TEAM HANOI

MSc Software Workshop

19th of March 2018

Adam Robinson 1854199

Danyal Bajar 1855229

Lee Fletcher 1460836

Tom Creaven 1186564

Joshua Anderson 1804880

Table of Contents

Introduction

For this project, Team Hanoi decided to develop an instant messenger program 'Oi Oi Hanoi' based on client-server architecture. The program was built on the framework outlined in the assessment criteria, and is client-server in nature, meaning messages are passed to a central server, and then redistributed out to individual users.

Though sending messages from a client side program to a central server is relatively simple in concept, the team sought to challenge itself and build a scalable system resembling the GUI based messaging services prominent from the early 90s – mid 2000s. This meant incorporating features such as group conversation, statuses, notification of messages sent to offline users, avatars etc. In order to achieve this implementation a sophisticated database for storing relevant user information and 'conversations' in which they participate was necessary, to enable users to participate in both multiple conversations and group messaging. Furthermore the server developed to handle concurrent messaging in disparate locations was designed to send objects thereby transferring more information than primitive types, through efficient code encapsulation, increasing the extensibility of the system. The team challenged itself further by making use of Java FX, with members investing time in order to produce a more attractive Graphical User Interface (GUI) to a more modern and professional standard.

Brief Description

Oi Oi Hanoi's main objective as a messenger application is the sending and receiving of user messages via a centralized server from separate locations.

When the application is launched the user will be presented with a 'log-in' screen (fig.1a). The user will be connected to the server via the desired hostname and port number upon selecting to login or sign up. On selecting the sign up button the user will be directed to the to a new 'sign-up' screen (fig.1b). Feedback will be displayed on the login screen if a connection cannot be established.

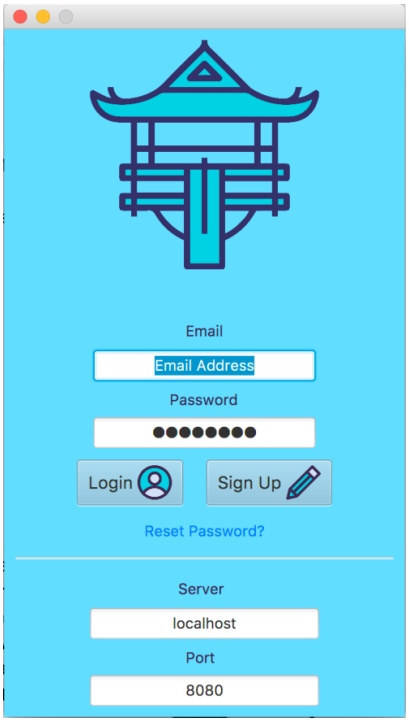


Figure 1a. Login Screen

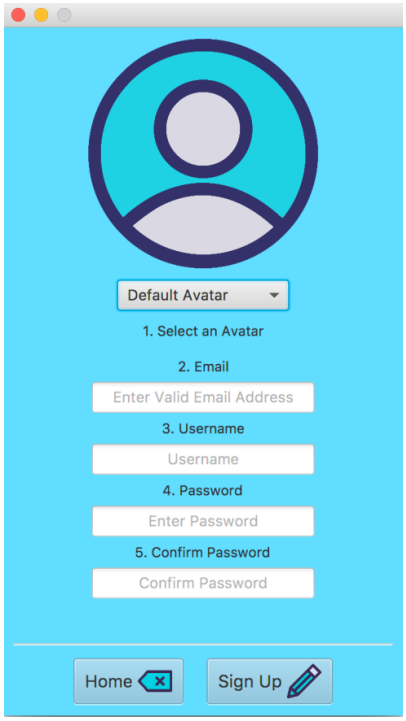


Figure 1b. Sign Up Screen

Sign-up requires the user to enter a unique username, a unique email, a password (double checked), and select an avatar from a default selection. This information is then securely stored in the database. Passwords are stored as hashed values to improve the security of the system. Logging in is achieved through entering the unique email and password of a previously registered user present in the database. Upon sign-up/log-in users can enter the main screen (fig 2),

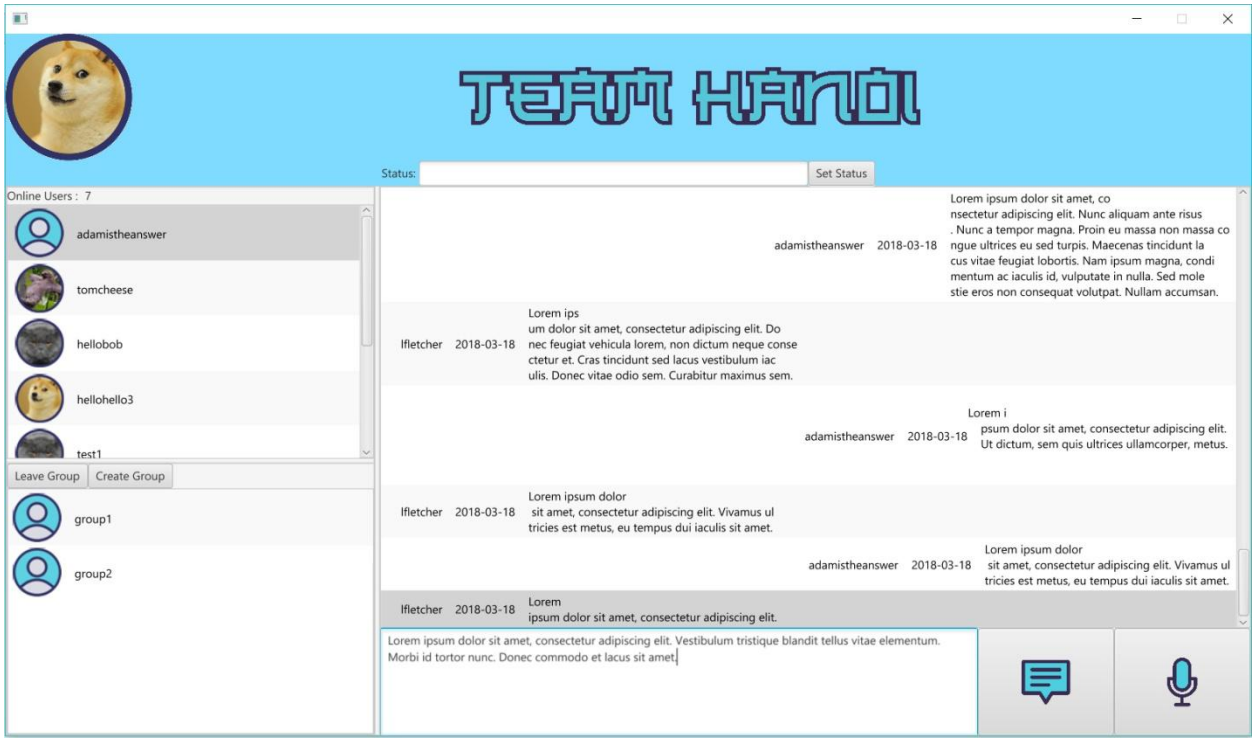


Figure 2. Main Chat Screen

and are met with a list of active online users who are also registered to the system, and notifications of any conversations of which they are a part which have been updated (i.e. new messages have been sent on said conversation) since the user was last online.

In order to initiate a conversation users simply need to click on one of the active users in the list (ordered in descending order from most frequently contact), the system will then display any previous conversation between the two users (pulled from the database. If a user wishes to initiate a group conversation they select ‘create group’ on the main screen and then click on the users they wish to add from the online list (or search for offline users in search box by their username) and confirm.

When a user has selected a conversation all chat history with the participants of that particular conversation will be retrieved (if present) and displayed to the user and new messages can be sent by the user via a text entry field. Messages are stored in the database, and updates to conversations will result in participants being pushed notifications.

When the application window is closed the user is notified that they are being signed out, which will result in the client being gracefully disconnected and removed from the server. (Fig 3)

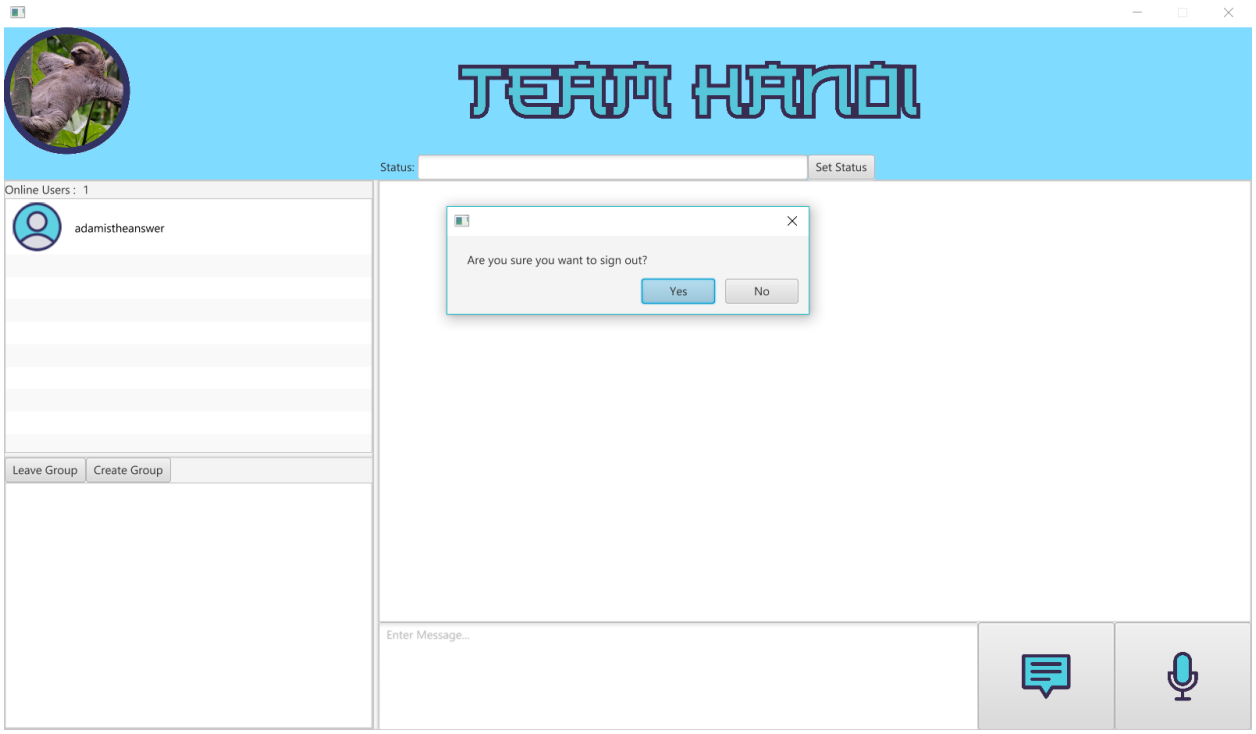


Figure 3. Sign out of the system

In the event a user needs to reset their password, this can be achieved by pressing the reset password link on the log in screen. From here the user is taken to a reset password window where a request can be sent. This replaces their password in the database with a random sting. This replaced string is then forwarded to the user via the email they used on signup so they can regain entry to the system. The team also decided to produce a lightweight GUI from which the server for our application can be initialized by administrators to a desired IP address and port number.

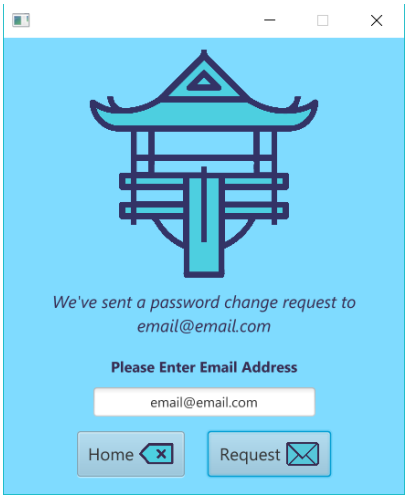


Figure 4. Reset password

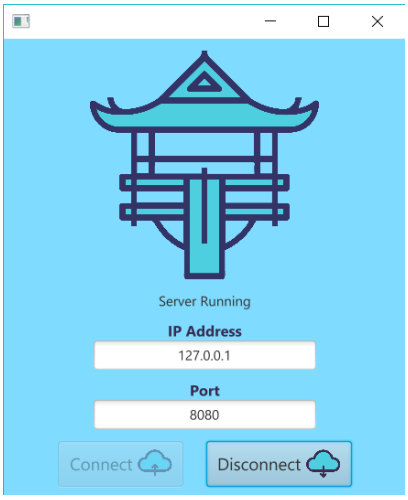


Figure 5. Server GUI

Project & Team Management Overview

Team planning

Following group allocation the team quickly arranged a meeting to discuss the work and potential development for the project. From the outset it was clear the team was unanimously in favor of developing a client server messaging application. This was attributable to its practical applications, the variety of configurations possible from a GUI and database standpoint and the technical challenge posed from a client-server perspective.

During this initial meeting the team took the opportunity to assess one another's levels of expertise, previous studies/occupation and fields of interest.

Adam's previous studies and work in graphic design were driving factors in the decision to task him with the GUI of the project. Tom was also interested in working on the GUI, having already felt proficient in SQL and instead choosing to challenge himself in an aspect of coding in which he had little experience.

The group allocated work on the databases to Danyal, primarily due to his management background and logical process with regards to allocated work, lending himself to SQL and database organisation. His previous studies (PPE) had reliance on critical review and reading which also made him the natural candidate to assume a role in producing the report.

Joshua was primarily interested in the client-server nature of the project and in this was joined by Lee whose extensive software development experience made him the obvious choice for the group to work on what was considered the most complex part of the program.

Furthermore Lee worked in a synergistic capacity on both the database and GUI area of the project, which was invaluable in producing the finished program in time.

Thus the responsibility breakdown was as follows:

Responsibility	Team member(s) assigned task
Client - Server	Lee and Joshua
GUI	Tom and Adam
Database	Danyal and Lee
Minutes	Joshua
Report	Danyal, Tom and Adam

Further Considerations made with regards to Team planning.

- During the project, it was imperative that all members of the group understood the core functionality of all subsections within the project. This ensured there was understanding of what each sections of code must provide/receive to function and also guarded against redundancy should a member be unavailable for any reason.

- It was determined from the outset that each subsection of the project would have a pair of team members that would specialise in their assigned section of the project. This meant team members always had a point of contact should they run into complications, and that development wasn't unnecessarily impaired by a team member being unavailable. This also allowed for productive paired programming sessions throughout the project.

- In the event that any member of the team was unable to complete their allocated tasks by the set deadline, it was agreed that communication would be made as soon as possible by the absent member with the task that needed to be completed.

- Team communication was primarily conducted via Facebook messenger with team members remaining in a state of constant contact. All relevant changes and issues to the program were reports as soon as they arose.

- Git (git.cs.bham.ac.uk) version control was also used within the project to manage source code and it was agreed that any iterations of code being pushed, needed to be documented sufficiently so that it was clear exactly what changes were made to the code base.

- The team agreed to spend at least 16 hours a week working together in the earlier stages of the project and increase the weekly meeting time to 20 hours in the final 2 weeks. This provided us with enough time to concentrate on the project but still allocate enough time to other modules, which fed back to the projects development.

Project timeline

Following the group decision to move forward with a chat messenger app, we decided upon a project timeline; allocating what we felt was appropriate time slots for the key features we believed the application to require. We then created a GNATT (fig 6.) chart to enable us to better visualise the development process and as a way to gauge progress. Given the limited time span for the project, other module deadlines and absences, we were not able to keep exactly to this; however the order of development remained consistent.

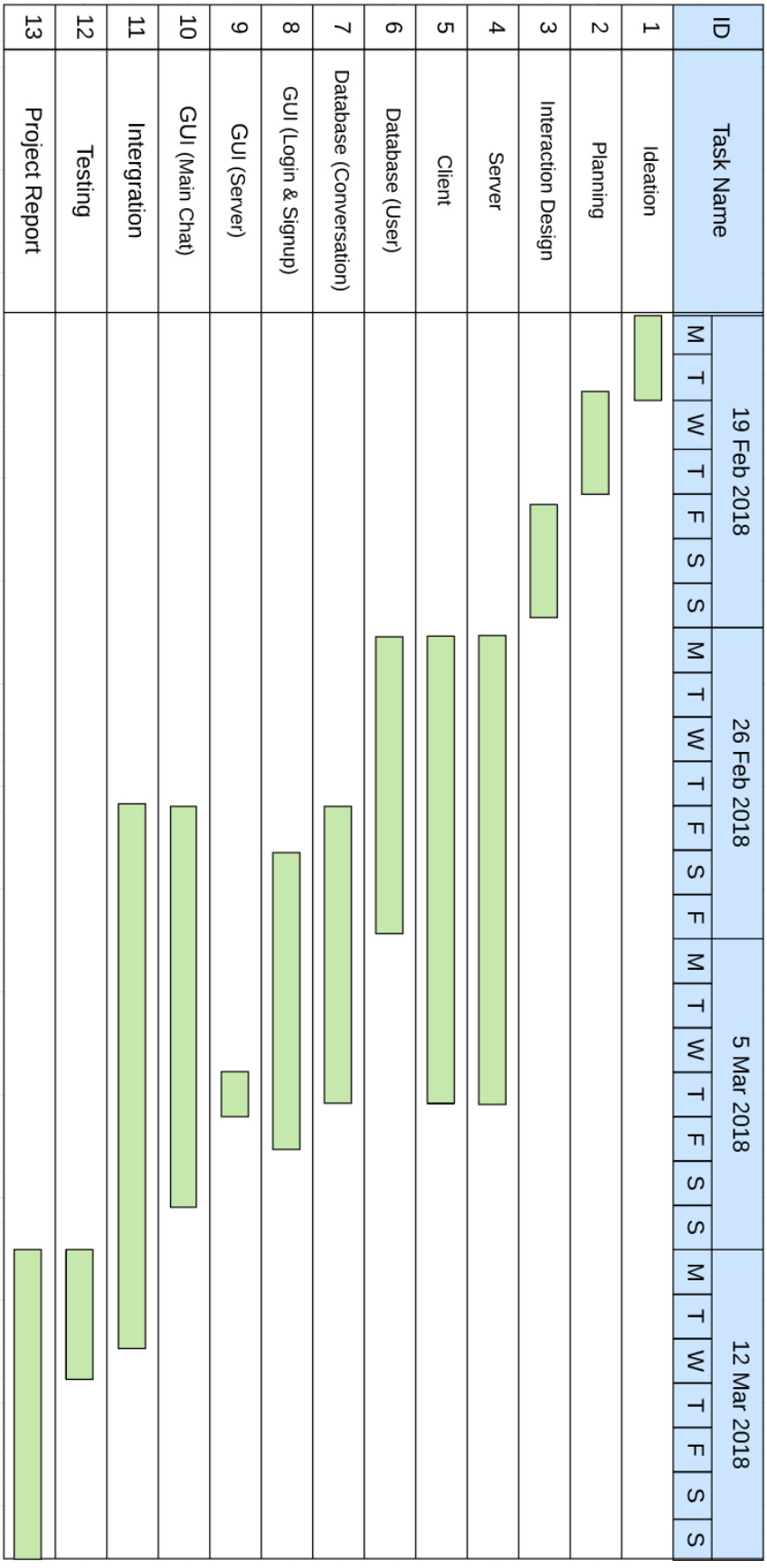


Figure 6. Team Gantt Chart

Design Decisions

GUI Research

For the messenger application, the graphical user interface is the most important component for user interaction and usability. A well-designed GUI should enable seamless interaction between the user and the business logic of the program.

To this end, the team decided to research the best methodology to design, and implement the user interaction side of the application.

The team intended to follow the model-view-controller separation paradigm for this project, and during this course we have been introduced to java Swing for graphical user interface development in java. Swing is a successor to java's initial AWT interface, and provides a lightweight, non-platform specific implementation. Swing is entirely component based and so provides a large amount of flexibility when designing interfaces, and since it follows the MVC paradigm, the UI would be easily modifiable without excessive changes to the business logic of the code. Swing has also been heavily used by developers, and as such has a lot of resources available on the web, so bug fixing and development queries are much easier.

However, while Swing has been the UI of choice for java development over the years, it is not without its faults, and as of Java 8, has been phased out in favor of JavaFX, Java's newer GUI implementation. In fact, as of the latest release of JDK, swing is now deprecated, and will no longer be updated in any way.

Java FX has many benefits over swing, such as using FXML files to construct the views, rather than raw java, and allows for the integration of CSS styling for easy customisability. FXML is an XML derivative that will enable us to create the view of the GUI entirely separate from the application logic. This would enable us to maintain a more dynamic and layered UI.

Oracle also provides a tool for JavaFX called scene builder. This is WYSIWYG editor used to more easily build UI components into a JavaFX scene; this automatically generates FXML code and can be integrated into in most IDEs, including Eclipse and IntelliJ. JavaFX has in-built features for adding custom animations, if we wish to expand the GUI features. JavaFX also provides support for modern touch devices, something that swing lacks entirely, which would allow us to more easily develop a mobile based version of the application if we chose to do so.

Process Model

For the development of the application we took an iterative design approach. This allowed us to add extra features throughout the project when core components such as messaging and user validation were signed off. This ensured that we had a minimum viable product produced within the project timeline. This also guarded against spreading the development team too thinly across the project.

Use Case Diagram

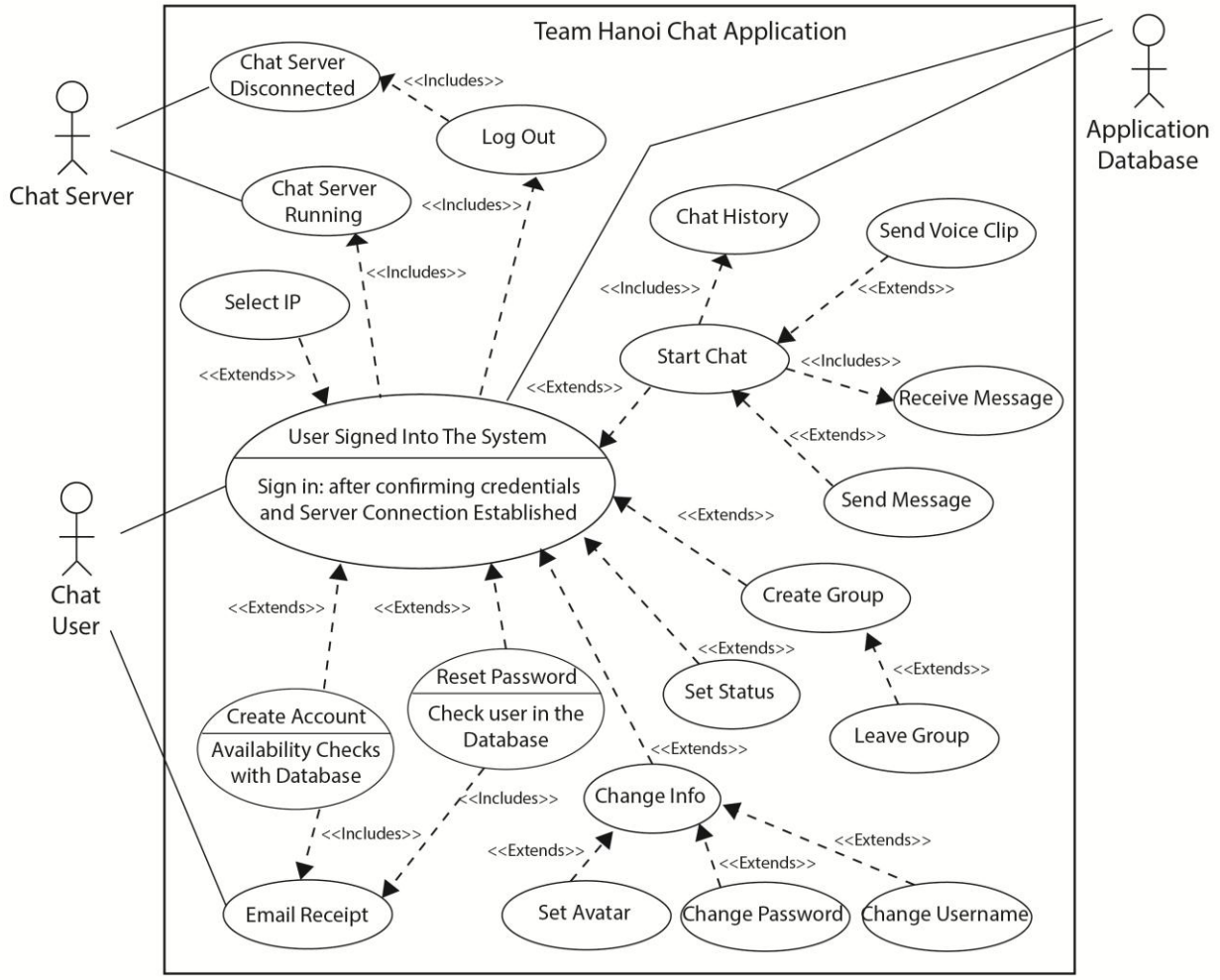


Figure 7. Use Case Diagram

To increase the visibility of the planned features for the application a use case diagram (fig 7.) was produced. This allowed for better understanding of the interaction for the planned systems features and their dependencies, from the perspective of external actors to the system. The use of extend annotation allowed the team to prioritise development as extends use cases can in some areas be defined as additional functionality. Here the desired features for the application and the stretch goals for the project were defined.

System Requirements

Assumptions:

- The system will have a keyboard input.
- The user will have an email address for registration and password recovery purposes.
- The system will have access to a database/server

Functional:

Architecture:

- The system must accept keyboard input.
- The system must allow user (clients) to concurrently interact with a single server.
- The system must inform users if server cannot be reached.
- The system must inform user if the database cannot be reached.

Registration/login:

- The system must ensure user has entered valid email address, username, and password prior to completion of registration.
    - A username must be between 5-20 characters.
    - Passwords must be between 8 – 20 characters.
    - Email addresses must be a valid email address following standard conventions.
- The system must ensure that usernames and email address are unique upon sign up.
- The system must authenticate users by email-address and password in order to login
- The system must differentiate users by login credentials.
- The system should enable user to reset password if forgotten via recovery service, utilising email provided on registration.
- The system should send the user an email receipt upon successful registration containing, registration details.
- The system could contain remember me functionality for faster login.

Termination:

- The system must have log-out functionality.
- The system must terminate instances of the program when a user closes the program.
- The system must gracefully handle lost connections from a given client.

User Profiles:

- The system should allow a user to set a status.
- The system could allow user to view their most frequently contacted list.
- The system could allow user to store avatar/profile picture.
- The system could incorporate an award system based on user usage, e.g. trophies for frequent messaging/activity.

Chat:

- The system must maintain chat logs of users.
- The system must enable concurrent messaging to multiple users.
- The system must push new messages to clients.
- The system must indicate users currently active/online with whom, a user can message.

- The system should allow for group chats with multiple users at once.
- The system should ensure users can be added to group chats.
- The system must ensure only users in a chat can view their shared message log.
- The system should allow users to be part of multiple group chats.
- The system could allow user to message all other users of the server.
- The system could allow user to search through users of the server to message.
- The system must notify users if a message has been received from another user
- The system could display a pop-up notification when a new message is received
- The system could allow users to delete their message logs.
- The system could send messages to offline users as emails to the account associated with their profile.
- The system could keep a record of the last 30 messages to be handled by the server in the case of server failure
- The system could display the currently active/online users by most frequently contacted.
- The system could implement lazy loading on chat logs.
- The system could support the sending of multimedia files across chats.
- The system could suggest a selection of quick messages e.g. "Hi there!", "How are you today?" etc.
- The system could support the use of emoji.
- The system could make a sound effect upon a message being received.
- The system could make sound effects mutable.
- The system could let users block other users from messaging them.
- The system could notify group chat members if a group member leaves the chat
- The system could notify a user currently in a chat if another user in the chat is typing a message at present

System:

- The system should encrypt user profile information for storage in the database.

Non-functional:

Usability/Ease of use:

- The system must have a graphical user interface with suitable signposting.
- The system must have consistency between user profile screens and chat logs.
- The system should be straightforward for user. Within 5 minutes user should have a basic understanding of the system. Following an hour of usage, errors per hour should be less than 1.

Efficiency:

- The system should not have excess latency - time between a chat message being sent by one user and received by recipient(s) should be no more than 2 seconds.
- The system must support multiple active chats at a time without a decreased in performance.
- The system should have capacity to handle increases in system load during peak usage.

Availability:

- The system should be available 24/7.

Security:

- The system will not permit a user access to any functionality without authentication.
- The system will allow user access to their profile and chat logs following one level of authentication.
- The system will guard against unwarranted disclosure of user information to third parties.
- The system should hash all user information and messages in the database.
- The system must only allow the participants in a conversation to view it's content

Maintainability:

- The system must be coded in an efficient manner and well documented in order to facilitate future maintenance and scaling with minimal complications/debugging required.

Implementation:

- The system should be coded in Java where possible making use of object - oriented programming techniques.

Scalability

- The system could allow the user remote access via smartphone, tablet or computer.

Ethical:

- By storing user information we take on a responsibility that it not be accessible to third parties, either through unwarranted distribution or negligence in storage.

Analysis of tools

InteliJ

The team decided to use the same IDE for the project to alleviate potential file conflicts throughout the project. InteliJ was selected as it offered ease of integration with the universities gitlab repository, along with other team members being familiar with its suite of advanced functionality. This allowed the team members to help each other if issues were to arise, especially in areas such as version control which was a new area for most team members.

Scene builder

With consistent use of InteliJ the team could all integrate Oracles scenebuilder into their IDEs. This allowed for easy recognition of areas of the GUI related to a code area which needed changing.

GIT Version Control (Gitlab)

Lee was familiar with Git from his previous role in software development. As version control was a new area for the other members of the team it was decided that it would be wise to capitalise on his experience. This came in particularly useful throughout the project when issues arose in the code base. Allowing Lee to revert commits back to a stable state. Both tutors were invited to the Git repository on creation.

DataGrip

DataGrip was used to visually interface with the database tables. This offered an efficient solution to quickly query the database to ensure that table structure was as intended and that data was being parsed as planned. During the trial and error implementation of password hashing this was very useful.

Photoshop & Illustrator

Adams prior work in graphic design allowed for small consistently styled UI elements to be generated in the form of button graphics, avatars, banners and an application logo.

System Design

Overview

This system as a whole has the primary task of connecting users and allowing them to converse in real time conversation through a network connection. This necessitated sign-up, log-in, sending / receiving messages and group conversation functionality.

The server is central to the infrastructure of the system as demonstrated in the deployment diagram (fig 8.), sitting at the nexus of the client threads interactions and handling queries passed to the database. Whilst the server handles all of business logic for the program, an assisting client thread class (which would act as a client server) handles the logic of each individual client.

When a new client connects to the server, the server utilises threads from a thread pool which instantiate threads for each client. The newly connected user, along with their thread, is initialised and stored inside a "Map" data structure within server class. The newly connected user's interactions are handled by its respective thread.

The logic handled by the thread:

- Signing up
- Logging in
- Sending UserMessage (Objects)
- Receiving UserMessage (Objects)
- Logging out

The server handles queries to the database and passes data retrieved to the receiving client, which interprets the data which is then processed and presented on the recipient's GUI, which is listening for certain modifications on the client data model.

The deployment diagram shows how users interact with the server via the GUI which passes information via the client to the server. The server processes client thread requests, and if validated against the database, will either return the data requested or makes requested updates to data in the database and inform the other client threads (and therefore other users) connected to it.

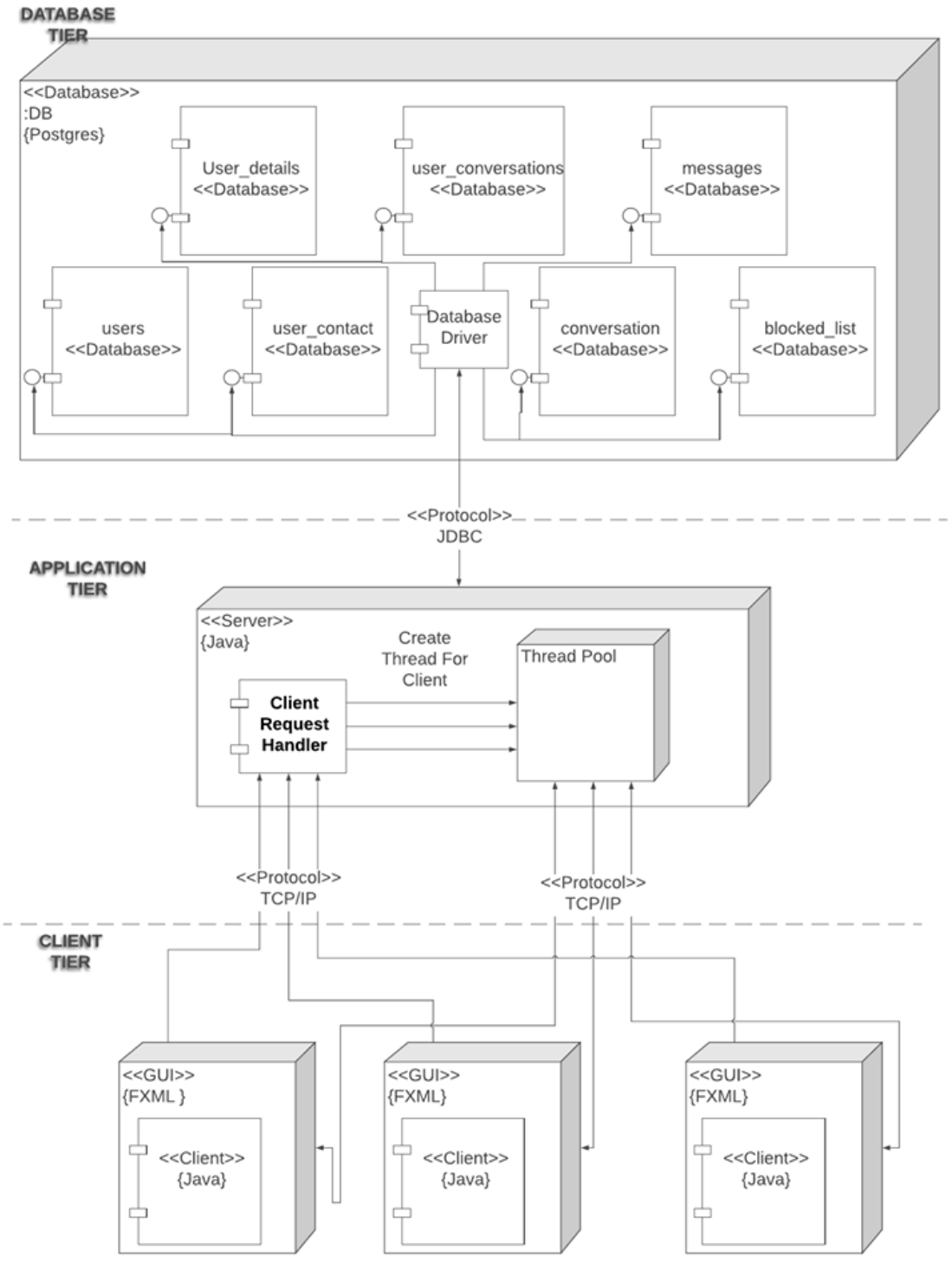


Figure 8. Deployment Diagram

Server

In an exercise to improve the usability of the application, a GUI was created via which an admin could enter a desired IP address and port number on which to set up the server. Upon calling the start() method the server creates a connection to the database and opens a server socket with the port number and IP provided. An instance of ClientRequestHandler() is then initialized, a thread in itself which is constructed with a thread pool and the server object. Owing to its nature as a thread, the ClientRequestObject() continues to listen for client requests to access this server's socket (for the duration said socket is open) for client requests. When a client connects to the server socket a client socket is created which will be used for all future communication with that client, and a new Client Thread is instantiated with the server and its socket passed via the handler. This client thread then acts as the conduit for communication between the client and the server until the client calls close(), the client thread is then terminated.

The server maintains a HashMap (the active user 'list') of the online users consisting of the client threads, which have an associated User object (the users 'profile') for a key. The server's role from this point is to listen for changes to the active clients, updating this list so as to keep all clients informed of other Users online status (via their ClientThread's). At the same time the server will be listening for 'Message'(see Message Protocol) from the ClientThread's which are currently in communication with the server. These Message's will require the database to utilise its key methods (detailed below), which constitute the core functionality of the system and require it to access the database.

start(): This method, when called, initialises the connections to the databases, creates a server socket and initialises it with the port number and IP (inputted by administrator), sets the state of the server to 'Running'. A clientRequestHandler object is then instantiated on the server socket which maintains a thread pool and once started awaits new client connection requests from clients to invoke a new client thread.

AddOnlineUser(): Upon a new client thread being created, the User object (the user's online profile so to speak) associated with that thread (entered during signIn()), and the client thread itself are added to the list of online users. All client threads will receive notification of this change, thus making newly signed in user available for contact.

SignOutUser(): Remove's the disconnecting user from the list of online users, again all client threads all receive notification of this change, making the now signed out user unavailable for contact.

SignUp(): Upon receiving an instance of SignUpMessage from a client thread, the server's signUp() method is called via the thread and first checks if the email or email are already present in the database. In instances where the email is present a response message with a string containing this information which is then written back to the client via their client thread. If the username/email is unique (indicating a new user is attempting to sign up) a series of if/else statements check the validity of the email username and then password entered (in any instances which fail, the user will be similarly informed via a response message with an appropriate string written out via the client thread). Only if all these checks are passed is the database method called to insert the user credentials provided into the database thus signing up the user, and the client informed once again via their client thread (setResponse now assigned 'true'). The associated client thread is then terminated and a welcome email sent to the email address provided by the user who just signed up.

signIn(): When a client thread makes a call to a server signIn() method the SignInMessage object's passed email is first retrieved and checked (via checkEmailLocally()) against the list of online users of the server, if present the client is informed via a response message that the user is already logged in. If not already online the SignInMessage's email and password are checked against those stored in the database. Only in instances where the credentials are found in the database are users logged in, and the list of online users on the server updated and other clients pushed the updated active user list. In instances where the credentials are not present the client is informed via a response message.

newConversation(): When called this method checks (against the database) if a conversation already exists between the participants within the StartConversationMessage object. When a conversation is present in the database, a conversation object is constructed from the data retrieved from the database and written back to the client. Else, if this is a new conversation a new conversation will be constructed and returned to the client, simultaneously creating new entries in the conversation and user_conversation tables.

newMessage(): If a user is in a conversation and wishes to send a new message, the client thread will call the server's newMessage() method with an instance of MessagingMessage. The recipient username will then be extracted from the database and passed in an argument to the database method newMessage() along with the user and message information required. This is used to update the Messages table. The conversation table will be updated by this as well and the conversation is subsequently returned, complete with the new message. Both the sender's client and recipient's client are then returned a Message instance with setResponse() reassigned to true, informing the respective listening clients of the updated conversation.

StopServer(): This methods function is to ensure that when the server is prompted to be terminated, all the incoming and outgoing data streams are closed gracefully. This method also ensures all sockets are disconnected; the clientRequestHandler is set to interrupted and its thread pool shutdown. This in turn ensures any connected client threads are terminated and the respective client is disconnected. Disconnection from the related databases is also ensured. When all these steps have been carried out, the server is then set to a state of 'Not running'.

Protocol

There are a variety of interactions between the server and the client, for this reason a protocol is required in order to provide a format for these events, in order that the server may process the request being made of it. In the application the protocol is defined in the abstract class Message. This class has a requestMessage, responseMessage and responseSuccess fields (String, String, Bool respectively), thus descendant classes have these fields and must implement methods which serve to update the client and server respectively in the way defined in Message. This ensures that the data being processed is done in a specific way to ensure uniformity, thus lessening errors. The "Messages" being used by the client threads are:

UpdateOnlineUsersMessage: This message is constructed by the server following a successful user sign-in or sign-out. The message (viewable notification) is sent to all online clients to inform them of the changed online users, the client then updates the local values of online users.

StartConversationMessage: This message is generated by a client when they are selecting an online user to begin conversation with, via the GUI. The message is sent to the server, which then uses the information present to either construct a conversation object containing all UserMessages

as well as all the participants of the conversation (each respectively in an array list) or, if no conversation has previously occurred between then a new conversation object is constructed and is sent back to the client. The client then adds this to their map of current conversations.

SignUpMessage: This message is generated by the client when completing the signup process having entered their credentials. The message is sent to the server, which then uses the information provided by the client to set up a user account, a number of checks are performed to first ensure that the sign-up request is valid i.e., A user does not already exist, with the same email address or username. Once the sign-up request has been validated the server sets a response, which is sent back to the client to be acted upon i.e., Signup is successful/unsuccessful – The user is then informed of this via the GUI.

SignInMessage: This protocols purpose is to simply enable users to sign into the application. The client creates one of these messages upon signing in, this is sent to the server and the server uses the information present in the message to validate the sign-in request. The server then sets a response to the same message, stating whether or not the sign-in was successful. This is passed back to the client, which then acts accordingly.

MessagingMessage: This protocol is used for clients to send message text to other clients. This is processed via the server.

Message: This protocol is an abstract class which forms the base for all other message types, this includes the ability to set a request type and to define a response.

Client Thread

For each client that connects to the Server a ClientThread object is instantiated - importantly each ClientThread is an executable belonging to the Server's threadPool and is designated to a single Client.

ClientThread is an extension of Thread and as such contains a run() method. The run() method operates by polling an instance of ObjectInputStream waiting for Messages (Messages as defined in the earlier defined protocol section). Upon receipt of a Message, the run() method calls a method interpretMessage() which returns the type of Message (as all Messages are a subtype of the abstract class Message). This return value is then provided as an argument to the processMessage() method which calls the appropriate Server method providing the Message as an argument (i.e. if the Client sends a SignInMessage, the processMessage() method will call the signIn() method on the server, passing the Message as an instanceOf SignInMessage).

Client

Client forms the underlying model of the client side application. An object of Client is instantiated when a user selects sign in or sign up from the GUI. The constructor of Client is passed the IP address and port number as defined via the GUI which allows the user to point at a running server. The Client constructor then uses these values to establish a connection with the Server, as detailed previously, the Server then opens a socket for all future communication with the given Client, and instantiates a ClientThread to manage their workload. At this point the Client now has a socket available to send future protocol messages via. Finally the Client constructer instantiates an ObjectOuputStream and ObjectInputStream using the aforementioned socket to allow for sending / receiving of messages to / from the server.

It is also worth noting that for a real life implementation of this system the host and port values would not be user selectable, instead they would be managed by a system administrator responsible for all clients of a given implementation. It made sense to keep these selectable within this current version as it allows for easy testing of servers *running at different locations. I*t is also worth noting th*is was* at we have the ability to manage the system in the manner described above (i.e. via a system administrator managing all clients) so the system could be implemented in this manner with no addition effort required*, making the system more extensible*.

Once the client has been instantiated, the relevant GUI controller (LogInController or SignUpController) constructs a Message (again, Message as defined in the protocol section of this document) and sends this to the Server using the ObjectOutputStream created during instantiation of the Client. Upon receipt of the Servers response, the Client is able to take appropriate action (i.e. successful sign up, user is forwarded to Chat view, Client variables are updated to reflect information provided by the Server).

The Client contains all of the data required for the end user to interact meaningfully with the application. Importantly the Client maintains two key data structures - an ArrayList<User> onlineUsers, and a HashMap<String, Conversation>. These are used by the GUI controllers to present the end user with appropriate information and are kept up to date via pushes from the Server.

Each Client utilises a single thread as its only source of input, giving that the thread can only perform one function at a time ensuring consistency in the data is maintained. When the ClientThread calls the Server methods that impact key data structures (i.e. Map of onlineUsers etc) those methods are synchronized meaning those global variables maintained at Server level are protected from race conditions in terms of deadlocks.

We have implemented model-view separation by using listener interfaces which are positioned within the Client model at relevant points to 'listen' for changes that the view needs to act upon. The ChatController adds these Listeners setting appropriate method calls at instantiation. For example, the ConversationListener is defined to call setChatView() when a change occurs to the users Map of conversations. setChatView() will then update the users display as appropriate (refresh the chat window if the sender matches the currently selected user, create a notification if not).

Client contains the following methods:

updateOnlineUsers(): Updates the Clients local list of onlineUsers() when a push is received from the server. A push will either consist of a user sign in, or user log out and this allows the user to always be provided with an up to date list of online users. UserListeners are positioned within the method to ensure all listeners are updated to the change (i.e. the ChatController can reflect the changes to *the data model within the view)*

updateConversationsList(): Updates the Clients local Map of conversations when a push is received from the server. A push will either have been instigated by the user in question (i.e. by selecting an online user) or by another user selecting them. This increases efficiency as in the second scenario, the user will already have their Conversation stored locally if they then want to interact with the other user (avoids a second Message being processed by the server and the relevant database calls etc). In the first scenario a Listener is positioned so that the users view can be updated with *the Conversation as requested.*

addMessageToConversation(): Updates the Clients local Map of conversations by adding the received UserMessage to the relevant Conversation. Listeners are placed here to inform the ChatController of the changes. The ChatController can then react as required (i.e. if the user was the sender, update their view of the chat, if the user was the receiver, provide a notification).

Database

Design

The database serves as a persistent source of data which is accessed via the server. Following the relational model, data was collected into entities and the relationships between them in order to facilitate the extraction of information required through queries. These relations are captured in the Entity Relationship diagram (fig.) which was produced during the planning stage of the project. This served to delineate table’s responsibilities, highlight dependencies and aided communication between the sub teams dedicated to client - server and database respectively in producing methods to return the information requested.

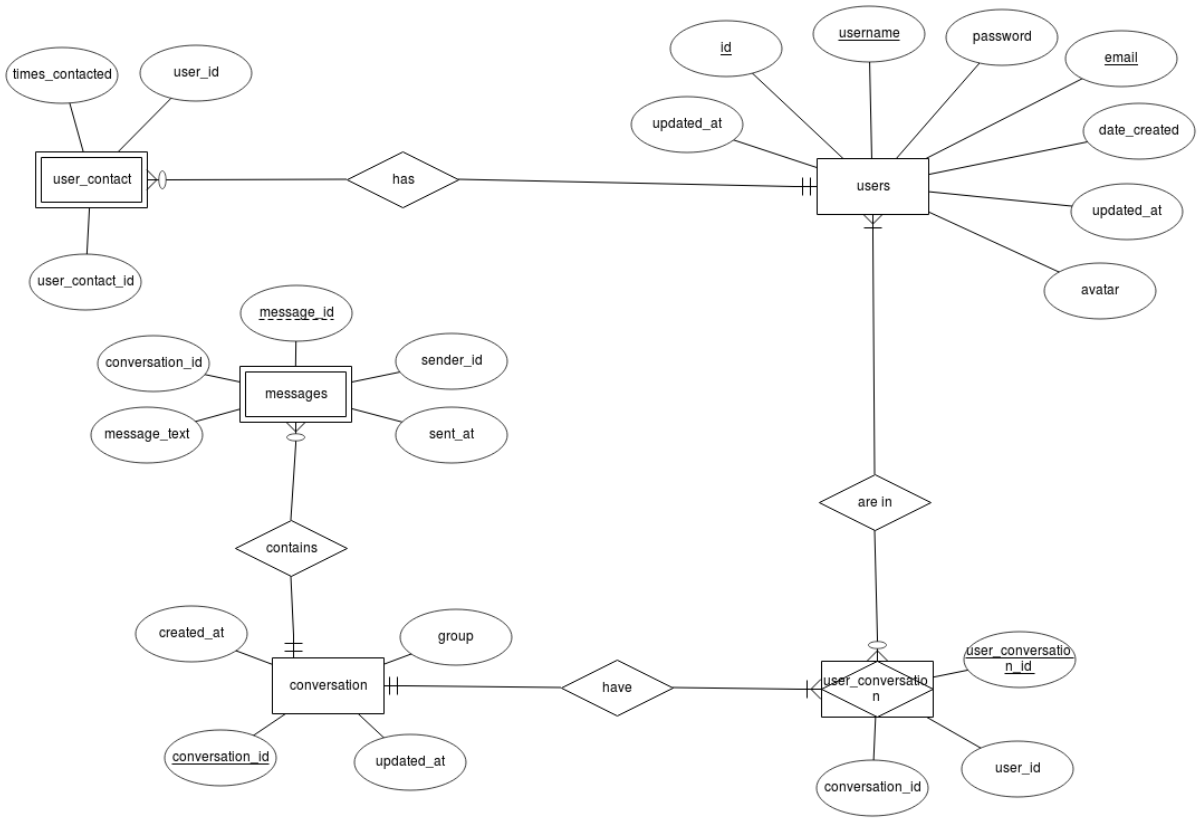


Figure 9. Entity Relationship Diagram

As a messenger application, database design was centered upon providing the users core functionalities: user information (for log-in and registration) and messages. Given efficient database design requires the minimising of redundancy and the encapsulation of distinct entities, user information was split further into users (core user information) and user_contact (stores number of interactions with a given contact).

Furthermore as messages need to be attributed to both a sender and at least one other user, a conversation table was made to log between whom the messages were being sent. This was also necessitated due to the many to one relationship between message and conversation (that is a given conversation can have many messages attributable to it, thus message must have a foreign key to a conversation primary key).

As there is a many-to-many relationship between users and conversations, that is one user can be part of many conversations and conversations can have many users, a further associative table (user_conversation) was created containing foreign keys to the id's of both the users and conversation tables.

In this way the database was left with six tables; however through normalising the table (distinguishing entities responsibilities, ensuring tables had unique id's) scope was left for extending the database further, ensuring system scalability. For instance if time had allowed, a blocked list could have been incorporated (utilising an associative table user_blocked) enabling users to block other users they don't wish to be contacted by.

Methodology

In order to access the database through the program the systems server required importing the Java Database Connectivity (JDBC) API to classes within the server package, which defines how a client can access a database. The database properties were defined in a properties folder and included the driver and URL to the database's location, and the credentials required in order to access it.

A database class was created within the server package to handle the connection and communication between the server and the database. Within it, methods were defined for the construction of a connection to the database (connectToDatabase() method) and methods for handling the retrieving and setting of data within the tables outlined.

Within connectToDatabase(), the database properties are loaded in as a property object via a file input stream, and assigned to the relevant properties : url, driver, user and password. The driver is then assigned, before a connection object is instantiated through a call to the DriverManager class's static method getConnection(url, user, password) and the connection (if not null) is returned.

As mentioned upon server setup this is the first method executed within server's start() method in order to instantiate a connection object through which the server's connection and subsequent interaction to the database can be regulated. As establishing the connection can be slow, and the database is required on a continual basis, it is left open until the server is closed.

The database methods the sub team designed which utilise the JDBC, all take the connection created as arguments, in order to access the database. All of the team's methods make use of Prepared Statements, an interface which represents an SQL statement, which are created on the

Connection object passed. These statements take SQL queries in a string format which are passed in the objects constructor. Prepared Statements were chosen by the database sub team over standard Statement objects in that they are given the SQL statement on creation, which is advantageous in that SQL statements in Prepared Statements are precompiled, reducing execution time. Furthermore they protect against SQL injection because they provide the SQL statement on object creation, and the parameters passed are then treated as data as opposed to part of the query (e.g. typing in drop users etc), thus avoiding risk of a user passing harmful input to the database. Finally they helped make clearer the distinction between the query code and the parameter values passed thereby improving readability, enabling the sub team to quickly ascertain the inputs and outputs of a given query. The actual parameters of the search's are passed through set methods and results are returned in resultSet objects which can be iterated through on a row by row basis using next(). All methods were embedded within try-catch-finally blocks in order to ensure that in the event of an SQLException it would be handled, and the prepared statement closed.

When possible methods only enacted a single statement at a time thus ensuring autoCommit could remain set to true and that queries would be treated as a transaction. This reduced likelihood that an error on the part of the sub team in handling commits would result in data corruption and harm table integrity.

Methods

The application's server is in constant communication with the database, and is required for all client functionalities. Registration requires the user database to check to see if a users desired username is already present (if so reporting this to the server and onto the client), and for storing new user information in instances where it is not.

Log in similarly necessitates a check to the users table to see if the credentials entered match what is stored for a given user. Upon log-in the database is required to return the information to construct the relevant users user object, the list of active users (these user objects are continuously updated for local access by the client), as well as returning conversations (and the associated messages) the user is present in (thereby utilising both conversation and user_conversation tables) which have been updated since the user was last online (GUI displays appropriate notification).

Selecting users from the active list to contact, results in a database call to check if there was previous conversation between the two users (via the user_conversation table). if there was then a call is made to the messages table returning the information required to construct userMessage objects in an arraylist which is associated with the relevant conversation participants (user objects; constructed from data pulled from users table) within a conversation object. If not constructing a new conversation object without any messages. In this manner group objects are also created with multiple users associated with a conversation, and each requiring access to the messages associated with that conversation.

Sending a message requires the database to access both the message table and the conversation table, in order that the relevant participants may have access to, and notified, about the new message.

Graphical User Interface

GUI development was not an area of familiarity for the team and problems were encountered in the early phases of the design. However when familiarity with the library, and core principles of scene changing were established, the team was able to produce GUIs with a professional appearance with dynamic functionality.

The figure below demonstrates a high level overview of the of the GUI (JavaFX scene) navigation.

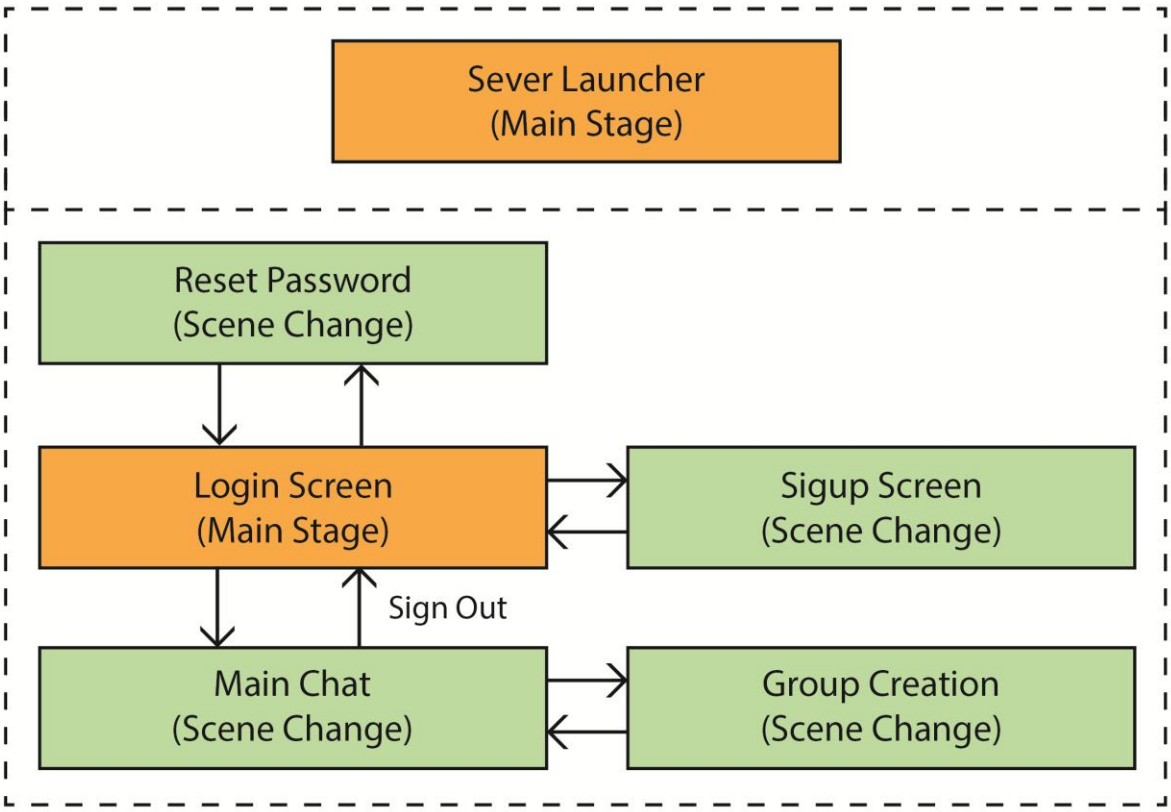


Figure 10. High level GUI navigation

Class & FXML	Purpose
ServerLauncher.class (FX Launcher) Server.fxml (View) ServerController.class (Controller)	Implemented as a primary stage in JavaFX. Allows for the creation of a standalone application to be ran on a web server that is independent of the Main chat app. GUI only allows for one instance of server to be started per instance of Server Launcher. Only contains the server.fxml scene. Server IP and port can be set at this scene and passed as arguments to server main.
Launcher.class (FX Launcher)	Creates the instance of the JavaFX application. FXML Loader sets main stage to Login.fxml. Dimension Constraints of (350 x 650px)

Login.fxml (View) LoginController.class (Controller)	On FX application start Login.fxml is set as main stage. From here users can select IP & port of the server (This can easily be removed when a more permanent server is deployed). Text Areas for email and password protected by REGEX to only allow valid user input. Information Label to give connection feedback. Three navigation options to reset password, login & sign-up. Each navigation replaces main stage with corresponding scene, keeping instance of application active. User thread and connection created on navigation from this screen.
Signup.fxml (View) SignUpController.class (Controller)	Signup.fxml new scene loaded to primary stage, dimension constraints of (350 x 650px). User thread and connection established during scene change based on login screen IP & port field. User can select avatar (choice box), email (text area), username (text area) and password (text area x2). Email, username, and password fields must comply with REGEX check. First layer of error handling, this is also checked by the server. Feedback displayed in label for server response and availability. Client thread terminated on successful signup. Signup becomes unavailable after success to guard against multiple entries to the database. Navigation to login resets primary stage to login.fxml
ForgotPassword.fxml (View) ForgotPasswordController.class (Controller)	Forgotpassword.fxml new scene loaded to primary stage. Text within field for email address sent to server on request and password replaced. Email handler then sends reset password to user via their signed up email. Back button returns user to login screen via scene change.
Mainchat.fxml (View) ChatController.class (Controller)	Mainchat.fxml new scene loaded to primary stage, dimension constraints of (1280 x 720px). To ensure graceful disconnection and user thread termination the GUI overrides close window action with an event consume terminating the user thread and then returning the user to the login screen so they may sign in again or with a different user. This helps protect the concurrency of the system by ensuring they leave the server on window close. The server also guards against multiple instances of the same user being present on the system. User initiates chat by clicking on the listviews of online users or groups. Controller stores updates field of last selected user. When an event is fired based on server response (such as new message) the users chat listview is updated if the corresponding conversation is active. If a new message from an unselected user is received the GUI notifies the user with an alert icon and the user may view the received message upon selecting that user from the online user list. User avatar set at top left of screen based as signup selection. Two message send buttons, one for text and one for voice. GUI guards against empty messages. Voice button hold down for record and send. User may set their status via text field at top which is displayed as a label on the online user list. Messages displayed in listview with word wrapping. Message

	cells constructed with information such as name and sent time. Create group button loads new scene of group.fxml.
group.fxml (View) GroupController.class (Controller)	group.fxml new scene loaded to primary stage. User can multi select from list of users registered with system to add to a group and create a group via create group button. Once group created user can return to mainchat via scene change. Group will be displayed on group list on main chat.

The team appreciated the importance of developing an application that made use of the model view separation paradigm. This was achieved this as follows; the server sends communications to the client listener class as message objects (the devised protocol) here they are interpreted using instanceof to determine which subclass of the message class the client has received. (example, updateOnlineUserMessage for informing the client of a change to users present on the system). Upon interpreting the protocol the correct action is fired to the client class. The client class acts as the model in the environment storing information client side in memory. When changes occur in the client's model, events are triggered in the form of method calls to affect the users view via the appropriate controller.

Evaluation

In order to maintain the integrity of the application, various forms of testing were undertaken during the duration of the project. The testing was split up into two main sections, pre-integration testing, and post integration testing.

Pre-Integration testing:

While individual components of the system were being developed, they were tested using various methods to ensure their viability prior to full system integration.

The GUI was tested by creating an array list of pre-made user objects, and an array of messages, stored locally. When the GUI was launched this would populate the main chat view with mock online users. Such methods allowed for the testing of individual GUI components without the need for end to end connection with either the server, or other clients. Simple test functions were created inside the GUI controllers, to perform simple actions such as printing certain statements upon mouse clicks and button presses, in order to test basic functionality of the GUI. The avatar selection was set to an integer, and printed to the terminal upon change, in order to confirm its status. The GUI was loaded on multiple different machines, to test that the look and feel of the views was consistent on different sized screens, with different resolutions. All different windows were tested before the main integration took place.

The client server interactions were tested by developing a simple console based program, which took arguments and allowed for a simple, cohesive understanding of basic client server interactions before integration with the database, and GUI elements of the system. Junit testing was used to check the results of objects passed between different components of the system, which can be seen in the code.

In order to test the database, DataGrip IDE was used in order to directly observe database changes, and to allow for easier testing, and convenient database interaction. Simple tables were created in order to test initial methods, such as user sign-up and sign-in, as this allowed the team to observe that simple interactions between components were functioning as intended without the need for full scale development. Furthermore, this enabled the team to assess whether database methods were correctly formatted, before full scale integration. Pre-determined values were inserted into the tables, database method results could then be tested by comparing against expected result sets.

JUnit tests were created for testing set database methods; these can be seen in the code.

Post Integration Testing

Following integration of the system components into one cohesive application, full end-to-end tests were performed covering the following areas:

- Launching the server
- Sign-up
- Sign-in
- Password resets
- User interaction
- Exiting the system

In order to detect as many potential issues with the system as possible, different scenarios were documented, and their outcome recorded. Interactions and observations were tested through the GUI, attempting to cover all possible instances of user interaction and find any potential system issues.

Specific interactions are listed below:

Launching the server:

Feature	Action	Expected result	Result
Invalid server launching	Select the connect button on the server launcher, with empty IP address and port fields	Prompt user to fill fields	Pass
Invalid server launching	Select the connect button on the server launcher, with empty IP field	Prompt user to fill port field	Pass
Invalid server launching	Select the connect button on the server launcher, with empty port field	Prompt user to fill IP field	Pass
Launching server Twice	Attempt to launch the server, while server is already running on machine	Inform user that server is already running	Pass

Successful server launch	Launch server with IP - localhost. Port - 8080	Server runs, connection button becomes unelectable	Pass

Testing Login functionality:

Feature	Test Case	Expected Result	Result
Incorrect Login	User attempts to login to system, without having registered	Login Screen Displays "No user found with this username and password"	Pass
Incorrect Password	User attempts to login to system with registered email but incorrect password	Login Screen Displays "No user found with this username and password"	Pass
Empty Login fields	Presses login button whilst email and password fields are blank	Login Screen Displays "No user found with this username and password"	Fail - GUI hangs as request to database with blank fields fails. Need to update GUI code to not accept empty fields
Login incorrect server	Presses login button whilst port and IP address are empty	GUI prompts user to fill in port and IP fields	Pass
Login incorrect server	Presses login button whilst port field is empty	GUI prompts user to fill in port field	Pass
Login incorrect server	Presses login button whilst IP field is empty	GUI prompts user to fill in IP field	Pass

Login incorrect server	Presses login button with invalid IP field	GUI fails to connect to server, informs user of failure. Prompts user to check IP address and port	Pass
Login incorrect server	User presses login button with incorrect port field	GUI fails to connect to server, informs user of failure. Prompts user to check IP address and port	Pass
Login server failure	User enters correct email address and password and clicks log-in, whilst server is not running	GUI fails to connect to server, informs user of failure. Prompts user to check IP address and port	Pass
Login server failure	User enters correct email address and password and clicks log-in, enters incorrect server port/IP info	GUI fails to connect to server, informs user of failure. Prompts user to check IP address and port	Pass
System Login	User attempts to sign-in to the system twice with same email address	User is informed a user with those details is currently signed in, sign-in fails	Pass
System Login	User attempts to login to the system twice with same email address, on a different machine	User is informed a user with those details is currently signed in, sign-in fails	Pass
System Login	User enters correct email address and password, and clicks log-in, with correct server info, whilst user is not currently signed in	User is successfully signed in and taken to the next screen.	Pass

Test Sign-up functionality:

Feature	Test Case	Expected Result	Result
Enter Sign-up screen	In the login view, user selects the Signup button with empty IP address and port fields	Prompts the user to fill in IP address and port fields	Pass
Enter Sign-up screen	In the login view, user selects the Signup button with incorrect IP/port fields	Informs user of failure to connect to server, prompts user to check IP and port fields	Pass
Enter Sign-up screen	In the login view, user selects the Signup button whilst server is not running	Informs user of failure to connect to server, prompts user to check IP and port fields	Pass
Enter Sign-up screen	In the login view, user selects the Signup button with correct IP/port fields, while server is running	Successfully enters the sign up screen	Pass
Sign-up attempt	In the sign-Up screen, user selects signup button with all fields empty	Prompts user to enter a valid email address for sign-up	Pass
Sign-up attempt	In the sign-Up screen, user selects signup button with an invalid email address (e.g. Joeblogs235)	Prompts user to enter a valid email address for sign-up	Pass
Sign-up attempt	In the sign-Up screen, user selects signup button with a valid email address, but an invalid username (not A-Z,0-9, 5-20 characters)	Prompts user to enter a valid alphanumeric username between 5-20 characters)	Pass
Sign-up attempt	In the sign-Up screen, user selects signup button with a valid email, valid username, but an invalid password (not a-z,0-9, 8-20 characters)	Prompts user to enter a valid alphanumeric password between 8-20 characters)	Pass
Sign-up attempt	In the sign-up screen, user selects signup button with a valid email, valid username, and valid password, but confirm password field doesn't match.	Informs user that passwords do not match.	Pass

Sign-up attempt	In sign-up screen user enters correct information in fields, but the email address is already registered	Informs user email address is already registered	Pass
Sign-up attempt	In sign-up screen user enters correct information in fields, but the username is already registered	Informs user username address is already registered	Pass
Sign-up attempt	In sign-up screen user enters correct information in fields, with unique username and email address, and signs up.	Takes user info and stores it in the database. Informs user of successful sign-up, sends welcome email	Pass

Chat Screen – user interaction:

Feature	Test Case	Expected Result	Result
Individual avatar displayed	User signs into chat application and chat screen launches	User's pre-selected avatar should be displayed in the top left corner of the chat screen	Pass
Online user label	Two or more users sign into system	The label displaying the number of online users should update to display the accurate number of online users	Pass
Display all online users	At least two users sign into system	The list pane should show all users currently online, their avatars, and their status	Pass
Display conversation with a selected user	Select a user from the online user list	Update chat view with conversation with previous user	Pass

Sending a message to a user	Typing in a message in the message box and hitting the send message button	message sends to selected user and gets added to the bottom right hand side of users chat screen	Pass
Trying to send a blank message	Selecting send button without typing anything into the message box	No change to chat view, nothing sends to user	Pass
Receiving a message from a user	Have one user send a message to the current online user	Message received should be added to bottom left of the chat view	Pass
Sending a message longer than one line	User types a long, multi-line message and sends it to another user	Message is word wrapped into a short, multi-line message of a set width.	Pass
Sending a message without selecting a user	User types a message into the message field, and selects the send button without selecting user to converse with.	User is prompted to select a recipient	Pass

Password Reset:

Feature	Test Case	Expected Result	Result
Reset password	User selects the "Reset password" button on the login screen	Reset password screen opens	Pass
Reset password	User enters email address in the reset password screen, that does not exist on database	GUI informs user that no such email is registered.	Pass
Reset password	User enters email address in the reset password screen, whilst not connected to server	GUI informs user that server connection failed	Pass
Reset password	User selects the request button without entering a password	GUI prompts user to enter a password	Pass
Reset password	User enters a valid email address and selects request.	GUI informs user that an email has been send	Pass

System Exit:

Feature	Test Case	Expected Result	Result
Exit attempt	User selects exit button from within main chat window	Prompt appears asking if user wishes to sign out of application	Pass
Returning to chat screen	User selects no to sign out prompt	User returns to main chat window, connection kept	Pass
Returning to chat screen	User selects yes to sign out prompt	User is signed out, returned to sign-in screen. User is removed from other users' online user list	Pass
Terminating the application forcefully	Client application is forcefully terminated inside an IDE.	Client thread is stopped gracefully, sign-out protocol followed as above.	Pass

User feedback

For the following part of the evaluation, usability testing was undertaken. This was to ensure that unbiased and constructive feedback could be gained from users who'd had no previous interaction with the application, and no knowledge of its internal structure/design.

A goal-focused evaluation was undertaken, whereby users were given specific tasks to perform, but without any specific instruction as to how. The application server was launched, with one pre-made profile logged in. Users were sat at a machine with the application running, on the initial launch screen and were tasked to perform the following actions

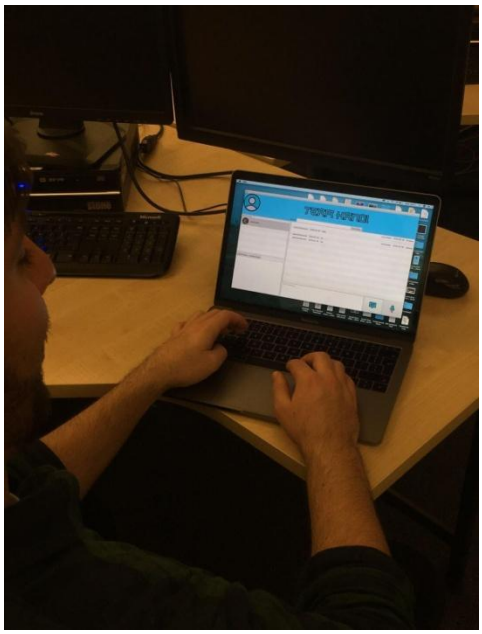
1. Sign-up to the system
2. Sign-in to the system
3. Start a conversation with another user, and send at least three messages
4. Exit the application

Given time and resource constraints, the usability study was limited to a small sample of individuals; however the feedback gained was useful nonetheless. All users were able to complete the actions, and most agreed that the application aesthetic was of a good standard.

The study did uncover several issues however that had not previously been identified in testing. Upon attempting to sign-up to the application, one user filled in the email and password field, and pressed the sign-up button, expecting to sign-up with those details, only to being taken to the stand-alone sign-up screen. This would suggest that the initial launch screen is not intuitive, and that users need prompting to select the sign-up button if they do not already have an account.

Another issue noted by the observers, occurred when a user typed a message and pressed the send button without first selecting another user to send a message to. This was likely due to only one other user being online at the time. It would perhaps be pertinent to prompt the users to select another user to begin a conversation with, or to disable the message box until a recipient is selected.

Other than these two issues, user feedback was generally positive. Users liked the look of the application, and its light-hearted aesthetic. Users noted they would prefer an increased selection of avatars or the ability to provide their own. This was something the team was keen to implement but failed to do so due to time constraints and lack of understanding of storing images on a database.



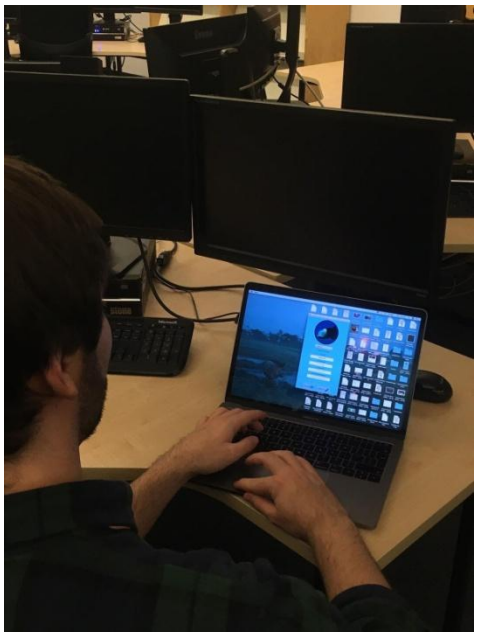


Figure 11. Usability Testing

Requirements Evaluation

Requirement	Status
Functional:	
Architecture:	
The system must accept keyboard input.	✔
The system must allow user (clients) to concurrently interact with a single server.	✔
The system must inform users if server cannot be reached.	✔
The system must inform user if the database cannot be reached.	X
Registration/login:	
The system must ensure user has entered valid email address, username, and password prior to completion of registration.	✔
A username must be between 5-20 characters.	✔
Passwords must be between 8 – 20 characters.	✔
Email addresses must be a valid email address following standard conventions.	✔
The system must ensure that usernames and email address are unique upon sign up.	✔
The system must authenticate users by email-address and password in order to login	✔
The system must differentiate users by login credentials.	✔
The system should enable user to reset password if forgotten via recovery service, utilising email provided on registration.	✔
The system should send the user an email receipt upon successful registration containing, registration details.	✔
The system could utilise remember me functionality for faster login.	X

Termination:	
The system must have log-out functionality.	✔
The system must terminate instances of the program when a user closes the program.	✔
The system must gracefully handle lost connections from a given client.	✔
User Profiles:	
The system should allow a user to set a status.	✔
The system could allow users to view their most frequently contacted list.	X
The system could allow user to store avatar/profile picture.	
The system could incorporate an award system based on user usage, e.g. trophies for frequent messaging/activity.	X
Chat:	
The system must maintain chat logs of users.	✔
The system must enable concurrent messaging to multiple users.	✔
The system must push new messages to clients.	✔
The system must indicate users currently active/online with whom, a user can message.	✔
The system should allow for group chats with multiple users at once.	-
The system should ensure users can be added to group chats.	-
The system must ensure only users in a chat can view their shared message log.	✔
The system should allow users to be part of multiple group chats.	-
The system could allow user to message all other users of the server.	✔
The system could allow user to search through users of the server to message.	X
The system must notify users if a message has been received from another user	✔
The system could display a pop-up notification when a new message is received	X
The system could allow users to delete their message logs.	X
The system could send messages to offline users as emails to the account associated with their profile.	X
The system could keep a record of the last 30 messages to be handled by the server in the case of server failure	X
The system could display the currently active/online users by most frequently contacted.	X
The system could implement lazy loading on chat logs.	X
The system could support the sending of multimedia files across chats.	X
The system could suggest a selection of quick messages e.g. "Hi there!", "How are you today?" etc.	X
The system could support the use of emoji.	X
The system could make a sound effect upon a message being received.	-
The system could make sound effects mutable.	X
The system could let users block other users from messaging them.	X
The system could notify group chat members if a group member leaves the chat	X
The system could notify a user currently in a chat if another user in the chat is typing a message at present	X
System:	
The system should encrypt user profile information for storage in the database.	✔
Non-functional:	
Usability/Ease of use:	
The system must have a graphical user interface with suitable signposting.	✔

The system must have consistency between user profile screens and chat logs.	✔
The system should be straightforward for user. Within 5 minutes user should have a basic understanding of the system. Following an hour of usage, errors per hour should be less than 1.	N/A
Efficiency:	
The system should not have excess latency time between a chat message being sent by one user and received by recipient(s) should be no more than 2 seconds.	✔
The system must support multiple active chats at a time without a decreased in performance.	✔
The system should have capacity to handle increases in system load during peak usage.	N/A
Availability:	
The system should be available 24/7.	N/A
Security:	
The system will not permit a user access to any functionality without authentication.	✔
The system will allow user access to their profile and chat logs following one level of authentication.	✔
The system will guard against unwarranted disclosure of user information to third parties.	✔
The system should hash all user information and messages in the database.	X
The system must only allow the participants in a conversation to view its content	✔
Maintainability:	
The system must be coded in an efficient manner and well documented in order to facilitate future maintenance and scaling with minimal complications/debugging required.	✔/X
Implementation:	
The system should be coded in Java where possible making use of object - oriented programming techniques.	✔
Scalability	
The system could allow the user remote access via smartphone, tablet or computer.	X

Self Evaluation

The project brief was to develop a system that implemented client server architecture across a network, utilising sockets, threads, syncronisation whilst avoiding races and deadlock. The system was also to include a 'polished GUI'.

As a team the consensus was that *Oi-Oi Hanoi* comfortably met this compulsory guideline, while also implementing features and functionality beyond the scope of the project, such as JavaFX utilisation for the GUI. Moreover, through the development of a robust database the team was able to produce a well realised client-server protocol allowing for the parsing of well thought-out message objects between all facets of the system.

Over the four weeks of the project Team Hanoi worked cohesively and built upon skills and learning introduced within the module, and built a product which we can be proud of.

Though the project was well planned and approached with a dedicated mindset, there were areas which were more challenging than anticipated, and in hindsight would have benefited from a more cautious approach. This would include the road map for developing the protocol, defining a more concrete plan for how key system components would interact before initiating any development. It was also felt that while team had an eagerness to populate the GUI in a finalised form, the design

would have benefited from a steadier integration to allow for increased focus on the importance of model-view separation.

Ironically, it was the teams haste to deploy the core functionality of the application that may have prevented the inclusion of more advanced features.

Such features would have included the ability to upload personal avatars, to customise the look and feel of the chat window from a configurations panel, the ability to order the online user list by most frequently contacted, and a more actualised group messaging component.

Appendix

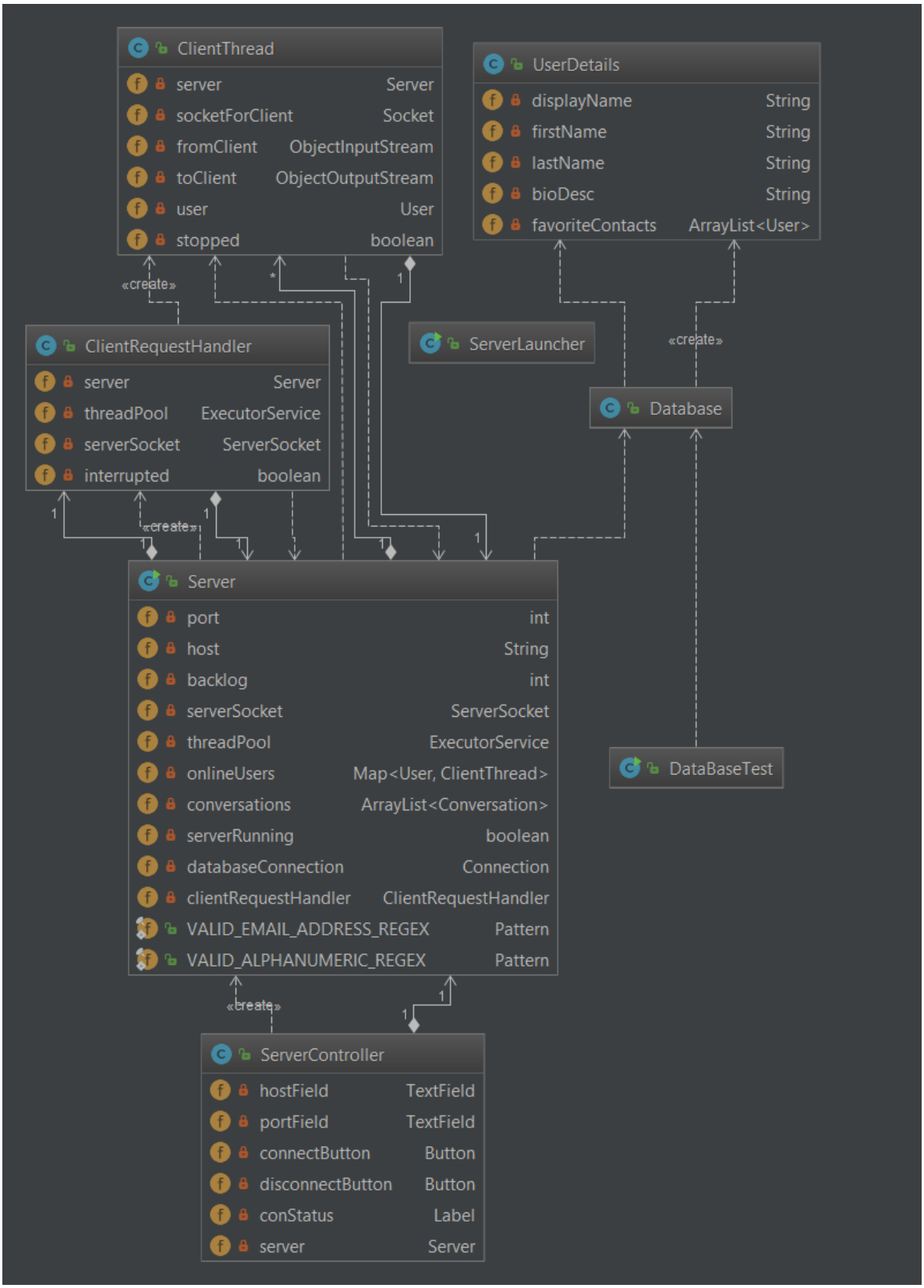


Figure 12. Server Class Diagram

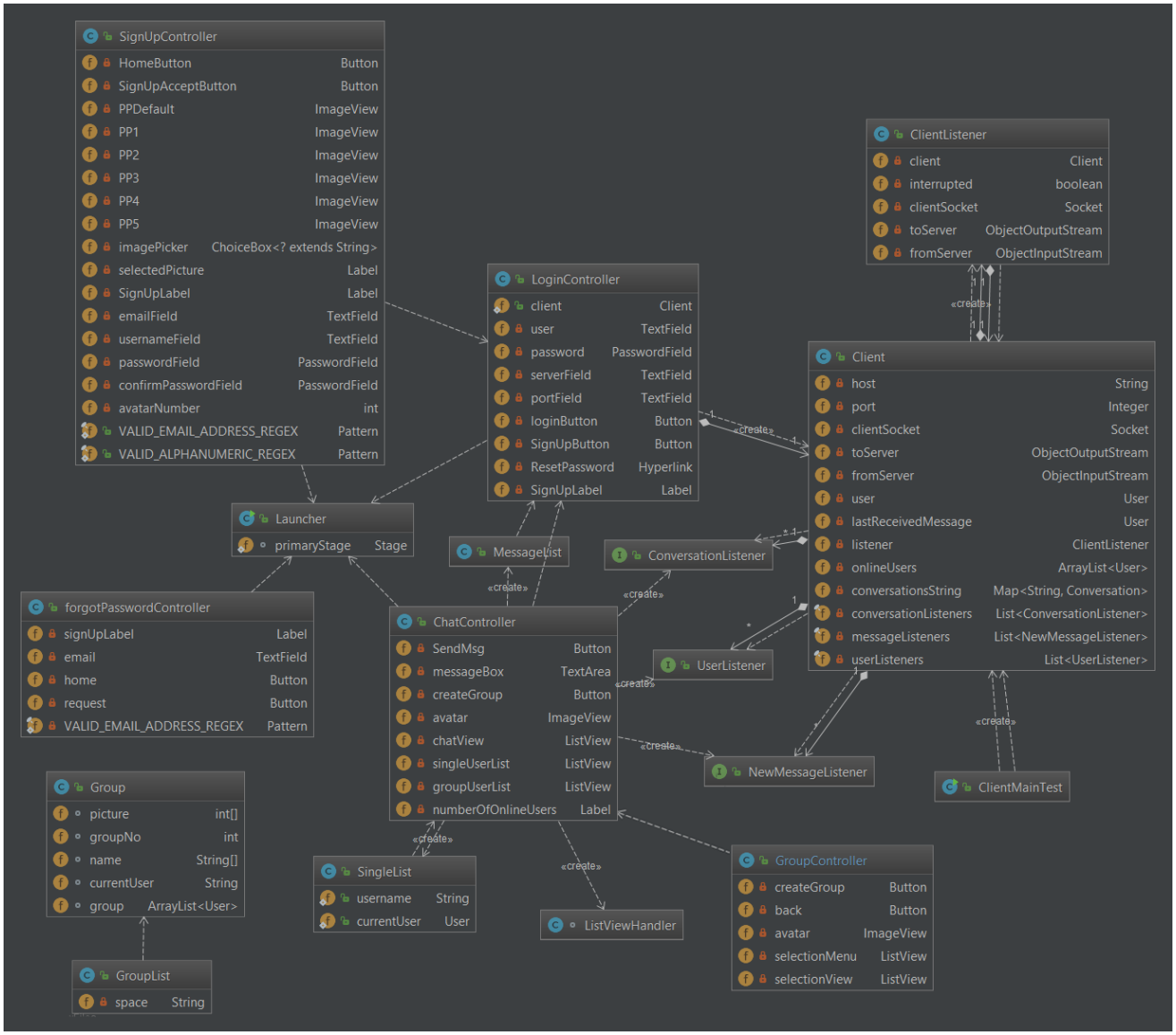


Figure 13. Client Class Diagram

Project Diary

Meeting minutes No.1

Location: University of Birmingham, Computer Science Building, Sloman Lounge

Date: 20th February 2018

Time: 11:00am

Attendees: Lee Fletcher, Danyal Bajar, Joshua Anderson, Tom Creaven, Adam Robinson

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Setting up git on all members machines for version control of source code	Ensure that everybody has git running correctly on personal machines	Postponed until we meet with Cory.
Decide on project area	Team discussed ideas on what project should be on. Team discussed own ideas, and the merits of doing a suggest idea in the handout.	Team mostly decided on creating a client server chat application. Will formalise in tutor meeting Tuesday 27th February.
Group communication	Created a messenger group to communicate and decided to meet at least once a week.	

Action items	Owner(s)	Deadline
Produce a Gantt chart to outline project plan for project time management	Adam	Next Tutor Meeting

Meeting minutes No.2

Location: University of Birmingham, Learning Centre

Date: 20th February 2018

Time: 2:00 pm

Attendance: Cory Knapp, Adam Robinson, Joshua Anderson, Tom Creaven, Lee Fletcher, Danyal Bajar

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Discuss definite allocations of roles for each team member within project.	Ensure all roles are concrete and responsibilities are understood.	All roles successfully allocated.
Query whether use of javaFX framework was allowed for project.	Discussion differences between JavaFX and SWING.	Cory said that the use of JavaFX framework to construct GUI was a good idea. However it needed to be verified by Uday.
Seek tutor approval of desired project. (Instant Messenger)	Brief proposal given to tutor of project plan	Cory advised that although it's a good idea, it would need extra functionality in order to receive the desired grade.
Get a better understanding of how we should be progressing	Asked Cory what he'd expect the group to have achieved by the following weeks tutorial.	Aim to have the client server protocol laid out. Aim to have a basic GUI frame created

Action items	Owner(s)	Deadline
Begin actively planning the structure of the project classes and methods. Plan protocols for server class and plan framework for GUI Gant chart here	All	Next tutor Meeting

Meeting minutes No.3

Location: University of Birmingham, Learning Centre

Date: 21th February 2018

Time: 1:00 pm

Attendance: Tom Creaven, Adam Robinson, Lee Fletcher, Danyal Bajar, Joshua Anderson

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Requirements Generation for the system	As a group we and generated functional and non functional requirements for the system	We collected some requirements but decided that we needed more.

Action items	Owner(s)	Deadline
Have functional and non-functional requirements completed and compiled.	Danyal Bajar	Next meeting

Meeting minutes No.4

Location: University of Birmingham, Learning Centre

Date: 23rd February 2018

Time: 2:30 pm

Attendance: Tom Creaven, Adam Robinson, Lee Fletcher, Danyal Bajar, Joshua Anderson

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Discuss project plan.	Based off the requirements that were brought back by Danyal, the team discussed the potential structure of the system, the classes that would be needed and method functions.	It was decided that more research would need to be done. Further research of existing systems would be beneficial as it would give idea of starting point.
Tutor not connected to Git repository.	Cory needed to be connected to git repository.	Email would be sent to Cory seeking his assistance in getting him connected to the project repository. Hai to be added to Git when he returns from holiday

Action items	Owner(s)	Deadline
Team to research potential structure of application in terms of code make up and visual layout	All	Next meeting
Email to be sent to Cory regarding his connection to the project repository.	Tom	Before end of the day.

Meeting minutes No.5

Location: University of Birmingham, Computer Science Building, Ground Floor Booths

Date: 26th February 2018

Time: 03:45 pm

Attendees: Lee Fletcher, Danyal Bajar, Joshua Anderson, Tom Creaven, Adam Robinson

Apologies:

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Discuss how to implement database	Team experimented with DataGrip and JDBC	Danyal was allocated to the task of research for the database and how it would connect to the program.
Database Tables	Brief discussion on tables that would be required within the database for storage of user credentials and conversations	Danyal to research key relations across tables

Action items	Owner(s)	Deadline	Status
Danyal to research implementation of relational databases and JDBC.	Danyal	Ongoing	

Meeting minutes No.6

Location: University of Birmingham, Learning Centre, Ground Floor

Date: 27th February 2018

Time: 02:00 pm

Attendees: Lee Fletcher, Danyal Bajar, Joshua Anderson, Tom Creaven, Adam Robinson, Cory Knapp

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Update on progress.	We as a team updated Cory on the progress on the project. We discussed timeline function ideas and timeline milestones for the project.	Cory informed us that the by the following week we should, at the least, have a skeleton structure for the classes being implemented within the program. He also informed us of what type of project quality is expected from our group to achieve the desired grade.
Update about team's Official Tutor.	We discussed Cory being invited to the project repository but he informed us that Hai Nguyen was our official tutor and this would be his last week tutoring us.	Cory instructed us to email Hai and wait for a response and add him to the project repository as quickly as possible.
GUI Interaction	High level discussion on the GUIs appearance and the screens that would be needed	Adam was to begin making a prototype of the program interface

Action items	Owner(s)	Deadline	Status
Ensure server has skeleton structure for class implementation to begin work.	Lee & Joshua	Following Week	
Email & Add Hai to the project repository.	Tom	ASAP	
Production of a prototype GUI	Adam	ASAP	

Meeting minutes No.7

Location: University of Birmingham, Computer Science Building, Ground Floor Booths

Date: 1st March 2018

Time: 03:45 pm

Attendees: Lee Fletcher, Danyal Bajar, Joshua Anderson, Tom Creaven, Adam Robinson

Apologies:

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Brief team on Server class updates	Lee and Josh had returned with various attempts of a Server class to begin the production of the business logic for the project.	It was decided as a team that Lees attempt was better suited as a lead for the project - Joshua agreed to pair program, using Lees current source code as the foundation.
Brief team on GUI class updates.	Adam & Tom to present ideas and prototype for GUI.	Group agreed that positive progress had been made. Adam & Tom continued with the development if the GUI.
Brief Team on Database table updates.	Danyal briefed the team on his progress with creating the database, It was agreed that another table would need to be created to accommodate "User".	Danyal and Lee agreed to work closely so that they could work together to connect the java program to the database.

Action items	Owner(s)	Deadline	Status
Remain in close communication in regards to JDBC	Danyal & Lee	Ongoing	
Go ahead given on UI layout and designs to be implemented	Adam & Tom	Ongoing	

Meeting minutes No.8

Location: University of Birmingham, Computer Science Building, Ground Floor Booths

Date: 2nd March 2018

Time: 4pm

Attendees: Tom Creaven, Adam Robinson and Uday Reddy (Head of Module)

Apologies:

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Is the use of JavaFX allowed for this project?	We spoke to Uday about our idea to use JavaFX over SWING and sought his approval.	Uday authorised the use of JavaFX.
Is it imperative that program must run exclusively from school machines?	Tom and Adam informed Uday that they had issues with running JavaFx on the university machines. Uday advised that it was not imperative but rather it was more important that demonstration of the project occurred on any machine. Uday stated that the use of personal machines would not costs marks. Uday advised that at least 3 machines were needed. 1 machine as the server and 2 other machines acting as clients.	Informed Uday that we will aim to have the program (Server & Clients) running on school computers, but if that failed we had a good contingency plan, in being able to use personal machines.

Action items	Owner(s)	Deadline	Status
Inform group of the update.	Adam & Tom	ASAP	

Meeting minutes No.9

Location: University of Birmingham, Learning Centre, Ground Floor

Date: 06th March 2018

Time: 03:45 pm

Attendees: Lee Fletcher, Tom Creaven, Adam Robinson, Joshua Anderson, Hai Nguyen

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Meet Hai and introduce him to our project.	Meet with Hai, our official project tutor and informed him of our chosen project area.	Hai was happy with the progression of our project area and gave constructive criticism of how to handle server protocol
Inform and show Hai of current progress of project.	As a team we informed Hai of the progress that had currently. We updated him on the advice that was previously given to us by Cory. We showed Hai our project GUI and explained its planned functionality. We laid a proposal down as to how we planned to structure our project from current stage to the finish.	Hai agreed that our project structure plan was good. Hai however, went on to advise that we would need more than one relational database for our project and made points for us to think about regarding user authentication and encryption in the form of hashed passwords.

Action items	Owner(s)	Deadline	Status
Seek ways in which "User authentication" can be implemented into the program.	All	ASAP	
Find a way to stored hashed passwords in the database for increased security	Adam & Danyal	ASAP	

Meeting minutes No.10

Location: University of Birmingham, Learning Centre, Ground Floor

Date: 07th March 2018

Time: 01:45 pm

Attendees: Lee Fletcher, Tom Creaven, Adam Robinson, Joshua Anderson

Apologies:

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Lee and Adam provide update on Server and GUI	Both team member presented updates to code for their respective tasks. Various questions were asked about functionality.	The team agreed that progress so far had been extremely positive.
Update team on Uday authorisation to run project on personal computers	Tom and Adam formally updated the team on the topic discussed with Uday in the meeting.	The team agreed that this provided a good contingency plan should things not go as desired, however it was still the goal to have the program (Clients and server) to run from school machines.
Network two laptops for cross communication	Visited IT support team to find out what would be required to network personal machines on university network	Using local IP address on the university network was sufficient. No additional networking hardware would be required

Action items	Owner(s)	Deadline	Status
Work closely and pair program with partners on allocated task area	All Members	ASAP	

Meeting minutes No.11

Location: University of Birmingham, Computer Science Building, Sloman Lounge, Ground Floor

Date: 09th March 2018

Time: 2:00pm

Attendees: Lee Fletcher, Danyal Bajar, Tom Creaven, Adam Robinson, Joshua Anderson

Apologies:

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Lee and Adam provide update on Server and GUI	Both team member presented updates to code for their respective tasks. Various questions were asked about functionality.	The team agreed that progress so far had been extremely positive.
Danyal provide update on database	Danyal provided a verbal update on the relational database and advised that members working on the server would need to work closely with him.	His proposal was agreed by the team.
Absence on Tuesday	Joshua advised the group of his absence for the next week tutorial due to Assessment Centre.	Team agreed that it was ok.

Action items	Owner(s)	Deadline	Status
Agreement made to work closely with Danyal.	Lee	ASAP	

Meeting minutes No.12

Location: University of Birmingham, Learning Centre, Ground Floor

Date: 11th March 2018

Time: 02:00pm

Attendees: Lee Fletcher, Tom Creaven, Adam Robinson, Danyal Bajar, Hai Nguyen

Apologies: Joshua Anderson

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Discuss difficulty passing objects between stages in JavaFX	The team present discussed with Hai the difficulties they were having with passing objects between view controllers in JavaFX.	It was decided by the team present that they would just change the way implementation of view modifiers in the GUI. Hai said he would update project leader with advice after doing some research.

Action items	Owner(s)	Deadline	Status
Implementation of agreed changes to be carried out	Adam & Tom	ASAP	

Meeting minutes No.13

Location: University of Birmingham, Computer Science Building, Ground Floor Booths

Date: 15th March 2018

Time: 5:00pm

Attendees: Lee Fletcher, Danyal Bajar, Joshua Anderson, Tom Creaven, Adam Robinson

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Decide how active conversations will be stored for each user inside the program.	A suggestion was given to store the chats locally as this would ease complexity but an argument was made raising concerns about potential backlog and how those messages would be stored.	It was decided that the chat history would be pulled from database and stored locally in memory when a user was selected.
Decide on other functionality that can embellish the final product.	Various suggestions were given from all group members.	We decided on image sharing and voice note sharing.
Project report	Team to focus on writing areas of the report related to their implementation area.	Project report required focus

Action items	Owner(s)	Deadline
Begin project report	All Members Except Lee	ASAP
Continue on assigned tasks	All	ASAP

Meeting minutes No.14

Location: University of Birmingham, Computer Science Building, Ground Floor Booths

Date: 16th March 2018

Time: 5:00pm

Attendees: Lee Fletcher, Danyal Bajar, Joshua Anderson, Tom Creaven, Adam Robinson

Apologies:

Agenda Focus:

Agenda Item	Discussion / Action	Conclusion
Lee and Adam present model view issue within program	Both team members explain the current problem and seek advice from team.	Team discussed possible solutions. It's agreed that Tom leaves the group report shortly to pair program with Adam and Lee to help fix bugs.
client listener having class issues with socket load and protocol	Lee discusses issue with client listener class not being able handle sockets correctly	Lee to experiment with opening two sockets for each client thread.

Action items	Owner(s)	Deadline
Continue with report.	Joshua & Danyal	ASAP
Continue on assigned tasks	All	ASAP