort : String  - A string that holds the target port of the destination of connection. - chats : Chat[]  - Holds an array of chat object for messaging sessions.  - users : User[]  - Stores an array of user objects for the client. - userId : String  - A unique identifier for the client for session tracking. - isITAdmin : Boolean  - A Boolean that indicates if the client has IT user privileges. |
| + Client(targetIp: String, targetPort: String)   * Initializes client with specified IP and port.   + login(username: String, password: String) : void   * Autheticated user based on provided credentials   + sendMessage(authorId: String, chatId: String, content : String)   * Sends a message to specified chat.   + createChat(chatName : String, isPrivate : boolean) : void   * Creates a new chat, has an option on whether or not chat is private   + updateState() : void   * Refreshes client’s current data and connection status.   + createUser() : void   * Creates a new user in the system.   + enableUser() : void   * Grants a user access.   + disableUser() : void   * Restricts a user access.   + saveChatToTxt(chat : Chat) : void   * Saves a specified chat’s contents to a file.   + logout() : void   * Logs the current user out and closes the session. |

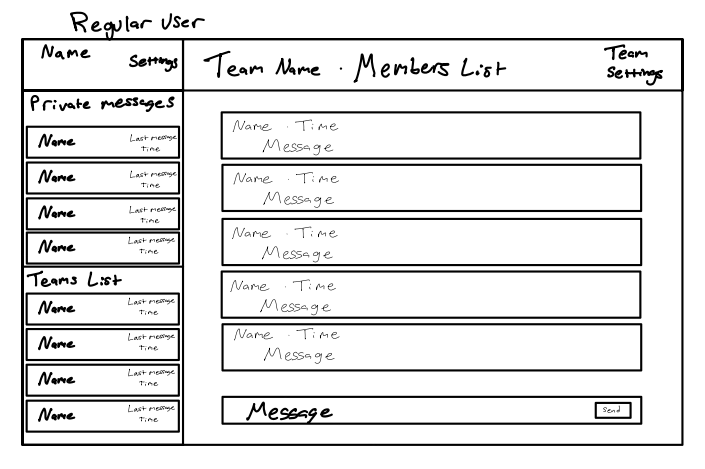
# UML Sequence Diagram







# Paper Prototype – Interface & Data



A screenshot of a chat

AI-generated content may be incorrect.

# Prototype Description

* 1. Login Pane
     1. Option pane that handles User name and Passwords with hidden password input.
     2. Buttons with Login and Cancel
  2. Main Frame
     1. Header
        1. Two horizontal boxes – Right Box with Name and Left Box with Chat details
        2. Name includes the Current User’s First and Last Name
        3. Chat Details include The Current Chat Name and a list of users also in the chat
     2. Left Column
        1. Two Vertical boxes that split between private chats and group chats
        2. Both are scrollable if there are too many chats in them.
        3. Each chat is a button that contains the recipient or chat name, last message time, and members
     3. Right Column
        1. The main chat section where each message is shown in chronological order and is updated from the bottom with older messages being above the latest message.
        2. Each message contains the name of the sender and the time the message was sent, as well as the main message content wrapping if it gets to the end of the bar
        3. The bottom of the screen has a text input field to send a message as well as a send button which will clear the field and send the message that was in the field.
  3. Main Frame – IT
     1. Header
        1. Same as the regular user but will have a badge under their name with ‘IT Admin’ as to show them having a different UI
     2. Left Column
        1. Similar to the regular user but also has access to all other messages and are bordered with a different color to determine a chat that they can see but not message in.
     3. Right Column
        1. Other chats can only be viewable and not messagable.