Chat Application Project Report

Project Group 6 – DAT055

Hussein Hafid

Jan Rahimi

Mohamad Alzein

Zakaria Abulkadir

March 21, 2025

Abstract

This report presents a chat application developed as part of the Object Oriented Applications (DAT055) course, featuring real-time text and image messaging alongside persistent data storage. The project adopts a client-server architecture, with a Java-based server managing user accounts, chat rooms, and database interactions via PostgreSQL, and a JavaFX client handling user interface and real-time updates. Core functionalities include account creation, login authentication, searching and joining chat rooms, and sending both text and images. Concurrency is achieved on the server side by dedicating a threaded handler to each incoming connection, while the client employs asynchronous listeners to avoid blocking graphical updates. The Model-View-Controller (MVC) pattern underpins the organization of client logic, enabling clear separation of concerns between scene rendering and network communication. Key challenges included establishing a robust TCP protocol, integrating multithreading with JavaFX, and coordinating team efforts with version control. Despite limitations in input validation and security, the resulting software fulfills essential requirements and serves as a foundation for potential enhancements such as encryption, refined UI features, and private chat rooms. This project highlights the importance of methodical design, careful concurrency management, and iterative development in creating a functional, scalable application.

Contents

Chapter 1

Introduction

This report documents the development of a chat application undertaken by a team of students over an eight-week period. The chat application was created as part of the Object Oriented Applications (DAT055) course and includes core features such as user account creation, global chat rooms, real-time messaging for text and images, and persistent data storage. The purpose of the project was to gain practical experience with object-oriented design, concurrency, JavaFX for graphical user interfaces, and relational database systems (in this case, PostgreSQL). Throughout the process, emphasis was placed on applying robust software engineering principles, including structured design patterns and clear separation of concerns between client and server components.

Chapter 2

Project Requirements

The DAT055 course outlined a set of high-level requirements for the project. The application was expected to handle multiple concurrent users who could browse and connect to chat rooms, exchange messages reliably, and maintain a persistent record of all conversations. The course specification also required following a Model-View-Controller (MVC) pattern, ensuring that user interaction, data manipulation, and presentation logic remained well-organized. In particular, the final product needed a functioning graphical interface, as well as support for both text and image messages within each chat.

During early analysis, these requirements were interpreted to include the creation of user accounts, joining multiple global chat rooms, sending and receiving real-time messages, and storing all chat data in a format that allowed users to retrieve previous conversation history. Additionally, the course recommended the use of Java (including JavaFX or Swing) for the user interface, along with guidance to properly cite any resources or AI tools consulted during development.

Chapter 3

Scope of the Application

This chat application is designed to address the central requirements, while some functionalities remain deferred for future enhancements. The project delivers a practical and direct chat system in which users can sign up or log in, create or join chat rooms, and exchange text and image messages. The system enforces minimal input validation, and users are permitted to choose any valid username or email address, which are stored in the PostgreSQL database without strict format checking. Although only a global chat environment exists in this version, the structure allows potential expansion to private or invite-only rooms later.

Since the project primarily serves as an educational exercise, it does not implement certain advanced features that might be common in production systems, such as strong password requirements, encryption for network traffic, or sophisticated user role management. Despite these limitations, the basic framework has been designed in such a way that these features could be added later if time and resources permit.

Chapter 4

CRC Cards

Before implementation, CRC (Class-Responsibility-Collaboration) cards were created to outline the initial design of key classes. These cards helped clarify class responsibilities and their interactions with other components of the system. While the final implementation evolved beyond this initial structure, the CRC approach provided valuable insights into defining the core architecture of the chat application. The table bellow showcases the CRC cards we designed in the early design phase of the project:

Class Name	**Responsibilities**	**Collaborators**
`UserController`	Manages user-related actions, such as login and logout. Communicates with the `AccountHandler` to retrieve user data and validate operations.	`AccountHandler`
`MessageController`	Handles user interactions related to messages. Forwards requests to the `MessageHandler` and validates input.	`MessageHandler`
`ChatroomController`	Manages chatroom-related interactions, including joining, leaving, and creating chat rooms. Relays data to the `ChatHandler`.	`ChatHandler`
`AccountHandler`	Interfaces with the database to manage user authentication, account creation, and deletion. Ensures secure handling of user credentials.	`Model`, `UserController`
`MessageHandler`	Manages message data, including storage, retrieval, and deletion. Interacts with the database to persist message history.	`Model`, `MessageController`
`ChatHandler`	Manages chatroom data, including creation, deletion, and membership tracking. Updates the model with chatroom state changes.	`Model`, `ChatroomController`
`Model`	Centralized data storage for users, messages, and chat rooms. Interfaces with the database to persist user activity.	`MessageHandler`, `ChatHandler`, `AccountHandler`
`View`	Renders the user interface, including login screens, chat rooms, and message history. Relays user inputs to controllers.	`MessageController`, `UserController`, `ChatroomController`
`User`	Represents user-related data, such as usernames and authentication status. Provides methods for retrieving or updating user attributes.	`Model`, `AccountHandler`
`Message`	Represents an individual message, including sender, content, and timestamp. Provides methods for message creation and modification.	`Model`, `MessageHandler`
`ChatRoom`	Represents a chatroom, including its name, participants, and message history. Manages operations specific to chat rooms.	`Model`, `ChatHandler`

Table 4.1: CRC Cards for the Chat Application

Chapter 5

Design of the Chat Application

5.1 Overview

The chat application follows a client-server architecture, where a dedicated server manages all chat room data and user authentication while clients connect to the server to interact with chat rooms. The application is designed using Java, with PostgreSQL for data persistence and JavaFX for the graphical user interface. The communication between the client and the server is facilitated using TCP sockets, allowing real-time message exchange.

5.2 Server-Side Design

The server is responsible for handling multiple client connections, managing chat rooms, storing messages, and processing user authentication. The main component of the server is the `ChatServer`, which listens for incoming TCP connections and assigns a dedicated `ClientHandler` to each connected user. The server also includes a `NotificationHandler` that listens for changes in the database and notifies relevant clients when new messages arrive.

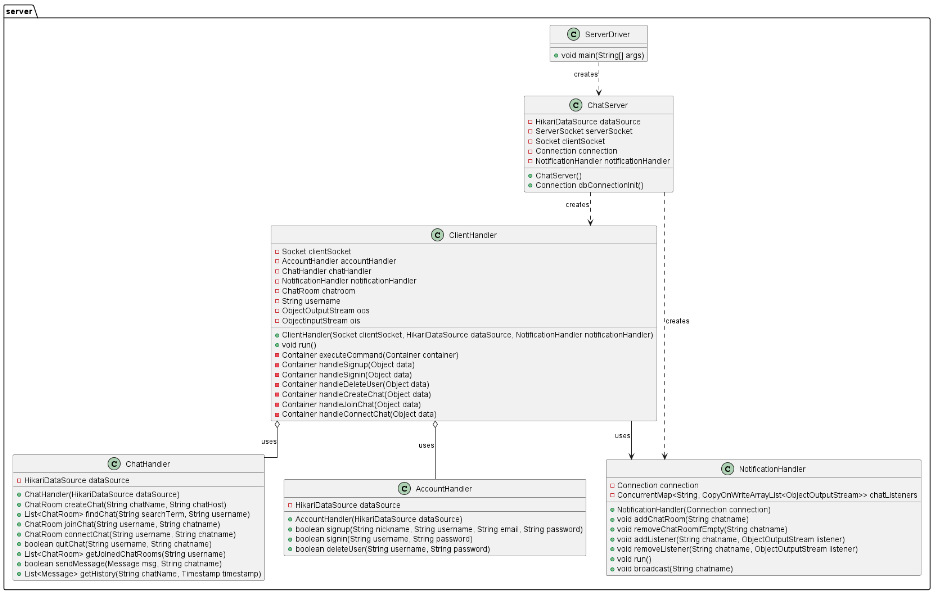
server
ServerDriver
void main(String[] args)
creates
ChatServer
HikariDataSource dataSource
ServerSocket serverSocket
Socket clientSocket
Connection connection
NotificationHandler notificationHandler
ChatServer()
Connection dbConnectionInit()
creates
creates
ClientHandler
Socket clientSocket
AccountHandler accountHandler
ChatHandler chatHandler
NotificationHandler notificationHandler
ChatRoom chatroom
String username
ObjectOutputStream oos
ObjectInputStream ois
ClientHandler(Socket clientSocket, HikariDataSource dataSource, NotificationHandler notificationHandler)
void run()
Container executeCommand(Container container)
Container handleSignup(Object data)
Container handleSignin(Object data)
Container handleDeleteUser(Object data)
Container handleCreateChat(Object data)
Container handleJoinChat(Object data)
Container handleConnectChat(Object data)
uses
uses
uses
ChatHandler
HikariDataSource dataSource
ChatHandler(HikariDataSource dataSource)
ChatRoom createChat(String chatName, String chatHost)
List<ChatRoom> findChat(String searchTerm, String username)
ChatRoom joinChat(String username, String chatname)
ChatRoom connectChat(String username, String chatname)
boolean quitChat(String username, String chatname)
List<ChatRoom> getJoinedChatRooms(String username)
boolean sendMessage(Message msg, String chatname)
List<Message> getHistory(String chatName, Timestamp timestamp)
AccountHandler
HikariDataSource dataSource
AccountHandler(HikariDataSource dataSource)
boolean signup(String nickname, String username, String email, String password)
boolean signin(String username, String password)
boolean deleteUser(String username, String password)
NotificationHandler
Connection connection
ConcurrentMap<String, CopyOnWriteArrayList<ObjectOutputStream>> chatListeners
NotificationHandler(Connection connection)
void addChatRoom(String chatname)
void removeChatRoomIfEmpty(String chatname)
void addListener(String chatname, ObjectOutputStream listener)
void removeListener(String chatname, ObjectOutputStream listener)
void run()
void broadcast(String chatname)

Figure 5.1: Server Application Class Diagram

5.3 Client-Side Design

The client application follows the Model-View-Controller (MVC) pattern, ensuring a clear separation of concerns between data handling, user interface, and server communication. The user interface is managed through JavaFX, where a `SceneManager` dynamically loads different views, such as the login screen, main chat view, and settings page. Communication with the server is handled through `ClientSender` and `ClientReceiver`, which ensure that messages and commands are sent and received asynchronously.

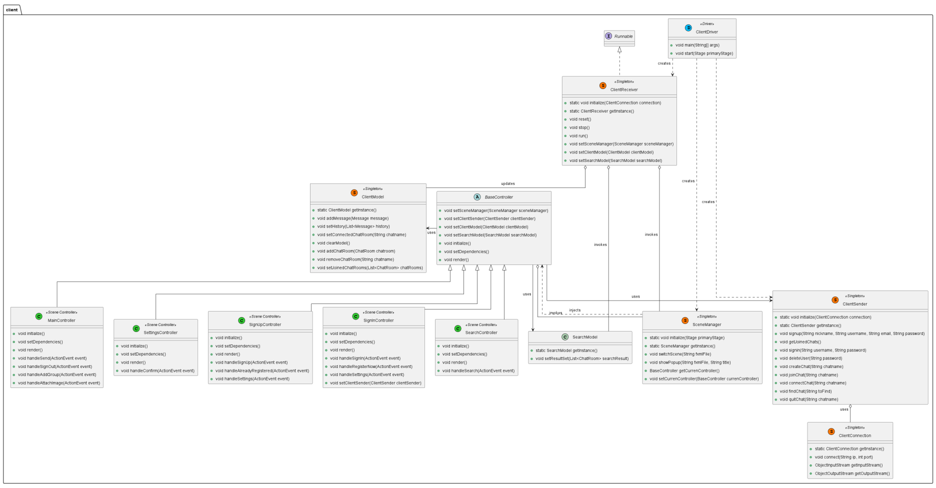
client
Runnable
«Driver»
ClientDriver
void main(String[] args)
void start(Stage primaryStage)
creates
«Singleton»
ClientReceiver
static void initialize(ClientConnection connection)
static ClientReceiver getInstance()
void reset()
void stop()
void run()
void setSceneManager(SceneManager sceneManager)
void setClientModel(ClientModel clientModel)
void setSearchModel(SearchModel searchModel)
creates
creates
updates
invokes
invokes
«Singleton»
ClientModel
static ClientModel getInstance()
void addMessage(Message message)
void setHistory(List<Message> history)
void setConnectedChatRoom(String chatname)
void clearModel()
void addChatRoom(ChatRoom chatroom)
void removeChatRoom(String chatname)
void setJoinedChatRooms(List<ChatRoom> chatRooms)
uses
BaseController
void setSceneManager(SceneManager sceneManager)
void setClientSender(ClientSender clientSender)
void setClientModel(ClientModel clientModel)
void setSearchModel(SearchModel searchModel)
void initialize()
void setDependencies()
void render()
uses
uses
invokes
injects
«Scene Controller»
MainController
void initialize()
void setDependencies()
void render()
void handleSend(ActionEvent event)
void handleSignOut(ActionEvent event)
void handleAddGroup(ActionEvent event)
void handleAttachImage(ActionEvent event)
«Scene Controller»
SettingsController
void initialize()
void setDependencies()
void render()
void handleConfirm(ActionEvent event)
«Scene Controller»
SignUpController
void initialize()
void setDependencies()
void render()
void handleSignUp(ActionEvent event)
void handleAlreadyRegistered(ActionEvent event)
void handleSettings(ActionEvent event)
«Scene Controller»
SignInController
void initialize()
void setDependencies()
void render()
void handleSignIn(ActionEvent event)
void handleRegisterNow(ActionEvent event)
void handleSettings(ActionEvent event)
void setClientSender(ClientSender clientSender)
«Scene Controller»
SearchController
void initialize()
void setDependencies()
void render()
void handleSearch(ActionEvent event)
SearchModel
static SearchModel getInstance()
void setResultSet(List<ChatRoom> searchResult)
«Singleton»
SceneManager
static void initialize(Stage primaryStage)
static SceneManager getInstance()
void switchScene(String fxmlFile)
void showPopup(String fxmlFile, String title)
BaseController getCurrenController()
void setCurrenController(BaseController currenController)
«Singleton»
ClientSender
static void initialize(ClientConnection connection)
static ClientSender getInstance()
void signup(String nickname, String username, String email, String password)
void getJoinedChats()
void signin(String username, String password)
void deleteUser(String password)
void createChat(String chatname)
void joinChat(String chatname)
void connectChat(String chatname)
void findChat(String toFind)
void quitChat(String chatname)
uses
«Singleton»
ClientConnection
static ClientConnection getInstance()
void connect(String ip, int port)
ObjectInputStream getInputStream()
ObjectOutputStream getOutputStream()

Figure 5.2: Client Application Class Diagram

5.4 Client State Diagram

The client transitions between multiple states throughout its lifecycle. Initially, the client is in a *Disconnected* state. Once a connection is established with the server, the client moves into the *Connected* state. After successful authentication, the client enters the *Authenticated* state. The client remains in this state until it joins a chat room, at which point it transitions to the *Connected to Chat Room* state. If the connection is lost, the client returns to the *Disconnected* state.

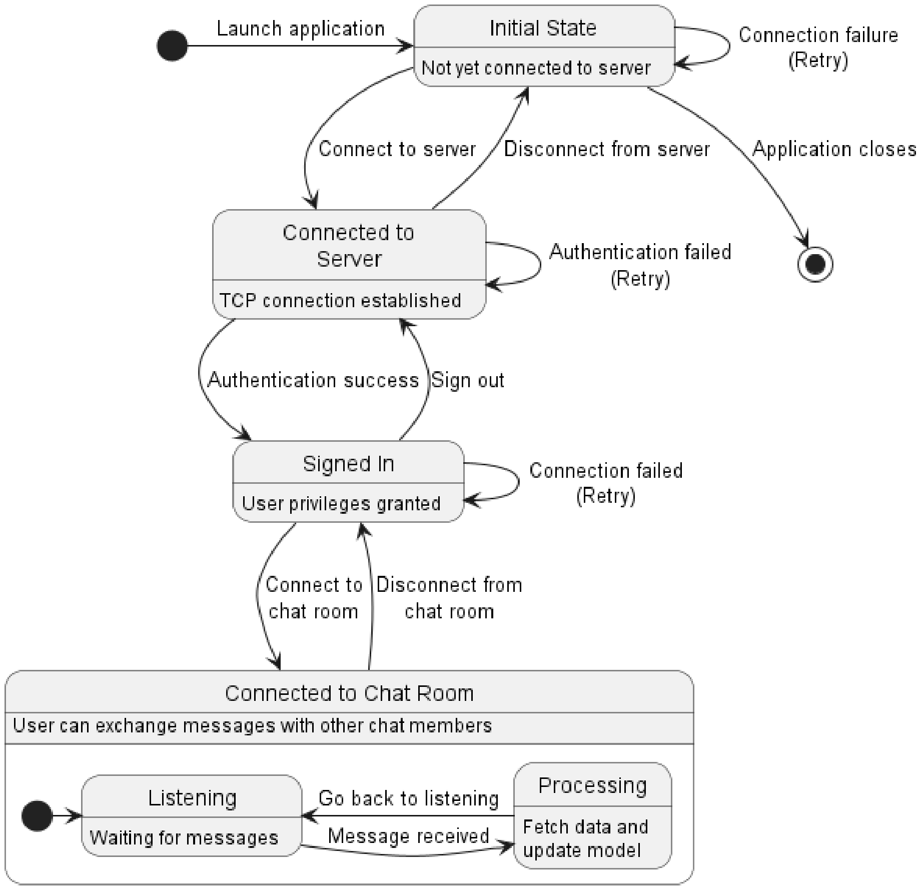

Figure 5.3: Client State Diagram

5.5 Login Procedure Sequence Diagram

The login procedure consists of several interactions between the client and the server. When a user enters their credentials and submits the login request, the client sends a `signin` command to the server. The server then verifies the credentials by querying the database. If the login is successful, the server responds with a success message, and the client transitions to the main chat view.

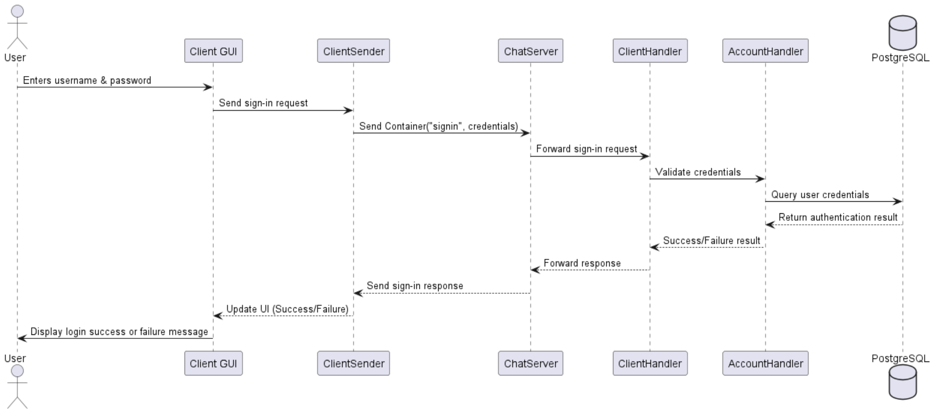

Figure 5.4: Login Procedure Sequence Diagram

Chapter 6

Implementation and Design Improvements

The design underwent several improvements during implementation. One of the foremost decisions was to adopt a relational database for storing all persistent data, including user credentials, chat rooms, and messages. This choice simplified data access patterns and improved reliability. Another key improvement emerged in the approach to concurrency: the team decided to run each connected client on its own thread in the server, making it feasible to handle multiple simultaneous chat operations without one client blocking another.

Additionally, the introduction of `Container` objects to package commands and arbitrary data proved efficient. Even though this strategy carries risks (such as sending large or maliciously crafted objects), it served as a quick solution to achieve a functioning protocol. Future iterations might adopt a more secure approach, such as sending JSON messages over encrypted channels, to better protect sensitive data.

Chapter 7

Workflow

Early in the project, a Waterfall-like approach was planned, with defined phases for requirements, design, and implementation. In practice, the team found that changes to the design were often necessary while coding was underway, leading to partial overlap of tasks. The attempt to split the team into server-side and client-side subgroups was only partially successful, as some members had limited availability or experience. The resulting imbalance meant that major architectural decisions were occasionally made by a smaller subset of the team.

Version control with Git and GitHub initially posed difficulties, especially with merging branches and preventing conflicts. It was discovered that working directly on the `main` branch caused significant confusion. Only in the later stages was a more systematic approach adopted, allowing pull requests to be used for code reviews and merges. This shift improved code quality and reduced the likelihood of major regressions.

Chapter 8

Running the Application

8.1 Server Application

The server can be compiled using Maven or by invoking a `make` command if the provided automation scripts are used. After configuring PostgreSQL through the project's initialization steps, running the `ServerDriver` class starts the application in listening mode. The server remains active, awaiting incoming TCP connections on the specified port. Notifications from the database are listened to in a dedicated thread, so any insertion of new messages triggers updates to the subscribed clients.

8.2 Client Application

The client requires JavaFX libraries and can similarly be launched by compiling and running the `ClientDriver` class. Upon startup, the client presents a Sign-In View, but it is first necessary to configure the server's IP address and port within the Settings View. Once the socket connection is established, the user can log in or register for an account. Successful login transitions to the Main View, which displays joined chat rooms, allows searching or creating new rooms, and provides a means to send text or image messages in real time.

Chapter 9

System Requirements

The application relies on Java 22 or higher, as well as a Bash shell and `make` for automating the setup process. On the server side, PostgreSQL must be installed and running, typically at version 15 or higher, with proper network accessibility for the client. The client side requires JavaFX, although Maven is used to pull in dependencies automatically. Firewall or network restrictions may impede successful connections, as the system expects an open TCP port on the server host.

Chapter 10

Usage

When starting the client, the Sign-In View appears. If an account already exists, the user can provide a username and password to authenticate. Otherwise, a link in the Sign-In View navigates to the Sign-Up View, where new credentials may be registered. Once logged in, the Main View appears and displays all joined chat rooms on the left, with a chat container in the center for messages. A text field permits entering and sending text, while an “Attach Image” button opens a file chooser to send images in PNG, JPG, or GIF formats.

The Main View also features an “Add” button that opens the Search View. In that scene, users can search for existing chat rooms by name or create a new one. Upon locating a desired room, selecting it automatically issues a join request to the server; a successful join results in that room appearing in the Main View’s list. By clicking on a joined room, the system retrieves the message history from the server and listens for new incoming messages. This mechanism relies on the notification functionality implemented server-side, which broadcasts changes to all connected members in real time.

Chapter 11

Use of AI in the Project

Artificial Intelligence (AI) tools were leveraged throughout the development of this project to enhance productivity, facilitate problem-solving, and improve the quality of both the implementation and documentation. The use of AI was guided by ethical considerations, ensuring that AI-assisted content was reviewed, refined, and integrated responsibly.

AI played a significant role in assisting with software development, particularly in debugging and optimizing code. During the implementation phase, AI tools were consulted for identifying and resolving syntax errors, runtime exceptions, and logic flaws. Additionally, AI provided suggestions for optimizing Java concurrency, socket communication, and database interactions. It also offered guidance on best practices for structuring the client-server architecture and assisted in configuring PostgreSQL, Maven dependencies, and JavaFX. Despite AI's assistance in debugging, all final decisions regarding code modifications were made manually, ensuring that AI-generated suggestions aligned with the intended functionality and course requirements.

Throughout the project, AI tools were used as a research assistant to efficiently find relevant documentation, tutorials, and technical references. This included locating official Java and PostgreSQL documentation, understanding JavaFX scene management and event handling, gathering information on best practices for using TCP sockets in Java, and exploring design patterns, including the Model-View-Controller (MVC) architecture. While AI provided guidance in locating useful resources, all external references were verified for accuracy before being implemented.

Given the complexity of structuring a formal project report, AI was employed to assist in formatting and organizing the document. It played a key role in structuring the report according to standard academic formatting, ensuring proper LaTeX styling for tables, figures, citations, and complex document elements. AI was particularly valuable in generating LaTeX-compatible UML diagrams and sequence diagrams, enhancing readability and coherence by refining explanations and transitions between sections. By leveraging AI's proficiency in LaTeX, the report benefited from well-formatted CRC card tables, structured sequence diagrams, and neatly aligned code snippets.

All AI-assisted content was critically reviewed and refined before inclusion in the final submission. AI was not used for direct code generation but rather as an advisory tool to streamline development, improve efficiency, and enhance documentation. The final report reflects the project team's understanding and manual adjustments to AI-assisted outputs, ensuring originality and academic integrity. The integration of AI tools into the project workflow significantly improved efficiency in debugging, research, report structuring, and LaTeX formatting. However, AI was used strictly as a consultative tool rather than as a replacement for critical thinking, problem-solving, or implementation. The final product represents the collective efforts of the team, with AI serving as a supplementary resource for enhancing productivity and documentation quality.

Chapter 12

Discussion

Development proceeded with sporadic adherence to the Waterfall methodology. Early design documents, such as CRC cards, did not entirely capture the eventual client-server separation that was decided upon. Challenges included dealing with Java sockets and concurrency, establishing database connections through a connection pool, and integrating JavaFX while maintaining responsive performance. Some team members encountered scheduling or experience barriers, creating an uneven distribution of workload. As a result, the project's leadership role became more concentrated among a few contributors who managed both conceptual design and a significant portion of the coding.

The major difficulties ranged from merging Git branches without causing code conflicts, to ensuring correct synchronization among threads that handled network input, broadcast notifications, and the JavaFX UI. In addition, storing passwords in plaintext, and transmitting them unencrypted, stands out as a security risk. While the project scope did not prioritize advanced security measures, future improvements would address these concerns. Nevertheless, the completed application meets essential requirements, and the experience provided insights into concurrency, database design, and user interface implementation.

Chapter 13

Conclusion

The chat application project demonstrates how a client-server system can be built in Java, employing both concurrency and a GUI framework. Despite challenges that arose from limited collaboration among team members, the final solution supports real-time text and image messaging, user authentication, and persistent data storage with PostgreSQL. The project's evolution reveals the importance of flexible design, thorough testing, and effective communication within a team. Although certain limitations remain—particularly around security and feature depth—the application serves as a functional foundation that could be extended or refined in future iterations.

Chapter 14

Future Improvements and Enhancements

While the basic requirements are fulfilled, several significant enhancements are anticipated. One key priority involves improving security, such as adding encryption for client-server communications and adopting password hashing and salting. Another improvement would be validating user input more rigorously, especially for email addresses, usernames, and passwords. Introducing advanced chat features, including private or invite-only rooms, message replies, or an advanced user roles system, could further enrich the user experience. Better packaging for distribution, such as generating .jar files or Docker images, may also prove useful in making the application accessible to a broader audience without requiring in-depth knowledge of Maven or Git.

Additional sample code, such as the `ClientReceiver` that runs on the client side to handle background communication, provides a complementary view of how messages flow from the server to the user interface in real time.

Appendix A

Additional Resources

Repository

You can visit our repository at:
https://github.com/Zakoroo/Project-Group-6