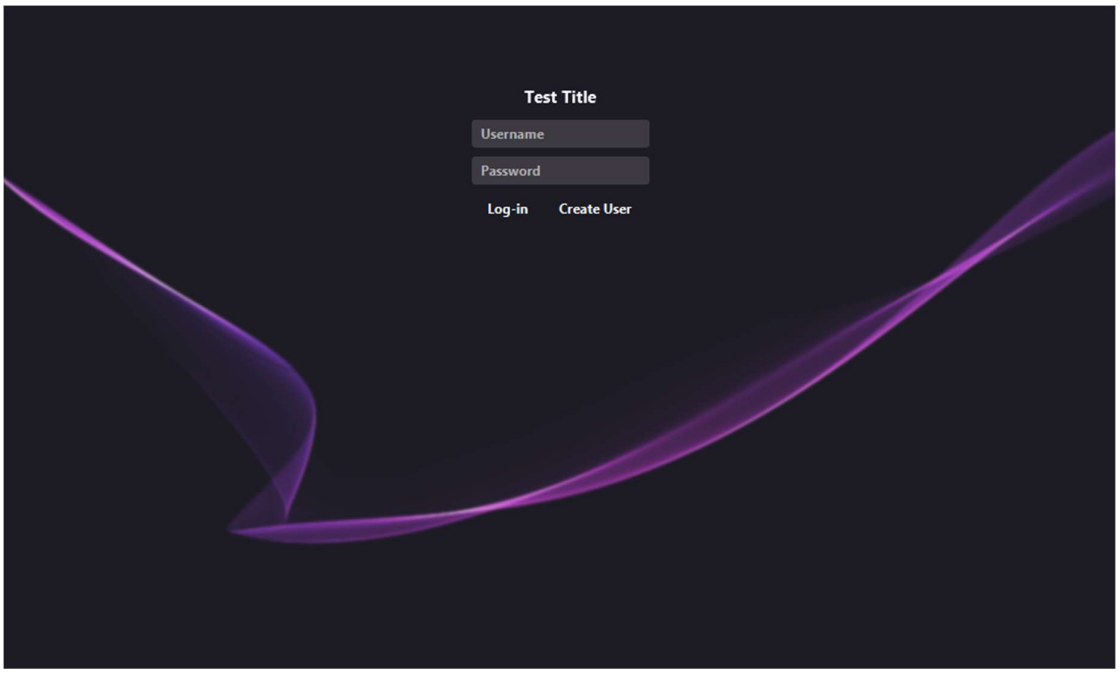
Mini Project Report:
Chat system



Group members:

Andreas Jin Sum

Frederik Darling Larsen

Nicolai Bouquin

Rune Korsgaard Ludvigsen

Ulrik Kampel

Teacher: Eva Triantafyllou

Date: 8/11 - 2020

Workload delegation

During this project there were some troubles with both delegation of work and even more so with the GitHub contributors' systems used to monitor the distribution of the workload amongst the group members. The issue with the contributors system was that it counted any lines, not just added but also changed during development which meant that there were some huge discrepancies between how many lines of code each member had contributed to. Even though this discrepancy in reality was not as significant as it was made to be. Small changes also completely erased other members' contribution and instead credited the member who made the changes. This meant that the GitHub contributors' system did not adequately present how well the workload was delegated. Other than these issues there were also some other issues regarding the delegation of the workload. This was due to how the previous workload delegation in semester projects had led to acquiring different skill sets during previous projects. It meant that about half the group had a decent amount of experience with coding and more specifically coding in groups, while approximately the other half of the group had focused more on report writing and testing, and therefore did not have the same experience with coding. The group was then split into two teams to delegate the workload; Client and Server. The client side team consisted of Rune and Jin. The server side team consisted of Frederik, Ulrik and Nicolai. Hence the challenges faced with the contributors system, it was decided to collectively work together on one computer. The server was primarily contributed to by the three man group on the server side team, which is not shown by Githubs system. At the project's end both teams merged into one and collaborated together from just a single group member's screen through the Discord screen sharing function.

In the beginning, the classes that had to be programmed were delegated almost equally among the group members. Quickly it was realized that some group members had an easier time implementing the classes they were given. This was both due to the difference in skill sets, but also due to the variation in complexity among the classes. When the first sprint for the programming of the server and client was done some classes were yet to be finished. The group members that still had classes to finish were assisted by other group members by the screen sharing function over Discord. In this way the group members that had issues with their classes were able to learn by entering the code themselves while attempting to understand it while being assisted by other group members. This meant that although these classes were written on one group member´s computer, they were created by multiple group

members working together. It did also mean that some members did not get to code all that much by themselves. This was partially resolved later by having the group members that had worked on most of the code to be the ones writing documentation and the report in the next sprint. These group members were then available to assist the other group members if they encountered any issues with coding the remainder of the program. There were still some issues with this approach because some group members also faced issues with implementing the rest of the program. Therefore, all group members had to contribute to finalizing the program which meant that there was a disparity in the workload all together by the end of the project.

Use of Scrum

In order to incorporate scrum into the mini project the online planning tool Trello was used in order to manage the sprints.

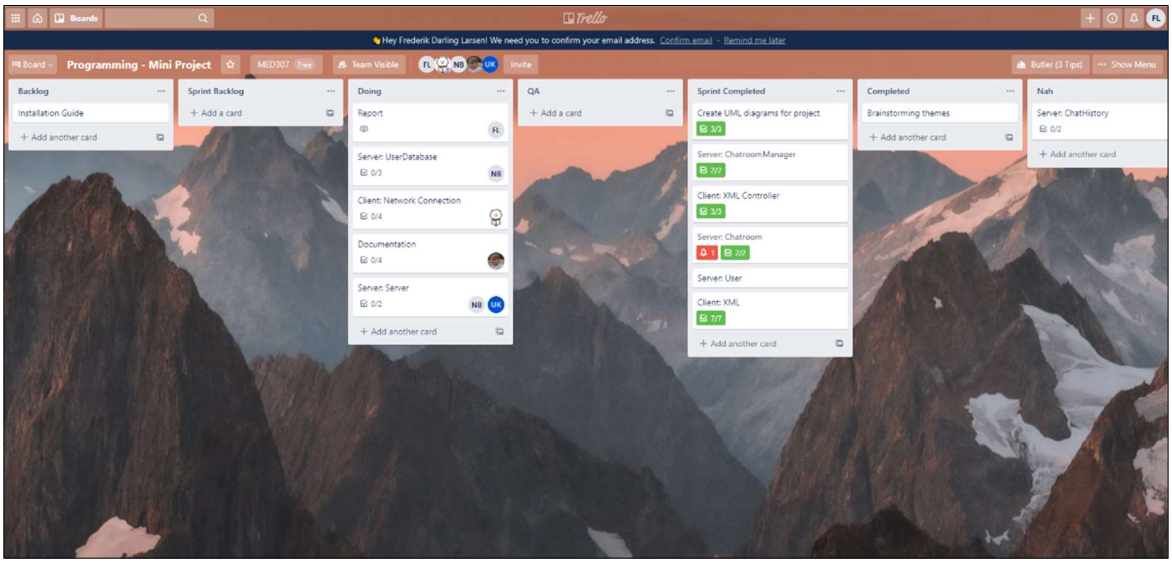


Fig 1: Screenshot of the Trello board showcasing the management of sprints

The board was divided into the sections: backlog, sprint backlog, doing, (Quality assurance) and completed. The backlog is where all the possible tasks to be completed for the whole project was listed. These tasks were all presented on cards each containing a checklist for subtasks. These subtasks could for example be the individual sections of the report or the individual methods of a class that had to be implemented.

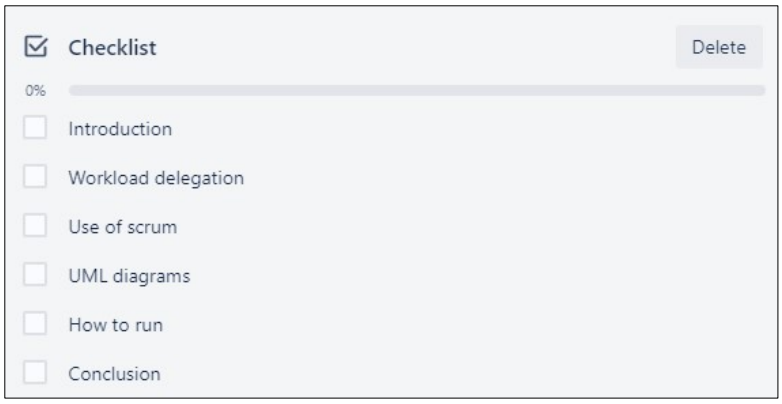


Fig 2: Shows the checklist for the report card

Upon the beginning of a new sprint the project tasks moved to the sprint backlog section. These tasks were then divided among the group members by adding the Trello users of the group members to the tasks. When these tasks were to be done by one or more of the group members, the task card was moved to the "doing" section. While in this section it was possible to track whether the subtasks were completed by looking at the checklist where the group members responsible for completing the task also had to checkmark each sub task when they were completed. When a task was fully completed, the task card was moved to QA. While in this section other group members would look at the result of the task, whether this was code or documentation in order to check whether the task was completed as intended and that no errors were to be found in the code. After QA had been done the card was finally moved to completed, which meant that the task was fully done and that the group members responsible for said task could move on to a new task.

This mini project also incorporated the concept of daily scrum. This was done with daily meetings in the beginning of each workday, where each group member would give a summary of what they had been working on along with mentioning the challenges that they might have faced while completing their tasks. These meetings were kept short in order to quickly begin working unless some crucial issue came up that had to be addressed.

UML diagrams

UML diagrams were created to showcase the intended structure and functionality of the code before implementing any of it. This was useful in order to get an idea of what had to be implemented and to know where to start. The challenge with creating the UML diagrams before writing a single line of code was that it was not certain that the final program would be implemented in the same manner as for example, the class diagrams initially illustrated.

Hence the class diagrams and sequence diagrams were updated when something was implemented that did not match the diagrams. This meant that the final diagrams gave a good overview of the final program.

Use Case diagram

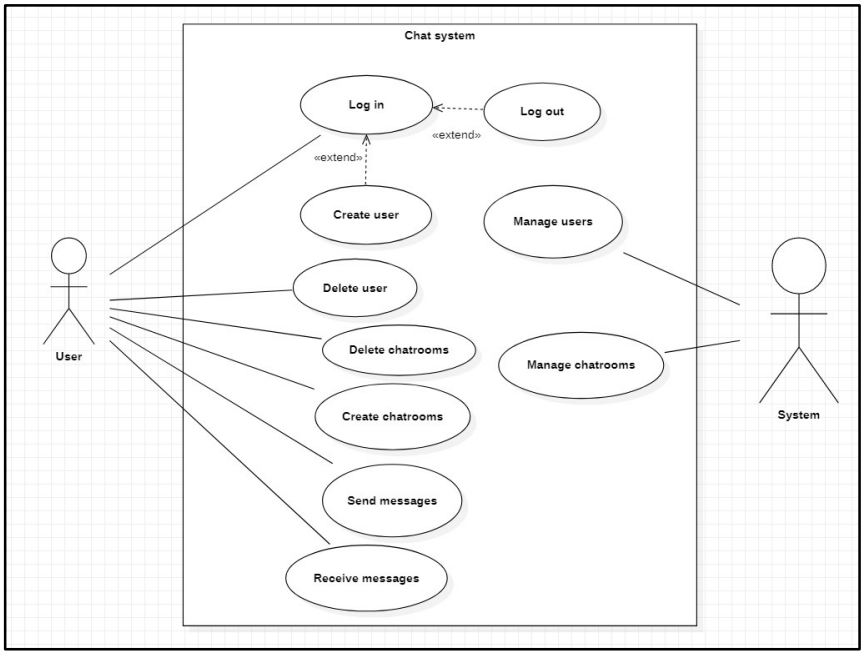
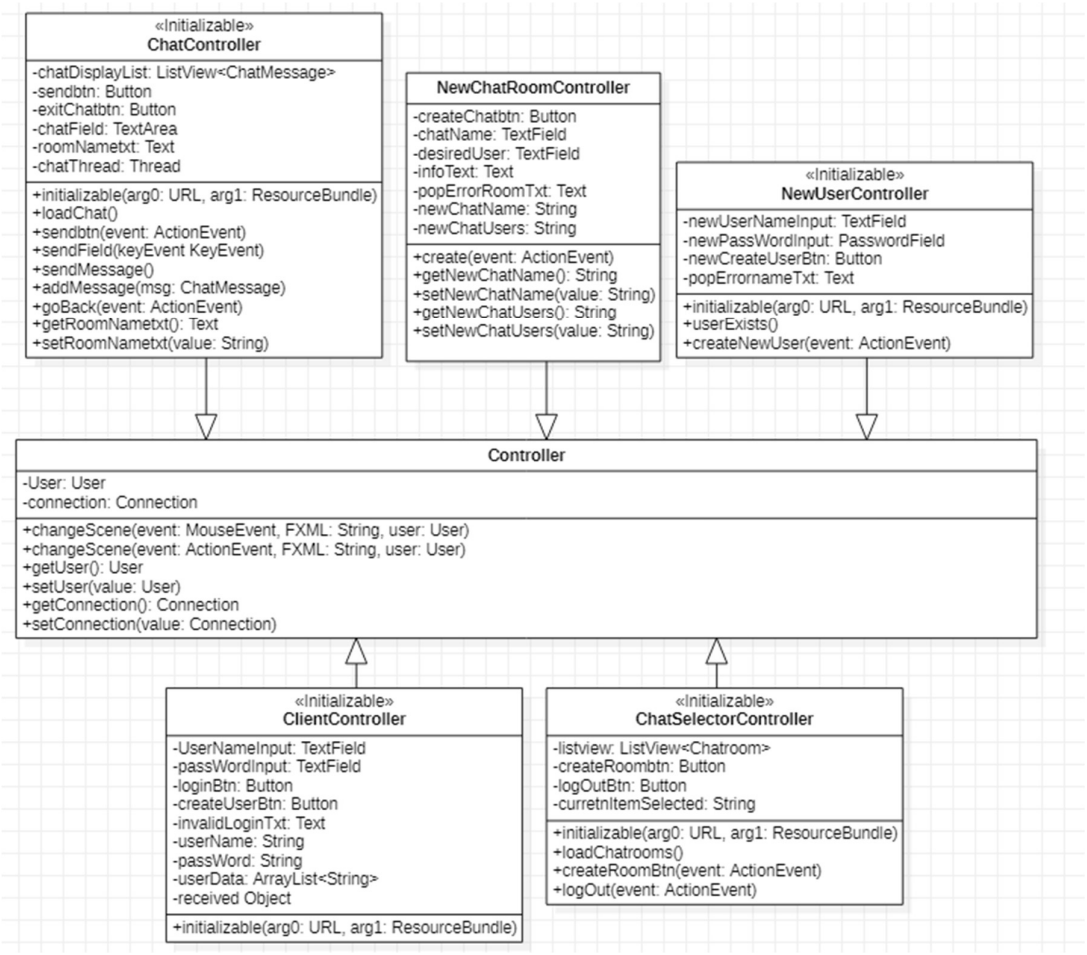


Fig 3: Use case diagram for the chat system

The first diagram created for the chat system was the use case diagram. This diagram was created in order to get a general overview of what the user would be able to do with the chat system and what the system would have to handle in order to provide the service to the user. The diagram shows how a user can log-in if they have a user. If they do not have a user, they have the option to create a new user. This is shown by the extends relationship from “create user” to “log in”. The extends relationship from log out to log in shows how the user has the option to log out when the user is logged in, as well as the option to log in again after having logged out. The user also has the option to delete a user. They have the option to create and delete chat rooms along with sending and receiving messages. The system as shown in the diagram must manage the users and manage the chatrooms. This is done by sending and receiving information about the users and chat rooms from the client to the server and back to the same or another client. The user and chat room objects are created from classes which will be elaborated on next in the explanation of the class diagrams and their relationships.

Class diagrams

The class diagram for the chat system is much closer to the final implementation than the use case diagram, since the specific classes that had to be implemented had to be named and given attributes and methods. This also meant that the class diagram had to be updated a lot during the implementation phase since the structure of the program changed during implementation, and certain classes had to be removed and added while other classes were implemented differently from the initial class diagram. One major change was that the group realized that the chat history and the created users did not have to be saved in a database in order to create a functioning application. The idea originally was to save the chat history and the users that had been created to a file, but this turned out to be difficult. If this was running on a real server, it might also not have been all that necessary because servers usually are running the whole time. Another major change was the data type that was used to store lists of data. When the class diagrams were first designed it was not accounted for that an Array List would be more useful when adding and removing elements like users and chatrooms to a list than regular arrays.



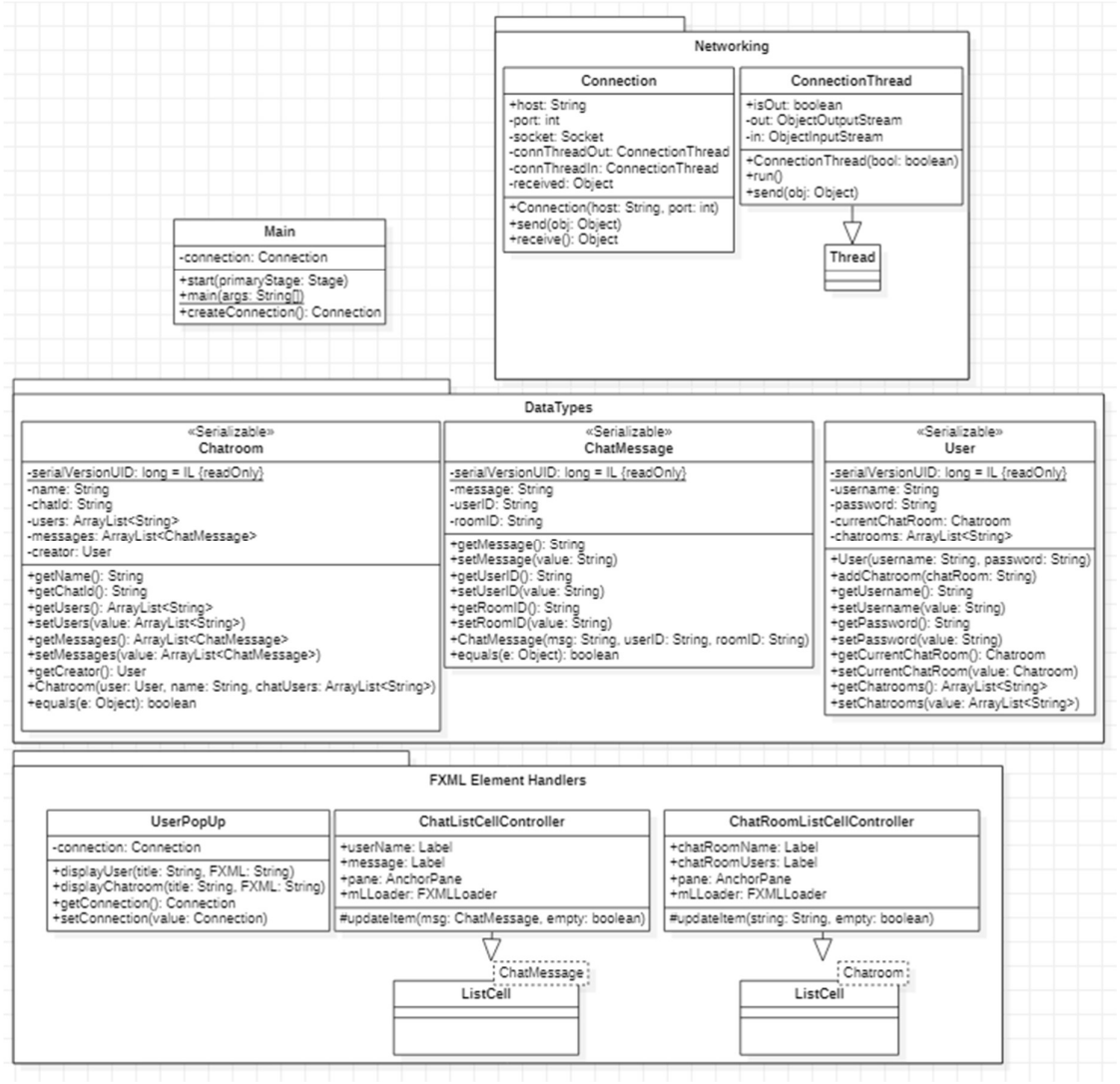


Fig 4: Class diagram for the Client

The class diagram shown in fig. 4, is of the classes implemented in the client side of the software. The Controller classes each have a FXML file that references them. The FXML files are called in the start in main, which also loads the controller class with it. The Controller classes function as the logic behind the GUI created from the FXML files.

We have three datatypes which are used to send data between client and server, therefore they implement the Serializable interface to make it easier to convert to binary and send to the server which then convert them back.

The FXML Element Handlers, are the creates the popups and determine how the lists in the GUI shows the data stored in them.

Network keeps track of what is sended and received, a connection Thread which extends Thread, updates constantly while the rest of the code can continue running, looking for data as well as sending what might be sended. It then sends the information to where it is asked for.

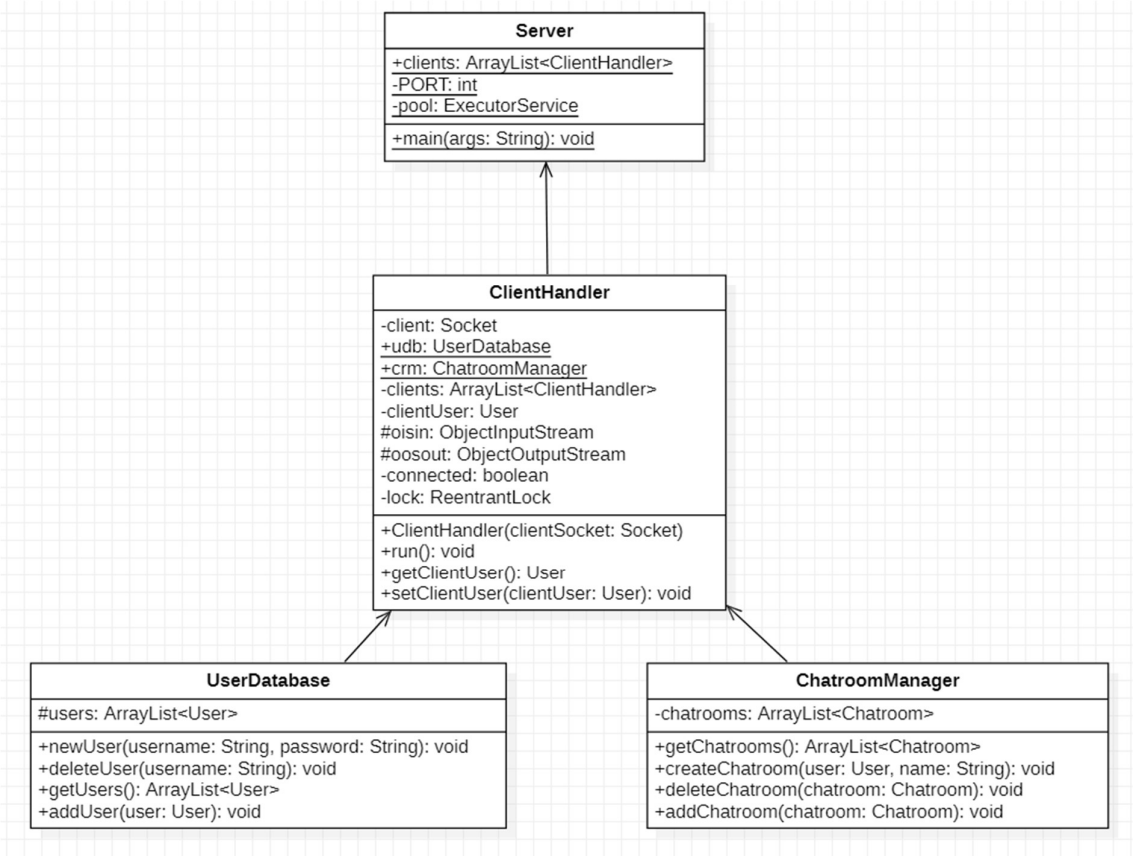


Fig. 5: The class diagram for the server

Shown above in fig. 5 is the class diagram for the server. The server also uses the classes from client listed as data types i.e. User, Chatroom and ChatMessage but is not shown here as they have already been shown and explained in the client class diagram. The class diagram for the server, shows the Server class which establishes the connection to the clients, the ClientHandler class that handles each client connected to the server and the two classes UserDatabase and ChatroomManager which stores the users and the chatrooms that are managed by the ClientHandler. The Server contains an Arraylist of ClientHandlers where a new ClientHandler is added for each new client that connects. The client also contains a port which is the number that that client must also have to be able to connect. The main method executes the server. The ClientHandler is where the server receives and sends messages through the ObjectInputStream and the ObjectOutputStream. This is also where the static

objects for the UserDatabase and the ChatroomManager are instantiated. The ClientHandler also contains a User object that stores the specific user that is currently using the specific client. Each client runs on its own thread. When the thread is running, the run() method of the ClientHandler is executed. It is in this method where the data is received from the clients and sent back to the clients. The UserDatabase class stores all the users that have been created by the clients in an Array List. The UserDatabase contains methods to create a new user, delete a user, get all the users and add a user. The ChatroomManager class stores all the chat rooms that the server has received from the clients. The chat rooms can be accessed using the getChatrooms() method. Chat rooms can be created with the createChatroom() method, deleted with the deleteChatroom() method and a chat room can be added with the addChatroom() method.

Sequence diagram

The sequence diagram tells the story of how the user and different parts of the program interacts with each other in order to complete the tasks intended by the user. The sequence diagram went through some changes during the span of the project, as we realised that we needed new methods, variables and classes. The final version of the sequence shows a simplified version of the interaction between the client, server and classes used for storing Arrays of users and chat rooms.

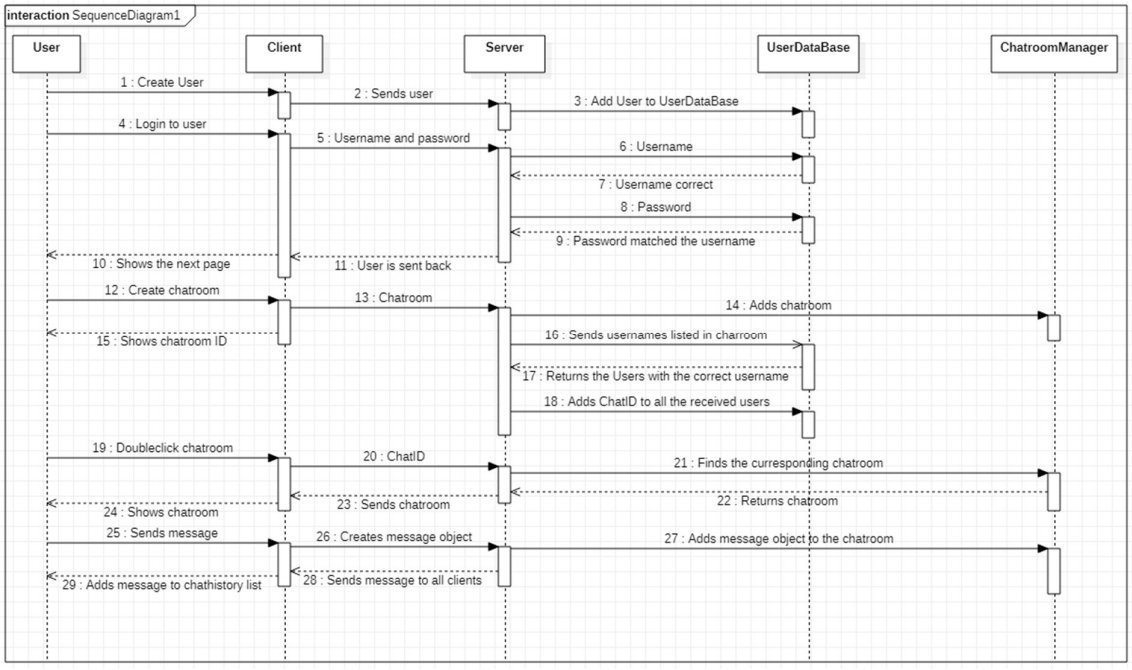


Fig. 6: Sequence diagram of the application.

The sequence shows how the user has an indent, which leads to an action. the user is here interacting with the client side of the program, and from there the client will send different objects to the server in order to create and store information in the UserDataBase and ChatroomManager. If we take a look at message nineteen we see the user open a chat. The client sends the chatID to the server, which then finds the corresponding chatroom and sends it back to the client, which then shows it to the user.