University of Southern Denmark, Odense

The Faculty of Engineering / MMMI

BSc in Software Engineering

SB3-GLO Design of software systems in a global context

Chat Application - SYNchat
Semester project E18

Semester coordinator: Ronald Jabangwe - rja@mmmi.sdu.dk
Supervisor: Danish Shaikh - danish@mmmi.sdu.dk

Project group 9

Name	Email
Alexander Micheelsen Rol	alrol17@student.sdu.dk
Lasse Fisker	lafis17@student.sdu.dk
Josef Ngoc Can Pham	jopha15@student.sdu.dk
Sigurd Eugen Espersen	siesp17@student.sdu.dk
Peter Stærdahl Andersen	pande15@student.sdu.dk
Rasmus Jensen	rasje17@student.sdu.dk
Patrick Christoffersen	pachr16@student.sdu.dk

Project period: 06/09/2018 - 18/12/2018

Number of pages: 40

Abstract

This paper examines the analysis, design, implementation and overall thoughts behind the project, SYNchat. The authors and developers in this project are Software Engineering students, and this paper is the main deliverable of their 3rd semester project. SYNchat is intended to be a free communication platform, with the focus of making it easier for people from all cultures to connect with each other digitally. Some of the focal points in the project are the cross-cultural communication, user-friendly interface, distribution, scalability and security.

The main purpose of this paper is to document the workflows and development process from project start till final hand-in of a functional and running prototype of the desired system. The reader will get a better insight into the thoughts and effort the group has put into the project with the main theme of design of software systems in a global context. The project follows an agile, iterative, and incremental development through the use of the Unified Process and Scrum.

Preface

This project on the 3rd semester is open-ended with only a few requirements. It has to be a Server-Client-System and is not allowed to be a video game. Also, the project has to relate to the overall theme of the semester “Design of software systems in a global context”.

In addition, the following semester subjects are all a part of and contribute to the project: Design of software systems (DES), Network and operating systems (OPN), Cross-cultural management (CCM) and Discrete mathematics (DM). All together they create the foundation for the project course; Design of software systems in a global context (GLO).

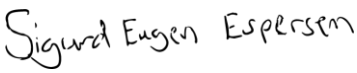
The project started on September 3rd and has a deadline on December 18th 2018 at the University of Southern Denmark, in Odense. Throughout the project, the group has been under guidance from supervisor Danish Shaikh and semester coordinator Ronald Jabangwe.

This paper is written by all 7 members of group 9, all of which study Software Engineering on their 3rd semester at the Faculty of Engineering under the Maersk Mc-Kinney Møller Institute. All members acknowledge their active participation in the project mutually through their signatures below.

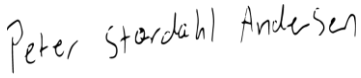
Alexander Rol

Lasse Fisker

Josef Pham



Sigurd Espersen



Peter Andersen

Rasmus Jensen

Patrick Christoffersen

Table of Contents

I Reading overview

This paper is written in a way that allows it to be read in a chronological order for the reader to get the best understanding of the overall project and system, SYNchat.

Subchapters are not independent and should not be read out of context from the full chapter, as previous subchapters may contain information about the content of the following section.
It is recommended to have a copy of the Java code open when reading the implementation section since references to the code will be used.
Classes and methods will be marked using *cursive font*, in addition methods will be tagged with the suffix ().
methods will be marked using *cursive font*, in addition methods will be tagged with the suffix
References throughout the paper, will be listed using Harvard Referencing, with the last name of the author, year and a numbering if the same author is used multiple times.

II Glossary

Term	**Description**
Docker	Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. (Opensource.com, 2018)
VM	Virtual Machine
Client-Server	Computer networking connecting a personal computer to an external server
GUI	Graphical User Interface - What the user sees
OS	Operating system
SDU	University of Southern Denmark
CSS	Cascading Style Sheets - Handles visualization of GUI
API	Application Programming Interface - Predefined methods used to communicate between components
SSH	Secure Shell - utilizes cryptographic network protocol for operating network services
TCP/IP	Transmission Control Protocol / Internet Protocol - all data on the internet is IP-packages and TCP ensures safe and collective transmission
Endpoint	The end destination for networking communication. Usually displayed as an IP-address with a port number (127.0.0.1:80) or a webpage URL (https://google.com).
URI	A string of characters that unambiguously identifies a particular resource. To guarantee uniformity, all URIs follow a predefined set of syntax rules, but also maintain extensibility through a separately defined hierarchical naming scheme (e.g. *"http://"*). (Wikipedia, 2018a)
VPN	Short for Virtual Private Network.VPN tunneling creates a point-to-point connection between two devices, often the VPN server and your device. Tunneling encapsulates data into standard TCP/IP packets and safely transfers it across the Internet. (ExpressVPN, 2018)

III System Guide

How to run the client

- Make sure you are connected to SDU eduroam or SDU VPN. (SDU, 2018)
- Open NetBeans project **Group9_Semesterproject_E18** and locate the starter package in the client source code.
- Run the "**SYNchatClient**" file.
- Register a new account or directly login with our premade account: as@syn.dk // 12345678 or see@syn.dk // 12345678.
- Login.
- Enjoy the SYNchat experience.
- If any trouble occurs please do contact us by mail (front page).

IV Figure overview

1 Introduction

Throughout history, new technologies of communication have had a profound impact and effect on cultures around the world. The culture of everyday life has become entwined with the Internet. Our everyday lives are filled with technology and gadgets that keep us constantly connected whenever and wherever we are. The Internet and social media have expanded our range and given us the power to reach almost anyone digitally. According to the latest statistics, there were approximately 2.46 billion social network users worldwide in 2017. (Statista, 2018a)

The social network called Facebook is currently the market leader in terms of both reach and scope with around 2.23 billion monthly active users in 2018 (Statista, 2018b). This platform has had a huge impact and has been shaping the social media landscape ever since its launch in 2005. Furthermore, Facebook has been an important factor in discussions about user privacy and also differentiating between the private and public online self.

This project is about developing a chat system called SYNchat. The project is based on the semester and project theme “Design of software systems in a global context”. The main idea behind the project is to develop a free communication platform that makes it easier for people to connect to their acquaintances and also to connect with new people while respecting the users’ privacy. Thus covering the global context by having a platform connecting one user to another, possibly from another culture.

1.1 Problem statement

There are several well-known social platforms such as Facebook/Messenger, Instagram, Twitter, Skype, Discord etc. The aim of this project is to develop a similar platform with familiar features like other social platforms. However, the desired platform should be separated from Facebook as they have a history of massive data collection. Therefore, one of the focus areas that the project team desires to achieve, is to be transparent when it comes to user data. The platform should only store the user data which is critical and necessary for the system to run, for example, login data and basic information about the users.

Specification of problem

- How do we tackle cross cultural management and communication? And how should it be implemented in the system.
- What do we define as user-friendliness and how do we achieve it?
- Why do we use encryption/hashing? And how do we implement it?
- How do we obtain a strong architecture with low coupling and high cohesion?
- What is scalability and how do we obtain it?

1.2 Purpose and goals

The overall purpose and goal of this project are to improve the knowledge and understanding of the project team in both the theoretical and practical parts of the overall semester. The theoretical part consists of gaining knowledge throughout the course lectures, whereas the practical part takes place in coding the project itself. The project team wishes to make use of the different skills obtained in the courses throughout the semester, and combine these into the making of a functional prototype. This ensures a combination of theory and practice and will moreover maximize the learning outcome, which is both the purpose of the project as well as one of the goals of the project team.
Another goal is to develop a distributed software system while also taking into account any cross-cultural management issues related to the development of the system.

1.3 Motivation

One of the motivational factors for the semester project is the freedom of choice regarding the project's topic and coding language as it is open-ended. The group is relatively free to choose the system they want to develop, which naturally causes them to be more motivated to actually work on the project. This also makes it possible to put more focus on the specific subjects that the team finds particularly interesting, such as prioritizing actual code over excessive documentation. Diagrams will still be made but only the ones that actually feel useful and beneficial for the development process according to the group. Additionally, the freedom of coding language will make it possible to get even better at an already known language or try out a whole new language depending on which skills the group decides to strive for.

1.4 Target audience

This paper is written to document the process of developing a chat system for the project of this semester. The target audience for the semester project is the project group itself alongside the group's supervisor and external examiner.

A system like SYNchat would, on a large scale, have a target audience of young adults in the 18 to 34 years of age when looking at Facebook's age statistics on a global scale (Statista, 2018c). The project team has chosen this particular audience as they are more likely to be active on different social media platforms in their everyday life.

2 Methods, planning and processes

In the following section, the development processes and working methods used in the project will be described in details. The sections will first and foremost present how the project team worked during the project but also how they implemented and managed iterative and agile models into the project. Moreover, there will be a specification of the planning of the project and also a detailed approach in the actual processes.

2.1 Methods

The methods used in this project is a combination of Scrum (ScrumBut) with a simplified version of Unified Process (UP). Both Scrum and UP are frameworks for software development. The reason why a simplified version of UP is being used is mainly because of the limited amount of time available for this project. Therefore, the focus has been put on the elaboration and construction phase of UP.

In addition to the two models mentioned, Pair Programming has also been used extensively throughout the project where the teamwork consisted of pairs of two or three.
Pair Programming is an agile software development technique originating from Extreme Programming (XP) which the team has found to be very beneficial. This technique helped to lower errors and mistakes, sharing knowledge by using each other's strengths and lastly to optimize and increase code quality. (Agile Alliance, 2018)

2.1.1 Unified Process

UP is a use-case-driven, architecture-centric, iterative and incremental development process. The image below (Figure 2.1 1) illustrates all elements that can be found in UP. It is seen that UP has 4 stages in which some stages can be split into several iterations. On the y-axis, all the core workflows in UP are listed and following the x-axis is the ideal time spent on each workflow per stage.

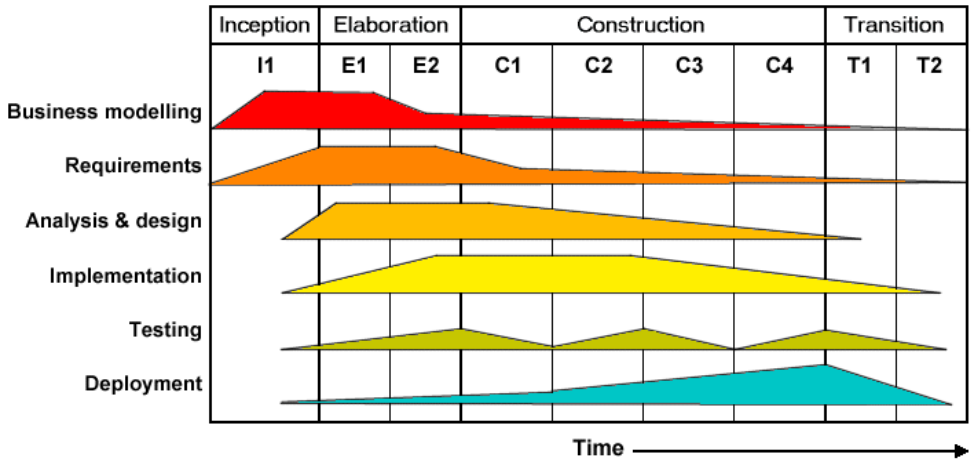


Figure 2.1 1 The image above presents an overview of the Unified Process.

One of the characteristics of UP is its iterative and incremental behavior. As mentioned before, UP has 4 stages in which planned iterations can occur. An iteration is a planned period of which actual development takes place. In each iteration, the following workflows will take place: Requirements, Analysis, Design, Implementation, and Test. After each iteration, there should be an increment in the work containing added or improved functionality compared to the previous iteration.

Secondly, UP is architecture-centric meaning that there is a focus on establishing the fundamental foundation of which the later development should rest on. Architecture is the framework or structure of the system being built, and UP insists that architecture should sit at the heart of the project team's effort to shape the system.

Lastly, use-case-driven refers to the use of use case diagrams in UML (Unified Modelling Language). Use cases are seen as one of the key aspects in UP and are used as a driving force for the development. The project team uses UML diagrams to drive all development work from initial gathering of requirements to code.

Another quality of UP is the focus on risks. Risks are detected and addressed in the earlier stages of the project life cycle. The most critical risks are being reduced by having a good and strong architecture for future development and also by specifying all necessary requirements and functionality for the desired system with use cases as an example. (Informit, 2018), (Si Alhir, 2018).

2.1.2 Scrum & ScrumBut

Scrum is an agile approach for project management with an emphasis on software development. The following aspects are some of the highlighted focuses of Scrum: Continuous improvement, scope flexibility, team input and delivering quality products. A key aspect of Scrum is the agile part, meaning the ability to create and respond to change in order to succeed in an uncertain and turbulent environment. Another characteristic is that Scrum is an iterative and incremental framework for managing the processes in software development.

Following the Scrum model, the project progresses in a series of fixed-length iterations called sprints. Figure 2.1.2 shows the Scrum framework and presents roughly how Scrum is used. The Scrum events being presented are sprint planning, current sprint, daily scrum, sprint review, and sprint retrospective. The Scrum artifacts being presented are product backlog, sprint backlog, the increment, and the burndown chart. The use of Scrum events and artifacts will be explained in details later in Project Process (section 2.3).

The main roles defined in Scrum is the development team, the Scrum Master and the Product Owner. Scrum is facilitated by the Scrum Master, who is responsible for making sure that the Scrum framework is being followed and that the agile process is being kept consistent. Scrum Master is also the one supporting the development team, clearing organizational roadblocks and impediments. The Product Owner is responsible for bridging

the gap between customer, stakeholders, and development team. In addition, the Product Owner is responsible for the backlog and for maximizing the value of what the team delivers.

The development process of each sprint starts with a sprint planning session, where the team gathers for a meeting to discuss the number of items to be committed from product backlog into the sprint backlog, which is a list of tasks to be implemented during the sprint. Then, in the actual sprint, the team prioritizes and works on selected tasks while also keeping a schedule of daily scrum meetings. After each sprint, the team performs a review of what was done in the sprint; each sprint should result in a functional prototype or part of a system being done or improved. There is also the option to perform the sprint retrospective where the team reflects on previous sprints and also how to improve for future sprints. (Cohn, 2018), (Scrum.org, 2018a).

There has been a couple of ScrumButs in this project, meaning that the team could not fully implement and take full advantage of the use of Scrum. First and foremost the team was not able to attend the scrum meetings every day due to the workload of other courses besides the project through the semester. Instead, the team met and worked 3-4 times a week. Secondly, the team did not fully utilize the role of Product Owner, because there were no customers or stakeholders. Also no specific member “owned” the product more than others, and everyone was responsible for the backlog and maximizing the value of the product. Lastly, the role as a Scrum Master was not decided for one specific member but instead alternated between the members on a weekly basis. (Scrum.org, 2018b)

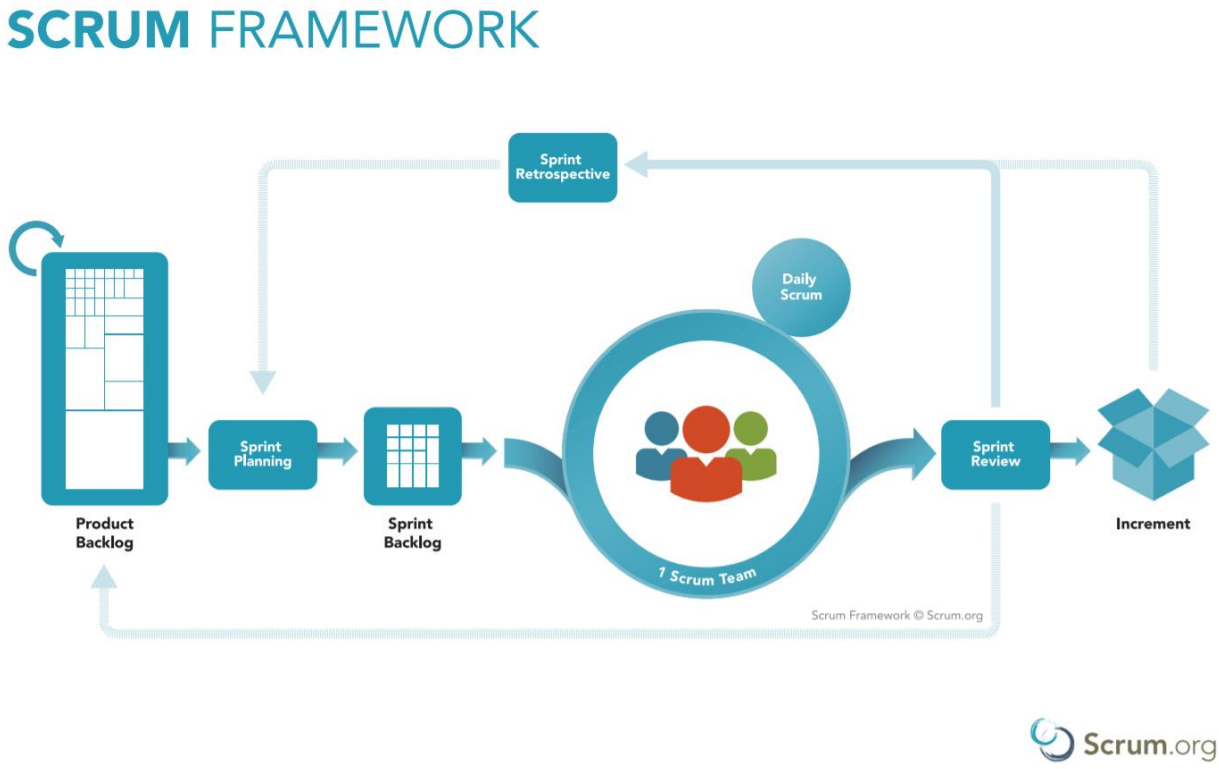


Figure 2.1 2 The image above presents the Scrum Framework and all the artifacts.

2.2 Planning

An essential part of this project is to schedule time for planning. Firstly, before starting each sprint, it is important to gather all members before any work is done, in order to plan the actual sprint. The team visualizes the Scrum artifacts in a Scrum Board by using schemas in Google Sheets from Google Drive (A3 Scrum Board) (Steardahl et al., 2018a). Before starting any work the project team defines every single task to be made and puts these into the product backlog. At this point, the group uses the prioritizing method called MoSCoW to prioritize each and every task defined (described later in section 3.2.1). These tasks will then be time estimated and further prioritized in the release backlog before each sprint. The planning in each sprint then consists of splitting each task into pair programming groups where the actual work begins. After each sprint, all members will gather again to perform the sprint review of everything that has been accomplished during the sprint. The review may contain a demonstration of the increment and also has the purpose of keeping every member up to date of what has been done in the respective sprint, and the project overall. Then, the group performs a sprint retrospective to reflect on both good and bad events in the previous sprints, and also to discuss what could be improved or be done differently for future sprints.

2.3 Project process

The project is divided into five stages: Idea stage, problem analysis stage, problem processing and dissemination stage, completion stage, evaluation and reflection stage.

The first stage, the idea stage, consisted of forming the project team and then to formulate an idea based on the project theme and topic. The result of the first stage is to make a project proposal which describes all the first thoughts and ideas for the project and formulates an outline of the first problem statements.

Secondly, during the problem analysis stage, a project foundation is established based on studies and analyses, and the final problem statement is formulated. At this point, the team defines the project scope and outlines the system architecture. This stage should establish a strong foundation, an architecture candidate, for the later work and development. The first and second stage can be seen as the inception phase from UP, where the project team gathers information and establishes a case for the viability of the proposed system. After this stage, the group will also discuss and eventually form the full product backlog with every single task for the desired system.

The third (processing and dissemination) and fourth (completion) stages are where the actual work in the project is made, through a combination of the methods UP and Scrum. These stages include most of the elaboration and construction stages of UP. We go from a tentative architecture to establishing the final architecture in the elaboration. Also, the highest risk elements are reduced or eliminated. The outcome of the elaboration is a system

with the most central functionality implemented. Under the construction stage, most of the considered features are implemented and tested. The outcome from this stage is a more or less operational system. During these stages, iterations are planned in scrum-modeled sprints where the workflow consists of analyses, design, implementation, and tests. As mentioned before the processes start at the Product Backlog (A4 Product backlog). All tasks are defined and put into categories, statuses, and prioritization. Before each sprint, the selected tasks from the backlog is put into the Release Backlog (A5 Release backlog). This backlog is similar to the product backlog, only with the selected tasks and their time estimation. During each sprint, the group works in pairs for implementing the selected tasks which are put into the corresponding Sprint Backlog (A6 Sprint backlog). In the Sprint Backlog, the team selects the most prioritized tasks to implement. As the sprint progresses the estimated time will decrease for every task which can be visualized with the scrum artifact: Burn down chart (A7 Burn down chart). After each sprint, the group will perform the sprint reviews and retrospectives.

The fifth and last stage includes both evaluation and reflection upon the project and semester as a whole. The evaluation part happens in the form of an exam after the project period has ended. The reflection part is achieved by the individual students' reflection upon both the project process and the overall semester. The end result is described in a follow-up document which should be handed in before the exam. It is after this stage that the transition stage from UP will occur. The transition is about deploying the final product to the market to possible customers or target users, but it is not meant for the project to get to this point.

3 Software requirements specification

Requirements are very important in software development. If they are not phrased in a very specific way, it can be easy to misinterpret how the system should be built. The intend to build a chat system is too broad of a concept to work with - it needs to be narrowed down. We need to specify the requirements of the system we are trying to develop, such as whether this chat system is based on communication between just two people, or if it should be more of a lobby chat, where hundreds of people can be chatting with each other at the same time. Both of these would be examples of a chat system, but in order to make sure we end up developing the product we actually want to make, these requirements have to be clearly defined. Most of the use case diagrams corresponding to the use case descriptions can be found in the appendix A8 Use Cases.

3.1 Overall requirements

In order to set up the requirements for the system, we have described several use cases. These use cases showcase what the system should be able to do, and therefore it will help emphasize the specific functionality that needs to be included.

The first use case is *LoginScreen*, which covers the basic functionality of the system when a user first opens the program - they see a screen on which they can log in with an existing user, or create a new one if they are not already registered within the system.

LoginScreen (Actor: User that is not logged in)
• Create User • Login

Table 3.1.1: Features a visitor will be able to access on the very first screen.

ProfileAdministration is a use case for describing what a user can do when they are logged in, to modify or otherwise interact with their own profile. This defines the possibility to delete the user, if the user no longer wants to be a part of the system. It also includes the functionality to be able to log out, or change the info that is connected to the user, such as the password.

ProfileAdministration (Actor: User that is logged in)
• Delete User (self) • Change info Change user-related info such as password and nationality. • Logout

Table 3.1.2: Features with which a user may interact with its own account.

The use case *AdminCommands* is for defining the additional functionality an admin should possess. An admin is also a user, so it would have access to the same things that a basic

user would - but also the extra functions described here. These include banning users, processing reports, contacting users directly, and moderating public chats.

AdminCommands (Actor: User that is logged in as admin)
● Ban user A user violated terms of use → Timeout or ban options. ● Process reports User was reported multiple times → Admin gains access to see the user's public chat messages. ● Contact user Admins may contact all users, even those who are not on their friends list. ● Manage public chat Admin commands, for things such as removing problematic content in chat rooms.

Table 3.1.3: Features only admins can access.

The use case *Chat* specifies the functionality that could be accessible to a user that is logged in and trying to use the actual chat in the system. This includes some features that would be very nice to have, but which are not at all necessary at this point in the development - such as minigames, a translation module, GIF's, emojis, and sound clips. The system will be able to do fine without these features, but they could be added later if the available resources and time allows it. This use case also contains the obvious chat-functions like sending a message and using chat commands such as muting other users in a chat.

Chat (Actor: User that is logged in)
● Minigames A user can initiate a minigame in the chat. ● Gifs ● Emojis ● Sound A user can send small files with sound (speaking out your message instead of writing). ● Send message (with text and/or pictures) ● Chat commands (mute, etc.) ● Translation module The possibility to translate a chat-message to match the language of the recipient.

Table 3.1.4: Features in the actual chat system.

System is an overarching use case for things that should be implemented for a user that is logged in to the system, and trying to interact with different things in the system. This includes entering or creating different chat lobbies, reporting other users for malicious use of the system, and administering the user's own friend list. Furthermore, it should be possible to search for other users based on username and view these other users profile - as well as ignore them, if the user wants to do so.

System (Actor: User that is logged in)
● Public chat A user can browse and enter open chat rooms, for chatting with multiple users. ● Private chat (by user) A user can create a new chat with a friend - a user on his/her friendlist.

- Report user
  A user can report another user for malicious use of the system in group chats.
- Ignore user
  A user may ignore another, blocking that user from sending messages to them.
- <<Include>> ProfileAdministration
- Search for users
  Search among the systems users, to see their profiles and/or add them to friend list
- Administrate friends
  A user may add people to their friend list, and have the option to remove them again.
- View user profile/avatar
  Users can see profiles of other users, to get an idea of who they may be chatting with.

Table 3.1.5: Features of the overall system, accessible to all users and admins.

The corresponding use case diagram for the System use case can be seen below. As mentioned earlier, the rest of the use case diagrams related to the use case descriptions can be found in appendix A8 Use cases.

3.1.1 Supplementary requirements - FURPS+

Use cases help describe the functionality of the system where all functional requirements are specified, but it is also necessary to define non-functional requirements. This can be done with the FURPS+ model. This model includes constraints and other quality attributes of the program that isn't necessarily a direct functionality. These are divided into the following five categories. (Dhami, 2014)

Functionality
These requirements cover the security, capability, and reusability of the system.

- Encryption
  Encrypted communication between client and server, and on the SQL server - this includes the passwords and chat logs for private messages.
- Password protected profiles
- Client-server communication

Usability
Requirements under this point covers human factors, consistency, and documentation for the system.

- User-friendly interface
  Prototypes of the program will be tested via. user feedback to measure how user-friendly the design is and changes can then be made accordingly.
- Interface changes according to nationality
  Welcome message and possible background change based on the nationality set by the user at profile creation.

Reliability
This point covers the accuracy, availability, and predictability of the system.

- Minimal data loss
  Neither accounts nor private messages may be lost.

Performance
Requirements under performance define efficiency, speed and resource consumption.

- Lightweight program (plug & play)

- Receive messages within a short timeframe
  Messages should be delivered within a second.

Supportability
These requirements describe adaptability, flexibility, and testability.

- Cross-platform
  The system should be supported both as a desktop application (Java application), a mobile app and an in-browser adaptation.

3.2 Specific requirements

The overall idea of the system has now been described through use cases and FURPS+. With these general guidelines in place, more specific requirements can be defined, and these can then be prioritized to show what will be the focus of this project, and which features would be nice to have, but will likely not be a part of the early system.

3.2.1 MoSCoW

For prioritization in this project, the MoSCoW model will be used. This model distinguishes between four different types of requirements; those that **M**ust be included in the system, those that **S**hould be, some that **C**ould be, and the ones that **W**on't be. From the overall requirements described in previous sections, the group has divided the specifically defined requirements into the four categories in the table shown below.

Prioritization of requirements	Semantic
Must have	**Mandatory requirements** ● Interface changes according to nationality ● Login/logout ● Send message (with text and possibly pictures) ● Create user ● Search for users ● Encryption of passwords ● Password protected profiles ● Client-server communication
Should have	**Important requirements** ● Multiple user communication (chat lobby / group chat) ● Change info ● Administrate friends ● View user profile / avatar ● Send images ● Emojis ● User friendly interface ● Receive messages within short time frame ● Enter chat lobby ● Encryption between server-client communication
Could have	**Optional requirements** ● Delete user (self) ● Gifs ● Report user ● Chat commands (mute etc.) ● Minimal data loss

	● Lightweight program (plug & play) ● Create chat lobby (by user)
Won't have	**Inconsequential requirements** ● Ignore user ● Ban user (as admin) ● Process reports (as admin) ● Manage public chat (as admin) ● Minigames ● Sound ● Cross-platform ● Contact users (as admin) ● Translation module

Table 3.2.1: Presents all the prioritized requirements.

As seen in the table above, the group has decided to focus on the most vital functions of the chat system, such as creating users, logging in, and actually sending messages between them. These requirements are put in the **M**ust-category, since they will be the backbone of the system - without them, it does not really qualify as being a chat system.

There are other feature that should be in the system, but are not absolutely critical - these are things like sending images, viewing other user profiles and administrating friends. These are categorized as **S**hould.

The **C**ould-category contains even less important features - in the final system it would definitely be an expected feature to be able to delete ones user in the system, but it is not important to the functionality of the system early in development.

The last category, **W**on't, are for all the features that will not be implemented during this stage of development, but which would be nice to include in the final system as a form of extra polish. There is no need to implement these things yet, but it may be beneficial to code the system in a way that supports adding these features at a later stage. An example of this would be preparing the basic message-class for also being able to handle the possibility of sending sound clips, in case that feature is added to the system later on.

4 Analysis

During the early stages of the project, the team worked through an analysis of the system that we wanted to create. This was done by analysing the artifacts from the previous workflows, identifying classes, methods, and interactions that define the system and apply them in an object-oriented programing scope. The resulting analysis class diagram is shown below in Appendix A9.

The process resulted in a system revolving around the individual users being represented by their core insystem attributes such as *userID*. Those users are affiliated with objects representing their outwards appearance in the system (*Profile*) and other similar related artifacts such as *Friends*.
Concerning the chat part of the system it was decided to share a *Chat* between two or more users that contain a *ChatHistory* which include the *Message* object sent from one of the *Users* affiliated with the *Chat*. At this point the difference between *LobbyChat* (later *PublicChat*) and *PrivateChat* was based on different derivatives of the *Chat* parent object.
During the process of fully analysing the system a small amount of design decisions crept into the analysis workflow such as the decision to implement *Message* as an abstract class, that could cover multiple types of messages. This way it would be easier to add the option of sending other types of messages later on, however, only the *TextMessage* class, which just contains a generic string, got implemented.

5 Design

Throughout the project, a considerable amount of design choices have come up, all of which build upon the earlier analysis of the system. During the project, the team has discussed the different choices as they came up so that the development could continue smoothly. It has been very important that these design choices were agreed upon so that both the client-side and server-side would be able to handle the connection between them without conflicts.
In this section, the design choices made in relation to the client and the server throughout the process, will be described separately.

5.1 Software architecture

Software architecture has an essential role in the project, as it is the main structure and foundation for the whole system. Thus it has been a significant factor on which architecture to use and how to implement it into the project. The approach to deciding the desired structure was through a combination of the *Top-Down* and *Bottom-Up* design strategies (Tutorialspoint, 2018a). The Top-Down approach first takes the whole system as an entity and then iteratively decomposes it into smaller components or subsystems, to finally create a hierarchical structure in the system. The Bottom-Up approach starts at the center of the system with the most specific and basic components and builds upwards into higher level components. As an example of the Bottom-Up, the project starts around the Client-Server components, as they are the most central part of the system. The goal of combining these

approaches is among others to obtain a loose coupling and strong cohesion. To achieve this, a 3-tiered server-client architecture is utilized. This includes the client(1), server(2) and the database(3), to form 3 tiers. In addition to this, the client and server are both using layered architecture. These architectural choices ensure the low coupling and high cohesion.

5.1.1 Layered architecture

The selected software architecture was the *layered architecture*, as it is the most commonly used architecture pattern and as it separates the code in a logic and manageable way (Richards, 2015). The different layers are illustrated in the image below and are classified as Presentation, Business, and Persistence.

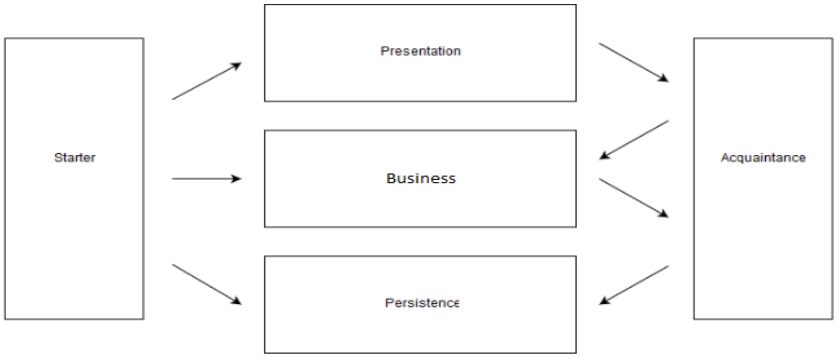


Figure 5.1 1 The image illustrates the layered architecture.

System components are divided and organized into *n* layers, with a specific role and responsibility within the system. The separation is a major advantage as to security and how data is being accessed. The interfaces or contracts are located in the acquaintance package, and they make sure that there are no direct calls down through the layers. The starter package is responsible for connecting the layers and starting the application.

5.1.2 Client-Server

The whole project has been focused around the *3-tier Client-Server architecture* as center point from the start, as it is a project requirement to develop a distributed system. The Client-Server architecture is simply a system split into three parts, one or multiple client(s), a server as seen in the image below and in this case the third part is an external database, thus the 3-tier architecture.

The client is responsible for the graphical view, the user interaction, and the communication to or from the server. The server is responsible for processing requests from the client(s), doing different procedures or operations for example storing or editing information or data and sending and receiving information to or from clients. Clients are “active” such that they are the ones taking the initiative and sends requests to the server. On the other hand, servers are “reactive” and reacts to all requests from the clients. The server processes these requests and sends an answer back to the respective clients. (Oracle, 1996)

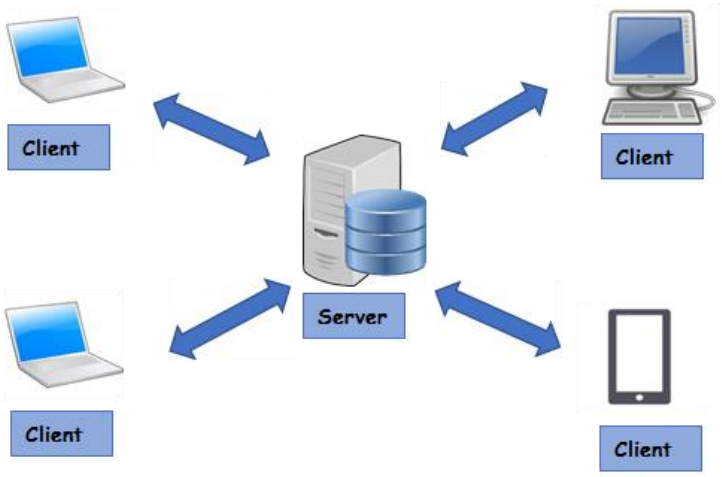


Figure 5.1 2The image illustrates the client-server architecture.

The communication of which clients connects to servers is through *sockets*. A server is normally run on a specific computer, which has a socket bound to a specific port number. The server then waits and listens to its socket for clients who want to connect through a *connection request*. The illustration is shown in the figure 5.1.3. When the client makes the connection request (image A), it is required to know the hostname and port number of the machine on which the server is running. It is also required that the client identifies itself through a local port number that will be bound during the connection. If everything run smoothly and without errors, the connection gets accepted. The server now gets a new socket bound to the same local port and has its remote endpoints set to the address and port of the connected client (image B). The server then has two sockets, the one for originally listening to connection requests and the new socket for tending to the needs of connected clients. (Oracle, 2017)

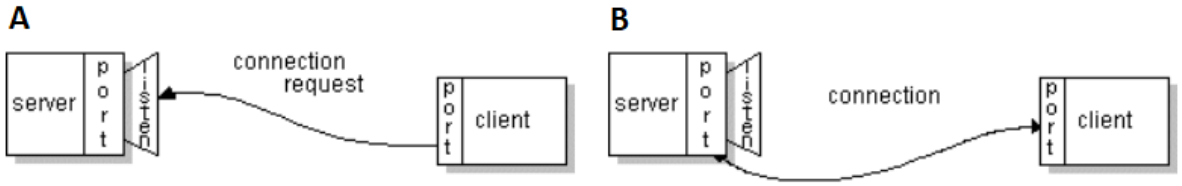


Figure 5.1 3 The image illustrates the connection process between client and server(Oracle, 2017).

5.2 Design class diagram

After forming our 3-tier architecture, our design class diagram had to be split into two diagrams. One for the client-side and the other for the server. First and foremost, the design class diagram got expanded extensively from the analysis diagram, by implementing the layered architecture. As the layered architecture avoids direct dependencies, facade classes were inserted to ensure low coupling. Another significant change from analysis to design was dividing the *user* class into having a separate class for login information, *login* class, which would ensure more security regarding login information. Also a users' nationality was decided to be moved to a separate Enumerator class. Because of the size of the current

diagrams, the design class diagrams can be seen in the appendix A10 Design class diagram.

5.3 Security

When dealing with a system that handles sensitive information, it is important to have security measures in place to ensure that the user and its data is safe when using the service. This is obtained by encryption and hashing.

5.3.1 Hashing

To best secure the users' login information it has been decided to obfuscate this information using the hashing algorithm called MD5 (Message-Digest 5).
MD5 takes all inputs and hashes them to a 128-bit hashcode no matter the size of the original input. Hashing is a mathematical way of converting data to a value that is unintelligible (Techopedia Inc., 2018). The MD5 hashing is practically irreversible, which means that once the data has been hashed the data is irretrievable. The key point is that hashing the same value multiple times returns the same hash value and therefore a possibility for a system to check if the hashed password stored in a database matches the hashed password just entered by the user when trying to log in.

5.3.2 Atbash

MD5 is a free way to securely store information but it is not possible to revert the hashing. Therefore the group has decided to use a more simplistic method for encrypting the chat messages as these can be private messages between two friends.
In short Atbash has an array containing all the letters of the alphabet in order (In this case we have the double amount in order to have uppercase and lowercase letters included as well).
This cipher only uses the English alphabet which means that languages with special characters like Germany, Sweden or Denmark can still use them when sending messages but they will not be ciphered before sending through the server. The same principle applies to characters like '*?*' and '*!*' and so on.

A	B	C	D	E	F	G	H	I	J	K	L	M
Z	Y	X	W	V	U	T	S	R	Q	P	O	N

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
M	L	K	J	I	H	G	F	E	D	C	B	A

5.3 1 The image illustrates the encryption using Atbash.

The mathematical representation of Atbash is somewhat simple. The alphabet is seen as circular, which means the operator "modulo" can be utilized to ensure the circularity.

$$f(x) = ((m-1) * x + (m-1)) \, mod \, m, where \, m \in 0..m-1$$

The letter that should be converted with Atbash, will be inserted into the function as x. In addition, m is the symbolization of the alphabet. The equation can be further simplified:

$$f(x) = (m-1) * (m+1) \, mod \, m$$
$$f(x) = -(x+1) \, mod \, m$$

For visual representation, a schema for the results for the Atbash cipher has been added (Figure 5.3.1). That is why Atbash is a good way of encrypting messages for later decryption because it can both encrypt and decrypt a message within the same exact method.
The disadvantage is that Atbash is not in any way secure as it is very easy to find the pattern for deciphering it. Despite Atbash being so easy to decipher, the group decided to use Atbash as a proof of concept.
The cipher uses an Enumerator containing all letters of the alphabet in switching uppercase and lowercase (i.e. “A, a, B, b, C, c…”). When encrypting a message the cipher takes each character of the message and run through the Enumerator to find out which number in the Enumerator the character is. When the number is found the cipher subtracts the character number from the total amount of letters in the Enumerator thus finding a “random” character to replace the original character with.

5.4 Client

The following section provides a general overview of how the client application of the SYNchat system implements its layered architecture and an overview of the individual layers

5.4.1 Presentation layer

Throughout the development of the system the non-functional requirement, “usability”, has been the main focus. The intuitiveness of the system should be so good, that the user should not need any instructions or tutorials on how to use the system. By following the KISS principle (Hanik, 2018) the GUI is kept simple and manageable, and as shown in the images of the system (appendix A12 System images), the design is carried through the entire system, hereby achieving the desired intuitiveness. The presentation layer is separated in several *fxml* files and controllers that all controls the GUI depending on which stage is active. In common they have the *PresentationFacade* class, which ensures communication with the business layer and stores data from the server which can be used by the controller classes. Especially the *SYNchatController* class is important, as this controller serves the purpose of handling user requests when chatting, which is the core functionality of the system, besides this specific controller also contains most of the logic in the presentations layer.

5.4.2 Business layer

The business layer contains the logic of the entire client-side system. This layer sends and receives data from the connection layer, and ensures the data is dealt with accordingly before sending it to the presentation layer.
Inside the business layer, classes such as “*Login*” and “*User*” contains information about the user. *Login* handles the entire login process, by first receiving login information in form of an email and password, next up it hashes the information and sends it as a login object to the server. *User* contains all the information about the user, such as an *Friend* class with a

friends list, a *Profile* class with information about *firstname*, *nationality* etc. and also an *UserID* which is used to differentiate between users in SYNchat.
In addition the business layer also keeps a *Management* class which holds an "*action*" integer, that determines which request should be made on the server, examples of actions could be to update profile information.
Finally the *ClientSystem* class updates the entire system with information of which users are logged in, in addition to handling all important business logic.

5.4.3 Connection layer

The third layer in our three-layered architecture on the client-side is the connection layer. This layer serves as the communication bridge between the client's business and server's. First and foremost, all communication happens by sending objects through sockets.

While using the public chat, the connection layer will keep checking for objects coming from the server. If the object from the server is a *ConTextMessage*, this object will be cast to a corresponding class object on the client and sent to the *ConnectionFacade* and sent from here further into the system. Likewise, will other public chat compatible objects be treated.

Since the logic on the server-side and client-side is different, shared classes between client and server need to be treated differently depending on which side they are on. Therefore all class objects in the connection layer both on server and client side are labeled with the prefix "*Con*" when the corresponding object is sent to the business layer, it will then be remade as a new class that serves the logic that is either on client or server side.

5.5 Server

Building on from the client the following section elaborates on the same concepts but within the context of the server application.

5.5.1 Connection layer

The servers connection layer uses the principles described in section 5.1.2, where the server has a socket open listening for incoming client requests to connect. This functionality is located in the *Server* class. When a connection has been established, the server will start a new thread for the client, which will be exclusive to that client. To keep track of all the alive clients, a class called *ClientHandler* is created. It keeps a reference to all the connected clients, together with their connection. The Servers thread will, however, continue to run concurrently to listen for new clients.
The protocol used to communicate between the server and client is TCP, mainly because it's ordered, reliable and error-checked, which is crucial for our system.
The connection layer also facilitates the translation process of the input from the client, which ensures only correct input is treated by the system, and is treated correctly.
For the server and client to be able to communicate with objects, a uniform connection class system must be used. This means that all objects being sent from the client to the server,

must have the exact same format in the connection layer. To meet this requirement, an interface implementation is made uniform between the client and the server, labeled “Con”-*classname*.

5.5.2 Business layer

The business layer handles the logic of the server. Whenever a client has sent a request to the server, and it has been interpreted by the *ClientHandler*, the Business layer will perform the required actions to the data.

5.5.3 Persistence layer

The servers persistence layer facilitate the communication with the SQL-database. This communication is through SQL-queries, which allows the server the retrieve and insert data into the database. A design choice made in the persistence layer was to use objects the moment data was loaded in. Not only does it increase security instead of just storing strings and send them between the layers, but it also contributes to high cohesion as well. To distinguish classes from the business layer and the persistence layer, the classes in the persistence layer are labeled “Per”-*classname*.

5.6 Database

This section seeks to illuminate the design choices for the SQL database made for the SYNchat system. It covers the distinct technologies and resources chosen for the project which links to a closer examination of how the actual design of the database is made.

5.6.1 PostgreSQL database

For this project, the team has chosen to utilize a PostgreSQL database made available by SDU. PgAdmin has been used to access and modify the database (appendix A12 pgAdmin), to suit the needs of the project. This was the obvious solution for data storage in this project, as the team has learned how to create and manage a PostgreSQL database on the 2nd semester of the education.
PostgreSQL is an open source object-relational database system(The PostgreSQL Global Development Group, 2018). The aim of PostgreSQL is to extend the SQL language and make it very user friendly and intuitive while still having security and a lot of features for creating both large and small databases.
Using the SDU servers for the PostgreSQL database requires the user of the system to either be logged in to the eduroam network on SDU property or to enable the SDU VPN (Cisco Proxy tool) as all access to the SDU servers are protected from all outside unauthorized access.

5.6.2 E/R diagram

The E/R diagram (Appendix 11) is a visual representation of the entities and relationships within a database. In the database for this project, there are not a lot of entities, but the

relationships between these entities are crucial to how the rest of the system is built. The choice was made to have two standard entities and two weak entities.
Since a *profile* has a one-to-one relationship with a *user* and it is not possible to identify a specific *profile* from the attributes on the *profile* entity, the team decided to put the *user_id* from the *users* entity into the *profile*, to make sure that it is possible to always find a specific *profile*, belonging to a *user*.
The other weak entity set is the set consisting of *chat_info* and *chat_messages*. *Chat_info* is the overview of the users' current *chats* while the *chat_messages* are the actual messages being sent in the chats. To prevent the *chat_info* entity from bloating too much, the two entities have been made into a weak entity set. In practice, this means that the key *chat_id* from the *chat_info* entity is placed in the *chat_messages* entity, to ensure the connection between the two.

5.7 Distribution

This section seeks to examine the implication that the implementation of the client-server architecture will have on the distribution of the SYNchat system. As a consequence of the client-server architecture the system requires a plan for both delivering the client to the end user and feeding the server application in a reliable and consistent manner to ensure the best possible availability for the users of the system. Firstly, different technologies for distributing software will be introduced. Thereafter the plan for the distribution of the SYNchat system is illustrated. The container technology considered in this project is Docker.

To understand the choices made regarding how to distribute the system it is essential to have a rudimentary understanding of the two primary distribution technologies discussed for the system. Those being virtual machines (VM) and containers using Docker.
The first way to run a program is to run it natively in an operating system, in the case of the client this would be on the user's computer on that computer's operating system. However, it does not work in the exactly same way on the server side. It is possible to run on a personal computer, but the resources are scarcer and the computer will always have to be turned on. Alternatively, this can be mitigated by running the program on specified server hardware. Despite it being possible to run an application directly on the server OS, that is often not the case for a number reasons, mainly being that it would allow all applications on the server to interact freely which would cause a major security concern for the individual application. Therefore, most server space is available as an implementation of a virtual machine running on the hypervisor. This allows each VM to be separated from everything else on the server and it makes it possible to offer a range of different services to the users of the same server hardware.

To elaborate, the hypervisor is a software layer that is required to enable virtualization by creating the environment that the VM operates on and supervises the resources available to the guest VM's. There are two types of hypervisors. Type 1 called native hypervisors which run directly on the server hardware, are the most common seen in server virtualization. Type

2 hypervisors requires a host operating system to run (Graziano, 2011). The benefits and specifics of each hypervisor type will not be covered further in this paper.
Though having been the de facto solution to safely utilize the available server hardware this method does create a considerable amount of overhead, due to the stacking of OS's on top of the hypervisor.
In terms of each application running on the VM having their own OS which run on top of the servers own infrastructure, this is visualized on the left side of figure 5.7.1.

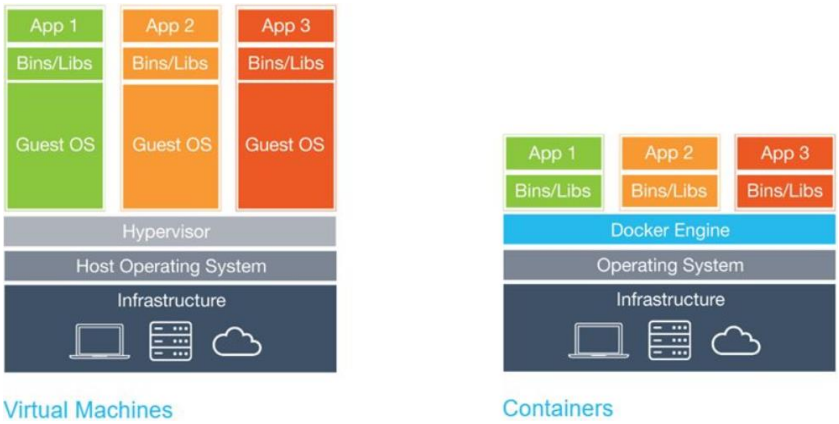


Figure 5.7 1 Visualization of the overhead created by type 2 VM and Docker containers (Neerup et al., 2018a).

However a way to resolve this overhead is to containerize the application. A container is a self-contained and self-sufficient package of software around the application that contains a virtualization of the operating system and the dependencies for the application. This results in a single executable package that can be run on any platform that runs the required platform (eg. Docker engine). The benefits of containers are among others that they can be run with considerably smaller amounts of overhead in a server infrastructure. Since the containers themselves are isolated it is possible to run them in a server's native OS running the Docker Engine eliminating the requirements of a VM. This is visualized in the right side of figure 5.7.1. Furthermore since the containers are self-sufficient with their own required OS, system libraries and settings run the same regardless of the underlying OS. Another benefit by containerizing an application is the improvement to portability and maintenance. Instead of having to deploy and update the software on a VM by VM basis, containers allow for fast image deployment since it comes with all dependencies bundled, disposes of old containers when updated and creates new containers from the updated image. (Docker Inc, 2018a)

The initial plan for deployment of the system was to containerize the client and run the server in a windows VM. This would allow for easy distribution of the client through Docker Hub but on the other hand, it would limit the user base considerably since the use of Docker is relatively unknown to the general public. Therefore it was decided to distribute the client as a *.jar* file.
To ease the load on the server and to facilitate scalability the server application is to be containerized and deployed on suitable hardware.

6 Implementation

To further understand the SYNchat system the following chapter seeks to illustrate how the project group transformed the design into a fully fledged prototype and all of the significant decisions taken along the way.
Firstly the chapter review the architecture of the implementation which leads on to the specific implementation details of each of the layers in the client and the server applications, thereafter it examines the implementations specifics concerning the security features of synchat. Finally it focuses on how the system was distributed.

Throughout the different layers of the system, different implementations of the interfaces occurs. The persistence layer labels its implementation of the interfaces with the prefix *Per*, where the connection layer labels them the prefix *Con*. To communicate between the client and server, the uniform *con* classes is a must. Therefore, constructors throughout the different layers will take their respective interface as parameter, which will then create the correct implementation based on the layer it is in.

6.1 Architecture

The current distribution of the system is based on the aforementioned 3-tier architecture, where the clients running on the users' personal machines connect to an application server socket through TCP/IP and the application server connects to the database server through an API called JDBC, which is a Java DataBase Connectivity API. The server is running in a Docker container on a Linux server offered internally at SDU.
As the client, application server and database server are running on different physical machines, it is a 3-tiered architecture. Had the system utilized an embedded database such as SQLite the system would be built in a 2-tiered architecture, as it would only be run on two separate physical machines.

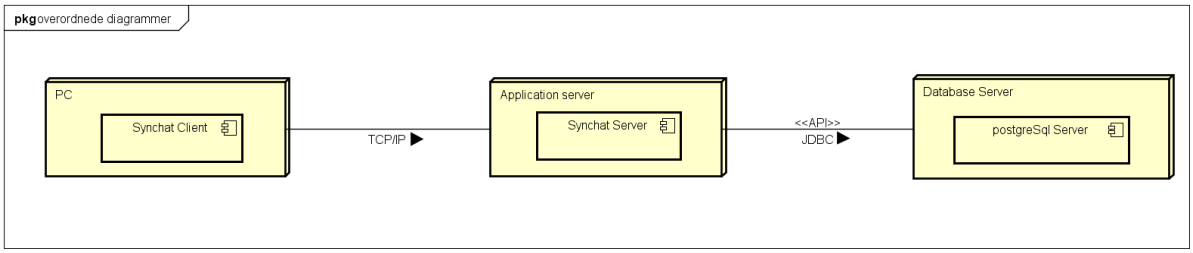


Figure 6.1 1 SYNchat deployment diagram.

6.2 Client

6.2.1 Presentation Layer

The presentation layer consists of a facade class and several fxml controller classes. The controller classes serves as translator from the user to the code, by mapping user actions to code that handles the request, as previously mentioned in section 5.4.1.

PresentationFacade

The *PresentationFacade* class communicates with the business layer through interfaces in the *acquaintance* package. Most method calls to the business layer returns some kind of data, so that it can be used in the *GUI* to visually represent success when carrying out requests. Besides communicating with the business layer, the presentation layer also updates its attributes like *IMessage*, *IUser* and a Map everytime a new message is sent or a new user enters the public chat. The reason to keep these kind of attributes stored in the presentation layer, is to access them from the controllers, which will make it possible to keep information from other users when chatting, displaying online users and looking at other users profiles.

SYNchatController: startRun()

The *startRun()* method in the *SYNchatController* class is one the most essential methods, as this method controls the public chat. When this method is called it loads the currently chatting users into a local "*comparisonMap*", next up a loop runs while you are in public chat that constantly checks if the *IMessage* or the map in the *PresentationFacade* has changed, if so, methods that handle the displaying of messages or listing of users will be called correspondingly, and this is how chatting works in SYNchat. For instance when receiving a new map with users from the server after a new user has joined, the *PresentationFacade* now have the most updated map, and the *comparisonMap* is missing the new user that just joined. The *startRun()* method then notices that the size of the two maps is different and if the map from the server is bigger than the *comparisonMap*, then the method will find the new user and add him to the list of users and the *comparisonMap*. Vice versa if the *comparisonMap* is bigger than the server map, then someone left the chat and that user needs to be removed.

ChangeInfoController: changeProfile()

The *changeProfile()* method in the *ChangeInfoController* is quite simple but can seem very complex at first. To break it down, this method serves as a checkpoint to see which fields the user is trying to update. First and foremost all profile fields are loaded with the data from the user's current profile. Next up it checks which fields is updated and overrides these fields. Next the method figures out if you are trying to update only profile information or if you are also trying to update your email or password (only either email or password can be updated at a time). If you are only trying to update profile information, you can do that immediately since this is not any sensitive data. However if you are trying to update either password or email, a "security" check is made first, in form of the user typing in the current password for the account, whereafter the server verifies if the password is correct, before the user can update the accounts email or password.

6.2.2 Business Layer

The business layer handles and processes all data from the server or the *GUI*. For instance when registering a new user, information about the user is received from the presentation layer, then a profile object is created with that information, along with a new user object, that

is finally placed upon a login object that holds the hashed email and password. Afterwards the login object is sent to the server for validation.
This way of receiving data and repackaging to other class objects, before sending it further to other layers, is how most of the methods function in the business layer, though it may be simple it is very important, otherwise SYNchat would not be able to store actions and data from the presentation layer.
Since SYNchat is a chat application, sending messages is a key feature. The Message class is an abstract super class that defines all types of messages in the system. The reason to use inheritance when creating messages, is due to the fact that messages share common information, such as a timestamp and a senderID to know which user sends the message. The abstract method "*getContext()*" will in subclasses receive whatever *context* that message have, for now only a *TextMessage* class is developed, but with the development of different kinds of messages, the abstract class *Message* would serve its purpose properly.

6.2.3 Connection Layer

Client: startPublicThreads()

The connection layer handles the communication with the server through sockets. The most interesting method in the connection layer is probably the "*startPublicThreads()*", since this method handles the actual chatting in the chat application.
First and foremost, the method creates a new *Thread* with a new *Runnable* since the chat should be running on another thread, so that other system actions is still possible to be made while chatting, next a loop is entered that continuously reads from the *ObjectInputStream*. If the received object is an instance of either a map or the *ConTextMessage* or *IUser* class, this object will be cast to that corresponding type and sent to the connection layers facade class to be handled. Finally if the user decides to stop public chatting, the method will break out of the loop, and reach the "finally" statement, where the thread is interrupted.

6.3 Server

6.3.1 Connection Layer

The Server is created taking a port and an IP-address. In the servers starter code, the IP address will be seen as 0.0.0.0, which listens to all requests on the machines IP. In addition, we decided to open port 8080 since its a port made for alternative web servers.
Looking at the *Server.class* in the Servers connection layer, the constructor calls the method *createSocket()*, and afterwards *acceptClient()*.
acceptClient() is a method of interest, since it listens for new clients trying to establish a connection. Whenever a client has established a connection, the *ClientHandler* class is instantiated with a reference to the clients socket.

ClientHandler

The *ClientHandler* has a static hashmap, containing a *userID* and a client socket. This is made because it is desired to keep track of all active clients.
The *ClientHandler* extends *Thread* since multiple clients should be able to use the server at once, therefore the *ClientHandler* has a run method, to facilitate the clients requests.
The run method is run as long as the connection is established, and whenever the client is disconnected, it is ensured that they are removed from the maps throughout the system.
To interact with the server, you must correctly login. For the client to login, they must send a login object. Therefore, the run method listens for a login object, and checks whether or not the user is trying to login or register a user. To handle this, a method called *checkLogin()* is created. *checkLogin()* will see if the login object contains a user, and if it does not, the client is trying to login. A call throughout the system will then be started, and if the client successfully logged in, he/she will be returned a login object with all his/her information attached. However, if the client did not manage to login, he/she will be returned the login object with an error code attached. If the client tried to register a user, a boolean will be returned according to the success.

When the client has logged in, the run method will check every object sent by the client, and call the method *readStream()*, to interpret the received object.

readStream() takes an *Object* as parameter and checks what type that object is.

Object type	**Action required**
IMessage	Broadcast the message to all clients in the public chat
IManagement	The client is trying to update their profile
IFriends	The client is either adding or removing a friend
String	The client either logged out or entered or left the public chat

Based on the objects type, a call to the *ConnectionFacade* will start a chain of operations throughout the system to complete the action.

ClientHandler: sendPublicMessage()

The method *sendPublicMessage()* takes an *IMessage* as parameter and sends the *IMessage* to all the clients subscribed to the public chat.

ClientHandler: sendPublicChatUser()

The method *sendPublicChatUser()* takes an *IUser* as parameter and sends the *IUser* to all the clients subscribed to the public chat. The clients will know whether that *IUser* is already in their map of public chat users, if the user already exists in the map that user will be removed, otherwise added.

6.3.2 Business Layer

The business layer is in charge of performing the logic operations on the data.
The *ServerSystem* class keeps track of which users are online in the system and who are subscribed to the public chat. In addition, it is facilitating the connection between the connection layer and the presentation layer. So often times, it will receive a call from the connection layer, perform some logic on it, forward it to the persistence layer that will gain the data, which will go back to the *ServerSystem* to have more logic performed, whereafter the data will be sent back to the connection layer to be returned to the client.

ServerSystem: changeInfo()

The *changeInfo()* method is utilizing the *IManagement* type, which includes an integer that represents an action. If the client has sent a management object with the action 0 or 1, it means that they wish to change their password or email. If that is the case, a method from the persistence layer will verify the information they have sent in the request, comparing e.g. the hashed “old password” with the hashed password in the database. The persistence layer will return a boolean based on success, and if it was correct, the server will return a boolean accordingly. The client will then change the action to 2, which allows it to change the profile information together with either mail or password.
However, if the client wishes to only update the profile information, without a change of password or mail, the *IManagement* action will be set to 2, and the changeInfo will only tell the persistence layer to update profile information.

6.3.3 Persistence Layer

The persistence layer of the server has the purpose of facilitating a communication between the server and database. This is done using queries in the form of prepared statements.
The *DatabaseHandler* holds most of the functionality of the persistence layer. This is the class communicating with the database, together with the responsibility of creating Persistence object labeled “*Per*”-classname.

DatabaseHandler: login()

The login method describes the general usage of the *DatabaseHandler* class. It takes an *ILogin* as parameter. This *ILogin* contains the hashed mail and hashed password. A call to the database will try to fetch the *userID* and hashed password connected to the hashed mail. If the fetch is successful, the *userID* will be saved in a *PerUser* and the hashed password of the *ILogin* will be compared to the hashed password fetched from the database. If they are the same, the attempting login had the correct information, and the rest of the users information can be fetched based on the *userID*. The information will then be bound on the *PerUser*, and the *PerUser* will be put into the *ILogin*, which will be returned to the requesting client.

6.4 Security

SYNchat uses MD5 hashing in order to securely store user information. This is done by running all user passwords and mails through the MD5 hashing method in the business layer as soon as the user registers on the client. This is done for two reasons:
Hashing the data on the client itself complicates “man in the middle” attacks (Symantec Corporation, 2018). Man in the middle attacks will still be a possibility but the only critical data that can be intercepted will be a 128-bit hash value and therefore of no use to the attacker.
Furthermore, by having hashed data going through the server it disables the possibility for anyone with access to the server to pull out user critical data before it can be stored in the database.

Sending messages between clients using Atbash ensures user privacy. It will also complicate “man in the middle” attacks (to some extent).
When the ciphered message is received on the other client it uses the same exact method to decipher the message back to the original text. This means that all transmitted communication between clients through the server, will be ciphered as shown in the diagram below.

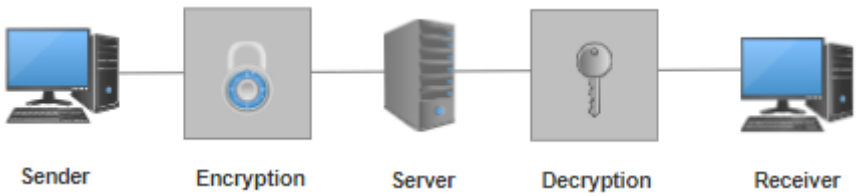


Figure 6.4 1 Overview of message communication within the system. The sender’s message is encrypted, sent through the server to the receiving client and decrypted before being shown to the receiving user

6.5 Distribution

The reason for using the SDU server is that there would be no problems accessing the database, as both the server and the database are now accessible on Eduroam (SDU closed network). Another solution could be to run the server on Azure virtual machines, however this would affect the entire project, including Java code and was not the top priority for the project team. Future releases would be expected to bring changes to this aspect of the system, as it would allow connections from outside of Eduroam without the usage of a VPN service and therefore make the running and restarting of the Docker image easier.

Building a new Docker image requires the process of writing a Dockerfile. It was decided to build the image from the base image of Ubuntu Linux, as it was the most familiar operating system to the project team as a whole.
From there, it was decided that the best way to make the image run the server was to download Java, copy the postgreSQL drivers onto the image and from there ordering the image to run the Starter class from the Java project.

Thereafter, all that was left to do was to get the image onto the server and run it.

This process involved pushing the image to Docker Hub, which is a site containing various Docker images, both official base images and user images.
After the push, the only thing left to do was to access the server via SSH and pulling the image from the online Docker Hub repository and run it.
The image containing the server now runs in a container on the Linux server provided by SDU. In order to contact the server, the call is actually made to the physical machine running the Docker container, and from there, the call is routed through to the container via. port mapping. By exposing port 8080 in the container and mapping port 8080 on the physical machine to 8080 inside the container, the traffic incoming on port 8080 will automatically be routed to the container where the server is running.

7 Testing

This chapter provides an overview of the testing that have been done throughout the production of the SYNchat system. At first, it describes the testing methodology used. Thereafter follows a section containing the unsolved bugs that have been discovered in the system.
Firstly, testing was focused around the server acknowledgement, of clients connecting through the correct IP and port. Later stages of testing consisted of connecting multiple clients (max test of 70 clients) at the same time and making sure these clients could send and receive messages to one another via the NetBeans command prompt.
Secondly, the testing went to focus on stress testing the server itself. This was done by letting one or more connected clients dump massive data onto the server at once thus checking if the server would crash due to the unexpected size of data rushing in from multiple IP's at once. The tests showcased that the server handled the stress test better than expected.

After implementing the GUI to the system, all tests were focused around testing all possible features before a new update was pushed to the master of either the server or the client site system.

Since most of the features are use case driven, we tested the use cases and if the output was as expected after executing the code related to the use case, the code was considered working. However, some features went across use cases and were system wide, which resulted in using the debugger to identify the problem, and under what conditions the problem occured. Automating the tests would have been a nice addition to this, however, it would have been very comprehensive since the code isn't component based, and many features has to be tested at once. E.g. a simple change to some code would cause all automated tests to fail, and would require all tests to be adjusted and rewritten. This would render most of the tests useless the majority of the time, unless part of our team was dedicated to test maintenance.

7.1 Bugs

The current state of SYNchat, contains few but major bugs. First of all, when a user enters the public chat, all other commands other than public chat commands will not work. Currently when entering the public chat, a new thread is started and a while loop listens for objects from the server, the thread then calls methods from the while loop and updates the GUI from here. However when attempting to leave public chat, both the thread and while loop should be interrupted, and SYNchat should be able to receive objects other than public chat compatible objects again. This functionality does not work as intended, since leaving the public chat does not interrupt the thread and leave the loop properly. A possible fix could be invoking a sort of wait command which would pause SYNchat briefly till the thread and loop was completely interrupted before doing any other actions.

Furthermore when logging out of SYNchat and a new login is attempted in the same session, the client will crash and possibly also the server. We believe that when logging out of the system, the client is not properly stopping the previous activities including the loops and threads running on the client. A fix would be to establish a whole new connection when attempting to re-login, and reset all data in the client, so that every login would be a completely fresh login.

8 Cultural analysis

Although technology has brought the world closer together, and given people the power to reach anyone worldwide, there are still a lot of difficulties regarding cross cultural management and communication. It is crucial to take all perspectives into consideration, when there are a variety of cultures involved which may be very different from one another. In this section circumstances regarding cultural management and communication will be examined using the dimensions described in Hofstede and Schein's model of culture. We will be looking at Denmark, USA and Japan in our analysis and have chosen these three because we believe there is a significant change in culture and values from western (US) to eastern (Japan) countries. Denmark is chosen because the developers are all Danish.

8.1 Hofstede's Model of Culture

Professor Geert Hofstede conducted a comprehensive cross-cultural research of how values in workplace are influenced by culture. Hofstede formulated a cultural framework of 6 dimensions, that distinguish countries from each other. Each dimension represents independent preferences for one state of affairs over another. The model consists of the following dimensions (further explanation can be found in appendix B1 Geert Hofstede):

- Power Distance Index (high versus low).
- Individualism Versus Collectivism.
- Masculinity Versus Femininity.
- Uncertainty Avoidance Index (high versus low).
- Long-Term Orientation Versus Short-Term Orientation.
- Indulgence Versus Restraint.

An analysis of the three chosen countries have been made, and the visualisation of this analysis can be found in the appendix B1 Geert Hofstede. Some of these dimensions will be analyzed and discussed according to how social media is used in the chosen countries. (Schneider et al. 2014)

8.1.1 Social media in Japan

The use of social media in Japan is very limited. Media like Facebook, LinkedIn and such has not really been taken in by the Japanese people due to the culture (Japan Association of Marketers, 2017). According to the Article at Medium the Japanese are very anonymous people.

Going through the Hofstede-analysis (Hofstede Insights, 2018) it is evident that Japan has a culture with very centralized ethics. In *masculinity* Japan scores the highest out of all three countries in this project case. Japan has a *masculinity* of a 95. This mean that the Japanese people live to work. They do not have much free time to do what they personally like to do. This can be both positive and negative when looking at releasing SYNchat to the the Japanese market. When the Japanese people work this much it can be hard to let them get used to the features of SYNchat but this might also let them use it in a business context as it may lead to uses like an intercorporate conference chat.

With the *uncertainty avoidance*, Japan scores a 92 which is very high compared to USA and Denmark. This has to do with the natural disasters Japan has endured over the years. This can make it a bit hard to enter the Japanese market as it will be something completely new. But if the Japanese people adopt SYNchat it might be the prefered chat system for many years to come as it will become known and mainstream in the culture as a whole. The high *uncertainty avoidance* has meant that rulebooks has been printed with instructions on how to dress and act at certain events. This has given a low score in *individualism* as everybody is afraid to do anything outside the 'code of conduct'. A low *individualism* will also make it difficult to get a foothold in the market as people are afraid of doing something outside the norms.

8.1.2 Social media in USA

The American culture really shows how proud of a nation they really are. USA has the highest score in *individualism* out of all three countries/cultures with a 91 out of 100. It is very important for Americans to show their worth. Therefore all goals in life has to be set to public display. A trophy from 3rd grade for example, will be displayed on the wall at home along with all other diplomas and such achievements. The American culture is focused on what every person has accomplished throughout their life. This will mean that the Americans will like that they can write a profile text in SYNchat in order to display his/her accomplishments for everyone on SYNchat to see.
The Americans are usually not that risk oriented (*uncertainty avoidance*) but after 9/11 and the Edward Snowden leaks they have been a bit more risk concerned. With the encryption and cipher on SYNchat this can make the Americans feel safe when using the system.
USA is not really long term oriented. This can be seen when looking at company goals. New goals are set to be completed within the quarterly accounting rate. This is also reflected to their private lives as the Americans live to work with their medium high *masculinity*.
Privately the Americans does not tend to have strong friendships. This is especially seen with men. However they are used to do business with people they do not know. This has a high chance of also reflecting to their personal life making them more outgoing and open to get to know new people. This is a good chance for SYNchat to get into the American market as users can chat with friends in private chats as well as random people from around the globe in the public chat.

8.1.3 Social media in Denmark

The Danish culture is very different to the American and japanese. The Danish people has a high *individualism* and a low *masculinity* which means that the culture focuses around freetime. This is seen with the Danish people working less than many other cultures in order to also get time for themselves.
The Danish people tend to plan ahead but do not mind last minutes change of plans. This is the reason for the low score of 23 out of 100 in *uncertainty avoidance*. This also affect SYNchat as the Danish people aren't scared of new technologies or gadgets. This means that they are likely to try out SYNchat in a very early release thus giving it a chance to go

viral quick as long as the Danish people has some friends/contacts internationally. One problem is that the Danish culture is not very focused on *long term orientation*. This could result in a quick spike in Danish users with a just as quick lack of daily Danish online users efter a few weeks or months.

8.2 Schein's Model

In the 80's an American management professor Edgar Schein developed an organizational culture model with the purpose of making culture more visible within an organization. The model is known as the **Onion model** or the **Iceberg model** because of its different layers as seen in the image below. Edgar Schein divided organizational culture into three different levels or layers: Artefacts and symbols, Espoused values, and Basic underlying assumptions. (Muldar, 2017)

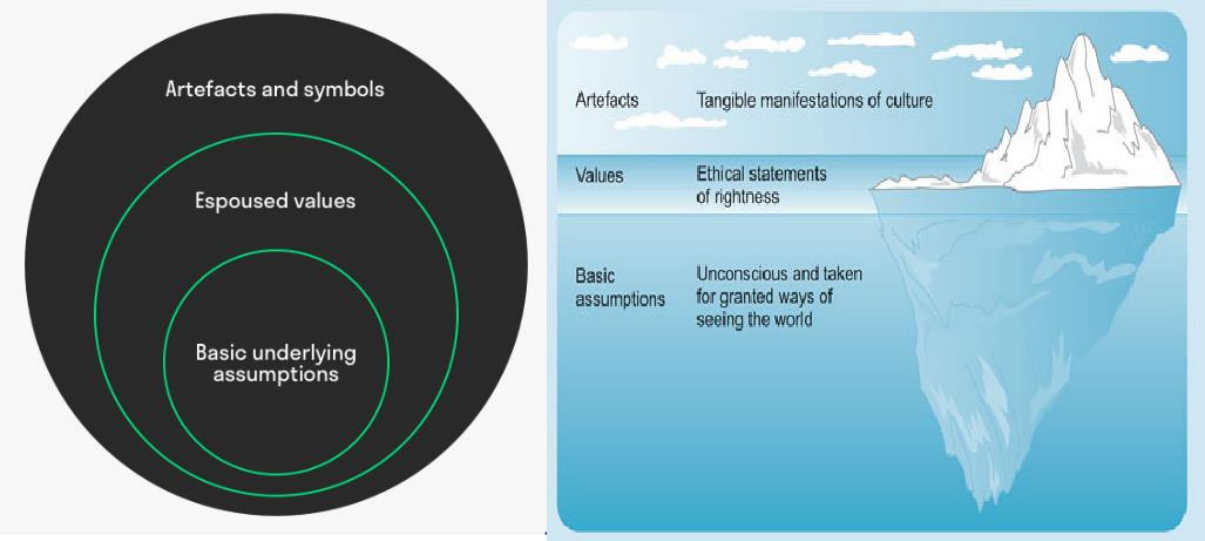


Figure 8.2 1 The image illustrates two views of Edgar Schein's model of culture.

This model can be applied to the project by analyzing and categorizing design choices and elements in SYNchat according to the different levels. The beliefs and values of the developers, which possibly are reflected onto the system, could easily be misinterpreted or misunderstood, but will still somehow have an effect on future users of the system.

8.2.1 Artefacts

The most visible artefact in SYNchat is the logo of a peach. The peach is not only a popular choice of fruit but is symbolic in many cultural traditions such as art, paintings and folk tales. In China, peaches symbolize immortality with the origin from a folktale called Peaches of Immortality, where immortals would consume peaches and obtain longevity. In Korea, the peach is the fruit of happiness, riches, honors and longevity. It is one of the ten immortal plants and animals, and therefore often appear in folk paintings. In Europe, peaches are often to be found in paintings. Artists of Renaissance symbolically used peach to represent a heart, and the leaf attached to the fruit as the tongue, and thereby implying to speak truth from one's heart (Wikipedia, 2018b). Today's pop culture has given the peach a very

different meaning. With the birth of Emojis, the peach is a plump, heart-shaped object with two cheeks, and thereby have similarities to a humans' rear end.

Next noticeable element of SYNchat is the overall theme of the color blue. The color blue is often used by companies that wish to convey reliability, trustworthiness and communication. The color is also associated with the sky or sea and therefore has calming and harmonious qualities. It can also be associated with feelings and emotions to express sadness or depression. (Gross, 2018)

SYNchat also has artefacts to visualize one's nationality either by displaying a flag or by choosing an avatar of pre-defined profile pictures.

8.2.2 Values

SYNchat targets a group of young people around the age of 18 to 34. The developers wanted a communication platform which would make it easier for all people to connect and developed the desired system with that specific mindset. The developers intended to develop a free and easy-to-use platform for people of all cultures. Some of the values that the developers believe in is trustworthiness, reliability, scalability, security, and equality.

It is not written anywhere in SYNchat, but there exists general different norms in every communication platform. There are guidelines whether it is written or not such that content must not be misleading, deceptive, illegal, violent, discriminating, sexual, or offending among many more. It is yet to be implemented but there are options in SYNchat, to handle issues and conflict in violation of rules and such. The options are to either ignore or report users who for example could be discriminating or use hate speech.

8.2.3 Assumptions

The developers of this project are all males which could derive to an assumption that the working environment has been defined by macho humor and a laissez-faire approach to regulation. Another assumption is that it has been a loud environment because males usually have an aggressive approach for example when arguing over which opinion is the most correct.

As the developers have the same nationality and a similar culture, their shared values and beliefs can be visible in the actual system, as they have developed a system purely based on their thoughts and wishes. The final system fulfills all the developers' needs rather than the future users'. However when looking at the actual system, one could assume that it is intended for young people to use, based on the choice of the peach as the logo.

9 Discussion and evaluation

The following chapter contains descriptions and discussions of the possible improvements that could be made to the SYNchat system.
If there was more time to develop the project, the development team would have liked to have finished the private chat feature, that also listed all of the users friends. Also functionality to be able to send other kinds of messages, such as pictures or sound messages would have been added. Another feature would be to completely translate the entire softwares interface when logging in with different nationalities.
The friends list that should have been added, should also be able to display offline users, and the idea was to instead of displaying a green online dot and a grey offline dot as most social media platforms do, SYNchat would instead display the flag for each users nationality in either color or black/white depending on the user were online or not.
Being able to search for users would also be a key feature to add, otherwise it would not be able to add new friends other than those you meet in public chat. Once you have added a new friend, as you already are able to in the system, it should be properly possible to remove them again. In addition users should also be able to both ignore and report other users, so that they will not hear from that person again.
Finally, with the search for user functionality it would be possible to display more information about other users, such as displaying another users friends in form of their names, instead of just ID's as SYNchat currently does.

9.1 Architecture

In the current implementation of the application the principles from layered architecture have been violated on the client due to calls running form the connection layer and up the system when certain objects are received through the object stream. This increases the coupling of the system and removes control from the presentation layer. As a solution to circumvent this issue could be to implement the Observer design pattern (Tutorialspoint, 2018b) which would allow to set up observable objects in the connection layer that can notify possible observers/listeners to any change to the object received from the object stream. This allows listeners in the above layers to remain in control and adhere to conventions of layered architecture. Furthermore it allows for implementation of *ActionEvent*s in the javaFX presentation to interact with updates through action methods which would allow for features such as notifications.

9.2 Client

9.2.1 CSS

If the project were to start over, one of the things that would have been done differently, is instead of writing all the style code in java and *SceneBuilder*, a dedicated *CSS* file would have been used. This would have saved hundreds of lines of code, and would have made both the style and java code more manageable.

9.2.2 High cohesion

Currently a lot of the logic on all client layers happens in the facade classes, which is not completely correct according to the three layered model. Instead, our facade classes should just be communication classes that handles communication in and out of the package, and let other package specific classes handle the logic. This would have ensured higher cohesion since facade classes would only do what they are meant to do, and so would the other classes such as the *ClientSystem* class in the business layer, that currently should have a lot more responsibility, which would finally make the client code more intuitive to understand and maintain.

9.3 Server-client

9.3.1 Remote Procedure Call

A Remote Procedure Call (RPC) (Arpaci-Dusseau, 2015) is when a program causes a procedure to be executed on a different machine. The procedures code will look close to identical if it's a remote procedure call or a local procedure. In Java, the remote procedure calls would be represented as remote method invocation. Specifically for our project, this would have been an advantage to use, to remove the insecurity of "instanceof". In addition, it is very convenient to be able to call procedures directly on the server.

9.3.2 Representational State Transfer

Representational State Transfer (REST) (Booth et al., 2004) is based on web-standards and the HTTP protocol. The REST interactions are stateless, this means that a message in REST does not depend on its conversation. It has the basic functionality of create, retrieve, update and delete. It utilizes the URI to identify resources and apply web protocols.

9.3.3 Server

SYNchat uses a sub-version of RPC, since it is the client who requests information of the server, and only gets the data it asks for. Our implementation does, however, not support direct method calls, but rather the server handles the data it receives in a way that triggers method calls. Had the server been made with RPC, the majority of the ClientHandler's complexity would have been superfluous, and security would have been increased.
If a web interface was desired for SYNchat, a RESTful api would have been a nice addition, with quick setup using the Spring boot and maven/gradle, which java supports. Using this does not mean we would have to redo the entire server side, but rather have a server for desktop applications in addition to a server for web applications. This could cause problems in conjunction with displaying online users and using the public chat cross platform, however this could be settled using a load balancer or middleware with logic attached to keep track of users across servers. Furthermore, if SYNchat would desire to expand to android and iOS, few changes would have to be made to the server-side code, which would include using JSON format instead of Java Objects (which would bring high cohesion with the REST api) or deciding to change to RPC.

9.3.4 Security

Atbash

The version of Atbash in SYNchat does not support the encryption of special characters. This is not hard to implement but it will take some time which was down prioritized as the chat would be tested using english.
The implementation of special characters would consist of updating the existing Enumerator Updating the Enumerator with all characters including special characters will be an easy task which can be seen as a working theory when looking at the mathematical function in section 5.3.2.

MD5

As stated in the design chapter of encryption, MD5 is not a secure hashing algorithm on its own. This was discovered long after the implementation of MD5 in SYNchat, that MD5 had a critical flaw making it possible to get the same hashed value from two different values(Ciampa, 2008, Page 290).
This does not mean that MD5 is useless for the final product at all but if SYNchat should have the security it needs, the method for hashing mails and passwords could use Salt (Defuse Security, 2018).
Salt is a way of adding strings in a password in different places in order to make it more difficult to crack the password used on the accounts. The string can be a random generated string placed after for example every 10th character or a fixed value added to the passwords in different positions. Salt is added to the password before it is hashed.
An example of this would be to append SYNchat to all passwords before hashing them. so a password like “p455w0rd” would be “SYNchatp455w0rd” after it has been salted. From here it will be hashed using the MD5 hashing algorithm implemented in SYNchat.

Argon2

Another possible solution would be to use the Argon2 hashing algorithm instead of MD5. Argon2 comes in various different versions and for this project the best solution would be to use the version called Argon2id which is the best version protecting against external attacks. Unlike the standard MD5 hash function, Argon2 ensures that salting is used as it has many different parameters to tweak in order to make sure that the hashing is done exactly as the developer wishes. Argon2 won first prize in Password Hashing Competition in 2015 and is the most secure hash function the team has been able to find through research.
(Khovratovich et al., 2015), (Khovratovich et al., 2017)

9.4 Scalability

At this moment, the system is limited in a number of ways, when it comes to the number of clients it can handle. This is due to the fact that there is only one instance of the server running on a single machine. Threads and sockets in the system can only handle a finite number of clients, before the load from all incoming and outgoing traffic could slow down the application significantly. This would cause extended loading time and possible application crashes from the user's perspective. Since the server is only run on one machine, this also means that if the machine breaks, connection to the server, and therefore the possibility of chatting, would not be available. In the following paragraphs, a few possible solutions will be described.

One way to solve this problem, is to scale the system as a whole. In this case, the system is mainly the server and therefore a solution would be to run several instances of the server, ensuring high availability even if one or more machines should break down. It is important to make sure that all instances of the server cluster is connected to each other, ensuring the possibility of clients communicating across different instances of the server application. Same principles can be used for the database as long as they share the data between each other.

Docker swarm is a possible solution to this issue. Docker swarm is a built-in mode, native in Docker since version 1.12.0 (Docker Inc, 2018b). As the name suggests, this will effectively work as a bee swarm, with the queen of the hive being the manager and the worker bees will be doing all the work. In swarm mode, containers are run on multiple physical machines and each container can either act as a worker or a manager. The way that tasks are given to the swarm is to give the managers a service to complete, by creating a service inside the manager container. The managers, will also help complete the task, if not put in drain mode, which will make the manager exclusively delegating the tasks, not helping with them.

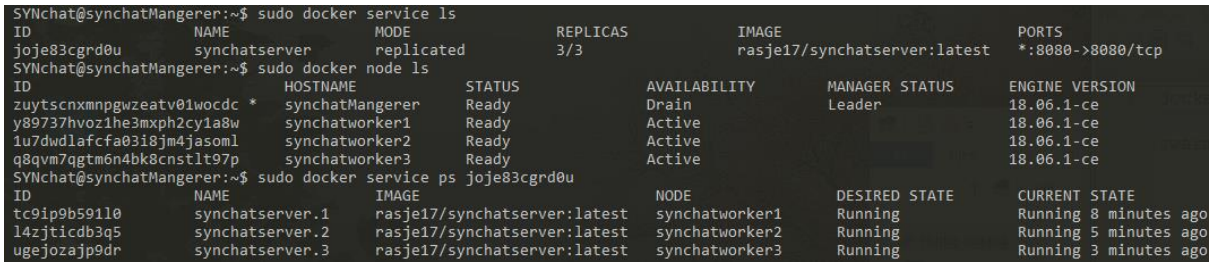


Figure 9.1 1 Screenshot of an overview of a running swarm.

The screenshot above is a visualization of how a swarm cluster could look from the container manager's perspective. There are 3 workers and 1 manager set to drain mode, which means that it only delegates the tasks, it does not perform it itself. When a process overview is requested, it is revealed that there are 3 nodes currently working on tasks, and the service they are fulfilling are the latest image of synchatserver. Ideally, there would be several managers in the swarm, making the system less vulnerable to total breakdown, as there are multiple managers to delegate the tasks to the swarm workers. In appendix B2, there is a table of how many managers can be lost, depending on the number of managers

in a swarm. Depending on the size of the system, the optimal number of managers will be 3-5, and even more when the systems are very large (Neerup et al., 2018b). The system would probably have 3 managers when the final release was made, with the possibility of expanding according to the user base.

When it comes to high availability, another important factor is load balancing. When using Docker swarm, the swarm cluster already has internal load balancing, which means that the manager(s) will distribute the tasks to the nodes that have the least load at the current moment.
However, as managers can also be the ones performing the services, it would make sense to also have an external load balancer before entering the swarm itself. This would mean that not only will the managers distribute the work amongst the workers, but the incoming services would also be distributed amongst the managers, therefore balancing the load between the physical machines even more. An overview of this solution can be seen in appendix B3.

An improvement that could be made to the Docker image of the server, would be to build it from an Alpine Linux base image instead of an Ubuntu Linux base image. The used Ubuntu image has a size of 188mb, whilst the Alpine image has a size of only 5mb. This would mean a much more lightweight image, while still maintaining all of the functionality of the Ubuntu based server image as no functionality unique to the Ubuntu base image is necessary.

One difficulty of transferring the already working solution of the SDU linux server to Azure is that the project team would not only have to create virtual machines, install Docker and run the containers on these machines. It would also require a migration of the database to either an SQL database on Azure or a Galera cluster. This would be the optimal solution but would affect the Java code as well, as Galera uses MariaDB instead of PostgreSQL which is already implemented in SYNchat.
Galera is basically a service, that allows the user to utilize a database cluster to store data. The advantages of using a database cluster are:

- Data redundancy
    - When using a database cluster such as Galera, the data is stored on several nodes instead of just one. This means that a lot would have to go wrong for the data to be lost, compared to when the data is stored only on a single computer. An alternative that would also solve this problem, would be to have a cloud-based database, which would also be online 24/7 and have regular data backups.
- Scalability
    - Database clusters like Galera use load balancing to distribute the incoming traffic evenly between the active nodes in the cluster. This means that no node will receive too many requests and fail because of that.
      In a MySQL cluster, which is what Galera is, the maximum number of nodes in a cluster is 255 (FromDual, 2018). This means that when using Galera, the

user has a lot of storage space and a lot of backups running simultaneously, which ensures that there is a minuscule chance of losing any data.

- Availability
    - When using several nodes to run the database, it improves the availability of the database quite a lot. Since the same data is stored on all nodes in the cluster (Multi Master), it is easy to keep the data available, even when one or a few nodes are undergoing maintenance or have failed. Although Galera will not put a number on their availability(five nines etc.), the only occurrence that would cause downtime, would be if all nodes in a single cluster would break at the same time, which is highly unlikely.

10 Conclusion

For this semester project about “Design of software systems in a global context”, we intended to create a communication platform that differs from other well-known social media platforms, and especially their take on privacy policies and commercial marketing.
Our solution for a chat application, SYNchat, aims to be simple and transparent to the users’ perspective without reducing efficiency.

We achieved this by focussing on user experience and what we thought would be intuitive. This resulted in a simple design that kept to a color scheme, in addition to not flooding the screen with unnecessary elements. Furthermore, we reused a lot of the familiar interfaces from other well-known social media platforms.

We also implemented features to reflect the users’ culture, for instance by being able to choose a nationality and personalize their profile with an avatar. When entering the application, the welcome messages will differ depending on nationality. The online users’ nationality will be displayed in chat rooms by their country’s flag besides their name and avatar as well, thus ensuring cultural diversity.

To achieve a strong foundation, we chose to implement three layered architecture in both client and server applications, among distributing the system in a 3-tier framework. The distribution of the server is running through Docker, which ensures a light-weight and scalable implementation.

Although the system is kept simple, security has been kept a high priority in SYNchat throughout the development. The users’ information is kept safe at all times through encryption and hashing. Concerning the users login information, both the mail and password is hashed with MD5 as soon as the user registers a new account, without the unhashed information ever being stored. This ensures that SYNchat will not be able to sell any personal data as this is not available to the developers, nor is it possible to access unhashed information in the database. To ensure user privacy all messages are also encrypted thus making them unreadable by malicious users.

Finally we can conclude that the project is satisfiedly completed as a functioning and distributed chat application prototype, that fulfills the requirements and our expectations.

11 References

Agile Alliance. (2018). What is Agile Software Development? Retrieved from https://www.agilealliance.org/agile101/ - Visited 18.12.2018

Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2015). Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books LLC.

Booth et al. (2004 February 11) 3.1.3 Relationship to the World Wide Web and REST Architectures. Retrieved from https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest - Visited 18.12.2018

Charles David, (2011). "A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project" Graduate Theses and Dissertations. 12215. Retrieved from https://lib.dr.iastate.edu/etd/12215 - Visited 5.12.2018 (Internet)

Ciampa, Mark, (2008). “CompTIA Security+ 2008 in depth”, First edition, Course Technology PTR, ISBN: 978-1598638134 Page 290

Cohn, M. (n.d.). (2018). Scrum Methodology and Project Management. Retrieved from http://www.mountaingoatsoftware.com/agile/scrum - Visited 18.12.2018

Defuse Security (2018 July 30). Salted Password Hashing - Doing it Right. Retrieved from https://crackstation.net/hashing-security.htm - Visited 12.12.2018

Dhami, M. (2014 August 5). What is FURPS+? Retrieved from https://businessanalysttraininghyderabad.wordpress.com/2014/08/05/what-is-furps/ - Visited 18.12.2018

Docker Swarm overview. (2018b). Retrieved from https://docs.docker.com/swarm/overview/ - Visited 18.12.2018

Docker, Inc. (2018a) What is a Container. Retrieved from https://www.docker.com/resources/what-container - Visited 18.12.2018

ExpressVPN (2018). VPN for Dummies: A Guide for Beginners. Retrieved from https://www.expressvpn.com/dk/how-to-use-vpn/vpn-for-dummies - Visited 12.12.2018

FromDual. (2014). Limitations of MySQL. Retrieved from https://www.fromdual.com/mysql-limitations#limits_galera - Visited 18.12.2018

Gross, Rebecca. (2018,December 04). Color meaning and symbolism: How to use the power of color in your branding – Learn. Retrieved from https://www.canva.com/learn/color-meanings-symbolism/ - Visited 18.12.2018

Hanik, F. (2018) KISS. Retrieved from http://people.apache.org/~fhanik/kiss.html - Visited 18.12.2018

Hofstede Insights. (2018). Country Comparison. Retrieved from https://www.hofstede-insights.com/country-comparison/denmark,japan,the-usa/ - Visited 18.12.2018

Informit. (2018). Retrieved from http://www.informit.com/articles/article.aspx?p=24671 - Visited 18.12.2018

Japan Association of Marketers. (2017, September 11). Why Anonymity is so Important for Japanese Social Media. Retrieved from https://medium.com/@DefinitionMedia/why-anonymity-is-so-important-for-japanese-social-media-f20166741625 - Visited 18.12.2018

Khovratovich et al. (2017 March 24). Argon2: the memory-hard function for password hashing and other applications. Retrieved from https://www.cryptolux.org/images/0/0d/Argon2.pdf - Visited 12.12.2018

Khovratovich, D & Aumasson, JP (2015 December 6). Password Hashing Competitionand our recommendation for hashing passwords: Argon2 Retrieved from https://password-hashing.net/ - Visited 12.12.2018

Muldar, Patty. (2017). Organizational Culture Model by Edgar Schein. Retrieved from https://www.toolshero.com/leadership/organizational-culture-model-schein/ - Visited 18.12.2018

Neerup, Mathias et al. (2018a). Operating systems and network: Lecture 1 - Introduction to the couse tools [pdf]. Retrieved from BlackBloard https://e-learn.sdu.dk/

Neerup, Mathias et al. (2018b). Operating systems and network: Lecture 12 - Distributing Systems [pdf]. Retrieved from BlackBloard https://e-learn.sdu.dk/

Opensource.com. 2018. What is Docker? Retrieved from https://opensource.com/resources/what-docker - Visited 18.12.2018

Oracle. (1996). Client/Server Architecture. Retrieved from https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch20.htm - Visited 18.12.2018

Oracle. (2017). What Is a Socket? Retrieved from https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html - Visited 18.12.2018

Richards, M. (2015). Software Architecture Patterns. Retrieved from https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html - Visited 18.12.2018

Schneider, S. C., Barsoux, J., & Stahl, G. K. (2014). Managing across cultures(3rd ed.). ISBN 978-0-273-74632-4 Harlow: Pearson.

Scrum.org. (2018a). What is Scrum? (n.d.). Retrieved from https://www.scrum.org/resources/what-is-scrum - Visited 18.12.2018

Scrum.org. (2018b). What is ScrumBut? (n.d.). Retrieved from https://www.scrum.org/resources/what-scrumbut - Visited 18.12.2018

Si Alhir, S (2018). Understanding the Unified Process (UP). (n.d.). Retrieved from http://www.methodsandtools.com/archive/archive.php?id=32 - Visited 18.12.2018

Staerdahl, Espersen, Pham, Fisker, Rol, Jensen, & Christoffersen. (2018a, September 18). Group 9s Scrum Board using Google Sheets on Google Drive. Retrieved from https://docs.google.com/spreadsheets/d/1inSsFh3Hbunjp5-m5Kp1TbdsKDnfZFSd3mgJfznMfDI/edit?usp=sharing

Statista. (2018a). Number of social media users worldwide 2010-2021. Retrieved from https://www.statista.com/statistics/278414/number-of-worldwide-social-network-users/ - Visited 18.12.2018

Statista. (2018b). Global social media ranking 2018 | Statistic. Retrieved from https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/ - Visited 18.12.2018

Statista. (2018c). Average age of social media users. Retrieved from https://www.statista.com/statistics/274829/age-distribution-of-active-social-media-users-worldwide-by-platform/ - Visited 18.12.2018

Symantec Corporation. (2018). What is a man-in-the-middle attack? Retrieved from https://us.norton.com/internetsecurity-wifi-what-is-a-man-in-the-middle-attack.html - Visited 18.12.2018

Techopedia Inc.. (2018). Hashing. Retrieved from https://www.techopedia.com/definition/14316/hashing - Visited 12.12.2018

The PostgreSQL Global Development Group. (2018). What is PostgreSQL? Retrieved from https://www.postgresql.org/about/ - Visited 18.12.2018

Tutorialspoint.com. (2018a). Software Design Strategies. Retrieved from https://www.tutorialspoint.com/software_engineering/software_design_strategies.htm - Visited 18.12.2018

Tutorialspoint.com. (2018b). Design Patterns Observer Pattern. Retrieved from https://www.tutorialspoint.com/design_pattern/observer_pattern.htm - Visited 18.12.2018

VPN access. Retrieved from https://www.sdu.dk/en/om_sdu/faellesomraadet/it-service/services/netvaerksadgang/vpn - Visited 18.12.2018

Wikipedia (2018b, December 07). Peach. Retrieved from https://en.wikipedia.org/wiki/Peach - Visited 18.12.2018

Wikipedia. (2018a December 8) Uniform Resource Identifier. Retrieved from https://en.wikipedia.org/wiki/Uniform_Resource_Identifier - Visited 12.12.2018

A Internal Appendices

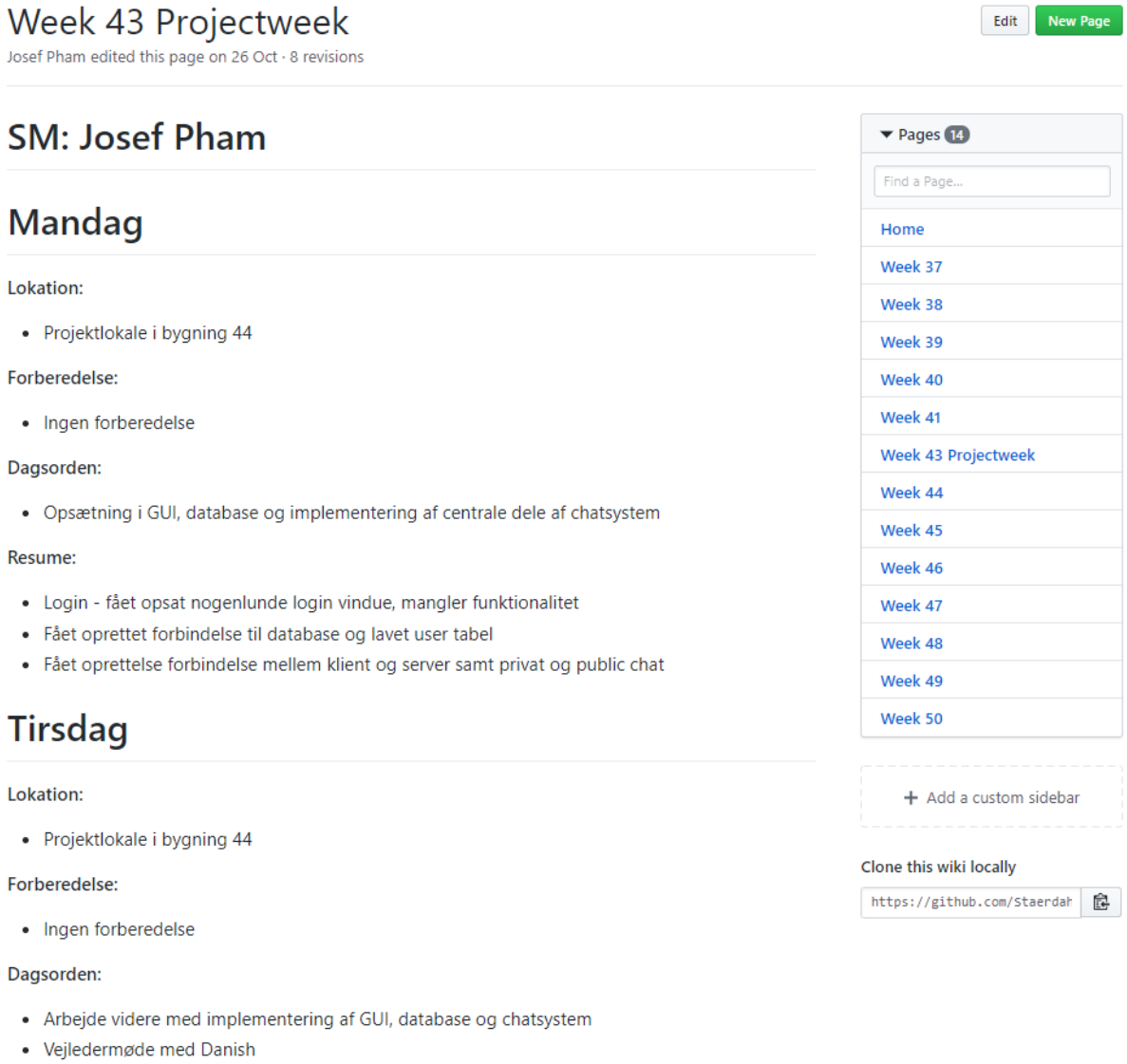
A1 Source code

Project source code can be found in the groups' GitHub. Source code is split into a client and a server part. GitHub: https://github.com/Staerdahl/SYNchatClient & https://github.com/Staerdahl/SYNchatServer

DockerHub: https://hub.docker.com/r/rasje17/synchatserver/

A2 Logs

Throughout the project the members have conducted a digital log on a weekly basis. An example can be seen in the image below. The log is written in the members' GitHub. It is written in danish, as it is a tool only for the group members. Each week a new member takes on the role as scrum master who is also responsible for writing the log, holding meetings etc. The log presents following elements: scrum master, week number, location, preparation, agenda and summary. Group 9s GitHub can be found on following link: https://github.com/Staerdahl/Semester-project-3



Week 43 Projectweek

Josef Pham edited this page on 26 Oct · 8 revisions

Edit | New Page

SM: Josef Pham

Mandag

Lokation:

- Projektlokale i bygning 44

Forberedelse:

- Ingen forberedelse

Dagsorden:

- Opsætning i GUI, database og implementering af centrale dele af chatsystem

Resume:

- Login - fået opsat nogenlunde login vindue, mangler funktionalitet
- Fået oprettet forbindelse til database og lavet user tabel
- Fået oprettelse forbindelse mellem klient og server samt privat og public chat

Tirsdag

Lokation:

- Projektlokale i bygning 44

Forberedelse:

- Ingen forberedelse

Dagsorden:

- Arbejde videre med implementering af GUI, database og chatsystem
- Vejledermøde med Danish

Pages 14

Find a Page...

Home
Week 37
Week 38
Week 39
Week 40
Week 41
Week 43 Projectweek
Week 44
Week 45
Week 46
Week 47
Week 48
Week 49
Week 50

+ Add a custom sidebar

Clone this wiki locally

https://github.com/Staerdah

A3 Scrum Board

The Scrum Board is made using Sheets on Google Drive. Images of each table can be seen in the next appendices, but the entire Scrum board can be found in following link: https://docs.google.com/spreadsheets/d/1inSsFh3Hbunjp5-m5Kp1TbdsKDnfZFSd3mgJfznMfDI/edit?usp=sharing

A4 Product backlog

Product Backlog			
6/9 - 18/12			
Requirements	**Categories**	**Status**	**Prioritized**
Project setup (git, client, server)	Implementation	Completed	Must have
Report	Documentation	Completed	Must have
Use cases	Documentation	Completed	Must have
Class diagram	Documentation	Completed	Must have
Sequence diagram	Documentation	Abandoned	Should have
GUI	Design	Completed	Must have
Database setup	Design	Completed	Must have
Database implementation	Implementation	Completed	Must have
Changing UI (nationality)	Design	Completed	Must have
Create user	Use case	Completed	Must have
Log in	Use case	Completed	Must have
Search user	Use case	Abandoned	Must have
Communication with users (send messages)	Use case	Completed	Must have
Password protected profiles	Use case	Completed	Must have
Encryption	Security	Completed	Must have
Multiple user communication	Security	Completed	Should have
Change info	Use case	Completed	Should have
Add/remove friends	Use case	Completed	Should have
Recieve messages within short timeframe	Design	Completed	Should have
View user profile / avatar	Use case	Completed	Should have
Send images	Use case	Abandoned	Should have
Emojis	Use case	Abandoned	Should have
User friendly interface	Design	Completed	Should have
Delete user	Use case	Abandoned	Could have
Contact users (as admin)	Use case	Abandoned	Could have
Manage chat (as admin)	Use case	Abandoned	Could have
Gifs	Use case	Abandoned	Could have
Report user	Use case	Abandoned	Could have
Chat commands (mute etc)	Use case	Abandoned	Could have
Full HD pictures	Design	Abandoned	Could have
Minimal data loss	Design	Abandoned	Could have
Lightweight, possible phone-port	Design	Abandoned	Could have
Block user	Use case	Abandoned	Won't have
Process reports	Usecase	Abandoned	Won't have
AdminChat	Security	Abandoned	Won't have
Minigames	Use case	Abandoned	Won't have
Sound clips	Use case	Abandoned	Won't have
Cross-platform	Design	Abandoned	Won't have

A5 Release backlog

Release Backlog 1

25/9 - 9/10

Requirements	Status	Prioritet	Est time	
Analysis class diagram	Completed	Must have	56	
Database setup	Out of time	Should have	7	
Report work	Out of time	Must have	70	
Use cases	Completed	Must have	21	
Log in	Out of time	Should have	70	
Create user	Out of time	Should have	70	
Chat between two users	Out of time	Should have	200	
Project setup (git, client, server)	Completed	Must have	14	
Functional requirements	Completed	Must have	7	
Non-functional requirements	Completed	Must have	7	
…				
Time in total			522	hours

Release Backlog 2

11/10 - 29/10

Requirements	Status	Prioritet	Est time	
Sprint planning	Completed	Must have	14	
Project setup(git, client, server)	Completed	Must have	6	
Sprint wrap-up	Completed	Must have	7	
Report work (1st edition)	Completed	Must have	56	
Scrum meetings	Completed	Must have	12	
Sequence diagrams for login	Completed	Should have	14	
Database setup	Completed	Should have	12	
Design class diagram (updated)	Out of time	Should have	35	
GUI setup	Completed	Should have	15	
Log in	Completed	Should have	70	
Create user	Completed	Should have	70	
Server-Client Connection	Completed	Should have	100	
Chat between users	Completed	Should have	100	
Time in total			511	hours

Release Backlog 3

30/10 - 14/11

Requirements	Status	Prioritet	Est time	

Sprint planning	Completed	Must have	14	
Sprint wrap-up	Completed	Must have	7	
Report work (1st edition)	Out of time	Must have	20	
Scrum meetings	Completed	Must have	12	
Video submission	Completed	Must have	7	
Video review	Out of time	Must have	4	
Design class diagram (updated)	Completed	Should have	50	
Friendslist	Out of time	Should have	50	
Profile	Completed	Should have	40	
Add friend	Out of time	Should have	20	
Search for user	Out of time	Should have	70	
View profiles	Out of time	Should have	70	
Send messages as objects	Completed	Must have	35	
Change welcome screen according to country	Completed	Must have	5	
Opening different chats	Out of time	Should have	100	
Change info	Out of time	Should have	40	
PrivateChat	Out of time	Should have	16	
Time in total			560	hours

Release Backlog 4

15/11 - 29/11

Requirements	**Status**	**Prioritet**	**Est time**	
Sprint planning	Completed	Must have	14	
Sprint wrap-up	Completed	Must have	7	
Report work (analysis, design, implementation)	Out of time	Must have	40	
Scrum meetings	Completed	Must have	12	
Scalability	Completed	Must have	100	
Display online	Completed	Must have	30	
Public chat	Completed	Must have	5	
Change info	Completed	Should have	4	
Search for user	Out of time	Should have	20	
Friendlist	Completed	Should have	15	
Add friend	Completed	Should have	10	
Private Chat	Out of time	Could have	30	
View profile	Completed	Could have	25	
Sequence diagrams	Out of time	Should have	20	
Design class diagram	Out of time	Should have	30	
Video review	Completed	Must have	2	

Time in total			364	hours

Release Backlog 4				
4/12 - 14/12				
Requirements	**Status**	**Prioritet**	**Est time**	
Sprint planning	Completed	Must have	14	
Sprint wrap-up	Completed	Must have	7	
Scrum meetings	Completed	Must have	12	
Report: Abstract, preface	Completed	Must have	6	
Report: Introduction	Completed	Must have	6	
Report: Methods	Completed	Must have	6	
Report: Analysis	Completed	Must have	6	
Report: Design	Completed	Must have	12	
Report: Implementation	Completed	Must have	80	
Report: Testing	Completed	Must have	36	
Report: CCM Analysis	Completed	Must have	60	
Report: Discussion	Completed	Must have	70	
Report: Conclusion	Completed	Must have	80	
Report: Appendices	Completed	Must have	6	
Design class diagram (updated)	Completed	Must have	30	
Sequence diagrams	Abandoned	Should have	20	
Time in total			451	hours

A6 Sprint backlog

Sprint Backlog 1 - Goal: Startup, setup, planning

25/9 - 9/10										
First sprint						Start				
					Date:	25/9 -18	27/9 -18	2/10-18	4/10 -18	9/10 -18
Requirement	**Category**	**Status**	**PIC**	**MoSCoW**	Time left:	**593**	**179**	**85**	**75**	**0**
Supporting workflows					Speed:	**75**				
Sprint planning	*Planning*	Completed	All	Must have		28	0	0	0	0
Project setup(git, client, server)	*Environment*	Completed	All	Must have		14	14	10	6	0
Sprint wrap-up	*Planning*	Completed	All	Must have		7	7	7	7	0
Report work (1st edition)	*Documentation*	Out of time	All	Must have		70	56	56	56	0
Scrum meetings	*Planning*	Completed	All	Must have		8	6	4	2	0
Core workflows										
Functional requirements		Completed	All	Must have		7	0	0	0	0
Non-functional requirements		Completed	All	Must have		7	0	0	0	0
Use case diagrams		Completed	All	Should have		21	5	0	0	0
Analysis										
Analysis class diagram		Completed	All	Must have		56	56	0	0	0
Design										
Database setup		Out of time		Should have		7	7	4	4	0
Design class diagram (foundation)		Completed	All	Should have		28	28	4	0	0
Implementation										
Log in		Out of time		Should have		70	0	0	0	0
Create user		Out of time		Should have		70	0	0	0	0
Chat between two users		Out of time		Should have		200	0	0	0	0
Test										

Sprint Backlog 2 - Goal: Being able to chat between two or more users

11/10 - 30/10												
Second sprint						**Start**						
					Date:	**11/10 -18**	**22/10 -18**	**23/10 -18**	**24/10 -18**	**25/10 -18**	**26/10 -18**	**29/10 -18**
Requirement	**Category**	**Status**	**PIC**	**MoS CoW**	Time left:	**511**	**518**	**400**	**296**	**192**	**97**	**48**
Supporting workflows					Speed:	**49**						
Sprint planning	*Planning*	Completed	All	Must have		14	0	0	0	0	0	0
Project setup(git, client, server)	*Environment*	Completed	All	Must have		6	0	0	0	0	0	0
Sprint wrap-up	*Planning*	Completed	All	Must have		7	7	7	7	7	7	0
Report work (intro, analysis)	*Documentation*	Out of time	All	Must have		56	56	54	52	50	48	0
Scrum meetings	*Planning*	Completed	All	Must have		12	10	8	6	4	2	0
Core workflows												
Analysis												
Sequence diagrams for login		Completed	All			14	0	0	0	0	0	0
Design												
Database setup		Completed	L + R	Should have		12	0	0	0	0	*0*	*0*
Design class diagram (updated)		Out of time	J	Should have		35	35	35	35	35	35	*0*
GUI setup		Completed	S	Should have		15	25	16	16	6	5	0
Implementation												
Log in		Completed	S + P	Should have		70	80	60	40	20	0	0
Create user		Completed	S + P	Should have		70	80	60	40	20	0	0
Server-Client Connection		Completed	S + P	Should have		100	25	0	0	0	0	0
Chat between users		Completed	S + P	Should have		100	200	160	100	50	0	0
Test												

Sprint Backlog 3 - Goal: Implement public chat with message objects

30/10 - 14/11														
Third sprint						**Start**								
					Date:	**30/10 -18**	**1/11 -18**	**2/11 -18**	**6/11 -18**	**7/11-18**	**8/11 -18**	**12/11 -18**	**13/11 -18**	**14/11 -18**
Requirement	**Category**	**Status**	**PIC**	**MoS CoW**	Time left:	**537**	**505**	**436**	**385**	**348**	**306**	**299**	**284**	**0**
Supporting workflows					Speed:	**49**								
Sprint planning	*Planning*	Completed	All	Must have		14	0	0	0	0	0	0	0	0
Sprint wrap-up	*Planning*	Completed	All	Must have		7	7	7	7	7	7	7	7	0
Report work (methods, design)	*Documentation*	Completed	All	Must have		20	17	15	12	15	12	7	4	0
Scrum meetings	*Planning*	Completed	All	Must have		16	14	12	10	8	6	4	2	0
Core workflows														
Video submission		Completed	All	Must have		0	0	7	7	7	0	0	0	0
Analysis														
Design														
Design class diagram (updated)		Completed	All	Should have		50	50	40	40	40	20	20	10	0
Implementation														
Friendslist		Out of time	R+J	Should have		50	50	35	10	10	10	10	10	0
Profile		Completed	S + P	Should have		40	40	10	8	5	5	5	5	0
Add friend		Out of time	S + P	Should have		20	20	20	20	15	15	15	15	0
Search for user		Abandoned		Should have		70	70	70	70	70	70	70	70	0
View profiles		Out of time	S + P	Should have		70	70	70	50	25	25	25	25	0
Send messages as objects		Completed	S + P	Must have		35	27	20	10	0	0	0	0	0
Change welcome screen		Completed	S + P	Must have		5	5	5	5	1	1	1	1	0
Opening different chats		Abandoned		Should have		100	95	95	100	100	100	100	100	0
Change info		Out of time	S + P	Should have		40	40	30	20	15	15	15	15	0
PrivateChat		Out of time	R+Pa	Should have					16	30	20	20	20	0
Test														

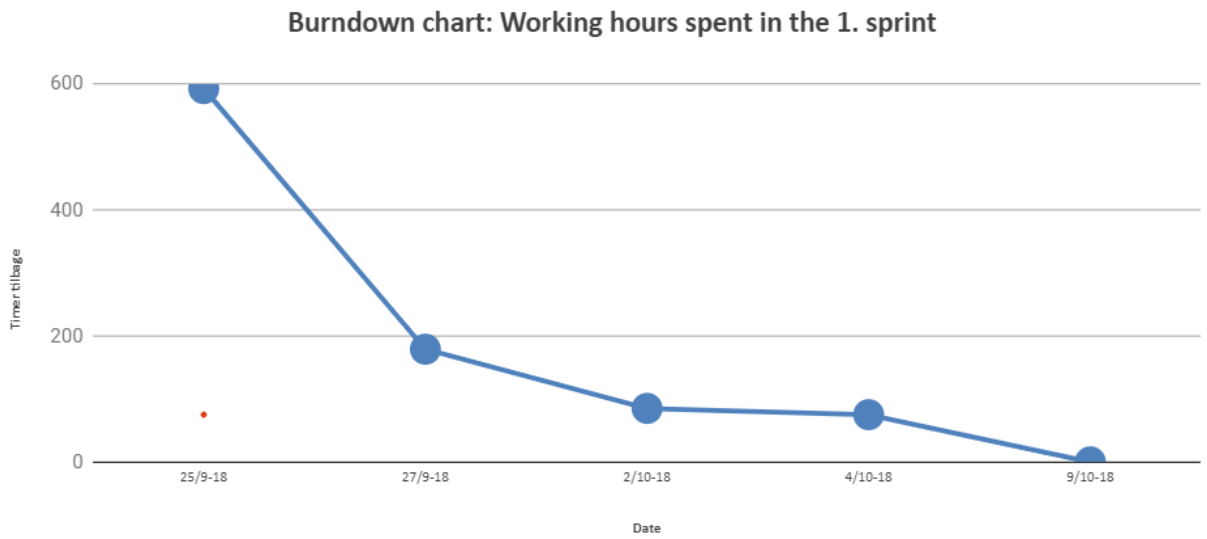
Sprint Backlog 4 - Goal: Run programme with public chat and user info scaled

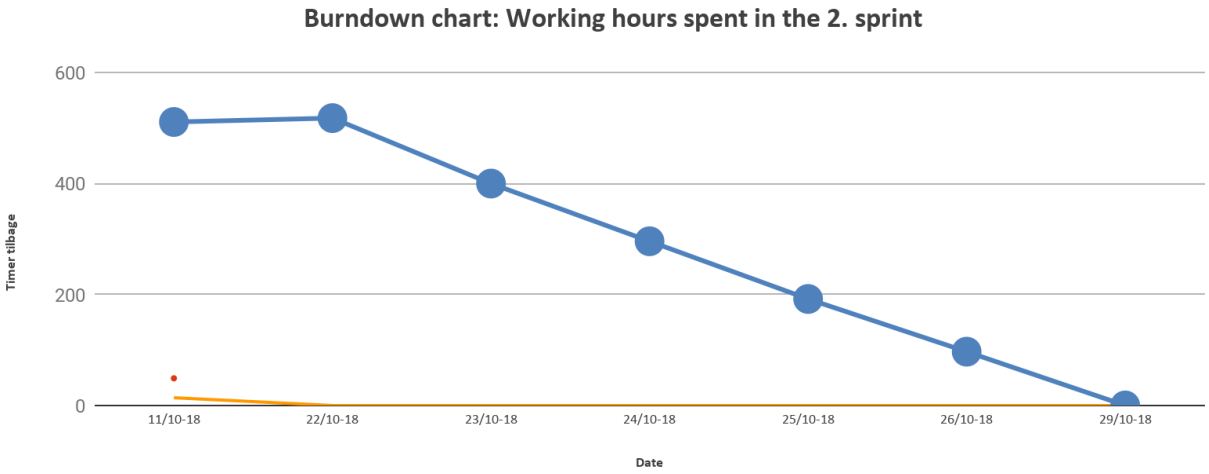
15/11 - 29/11												
Fourth sprint						**Start**						
					Date:	**15/11 -18**	**20/11 -18**	**21/11 -18**	**22/11 -18**	**27/11 -18**	**28/11 -18**	**29/11 -18**
Requirement	**Category**	**Status**	**PIC**	**MoS CoW**	Time left:	**364**	**332**	**327**	**307**	**186**	**178**	**0**
Supporting workflows					Speed:	**0**						
Sprint planning	*Planning*	Completed	All	Must have		14	0	0	0	0	0	0
Sprint wrap-up	*Planning*	Completed	All	Must have		7	7	7	7	7	7	0
Report work (design, implementation)	*Documentation*	Out of time	All	Must have		40	30	27	24	21	15	0
Scrum meetings	*Planning*	Completed	All	Must have		12	10	8	6	4	2	0
Core workflows												
Video review		Completed	All	Must have		2	0	0	0	0	0	0
Analysis												
Design												
Design class diagram (updated)		Completed	All	Should have		30	30	30	20	10	10	0
Sequence diagrams		Out of time		Should have		20	20	20	20	20	20	*0*
Implementation												
Scalability		Completed	R+L+A	Must have		100	100	100	100	40	40	0
Display online		Completed	S + P	Must have		30	30	30	30	2	2	0
Public chat		Completed	S + P	Must have		5	5	5	0	0	0	0
Change info		Completed	S + P	Should have		4	0	0	0	0	0	0
Search for user		Abandoned		Should have		20	20	20	20	20	20	0
Friendlist		Completed	R+J+S +P	Should have		15	15	15	15	15	15	0
Add friend		Completed	S + P	Should have		10	10	10	10	10	10	0
Private Chat		Abandoned		Could have		30	30	30	30	30	30	0
View profile		Completed	S + P	Could have		25	25	25	25	0	0	0
Test												
Public chat		Completed	S + P							5	5	0
Change info		Completed	S + P							2	2	0

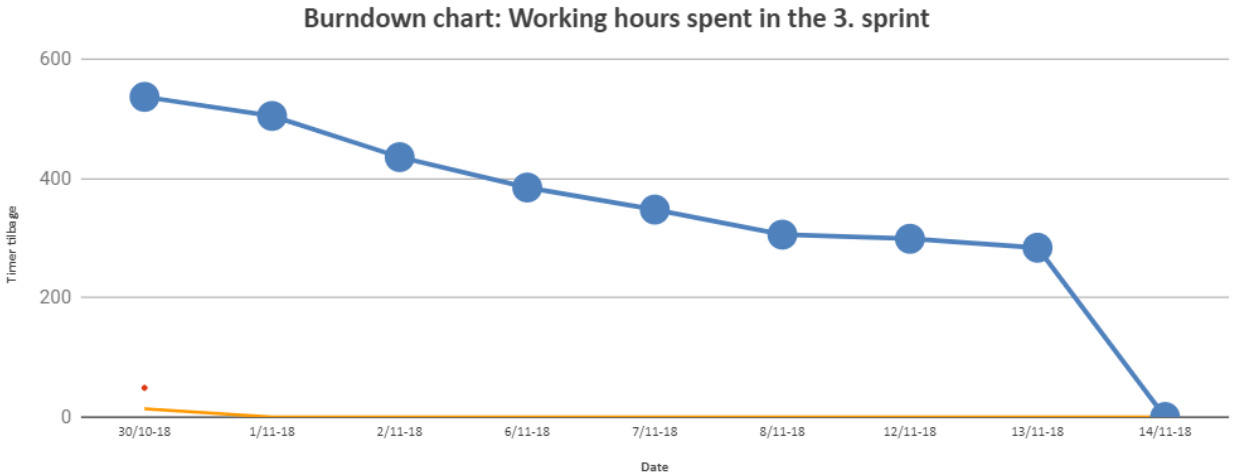
Sprint Backlog 5 - Goal: Finish documentation for deliverable

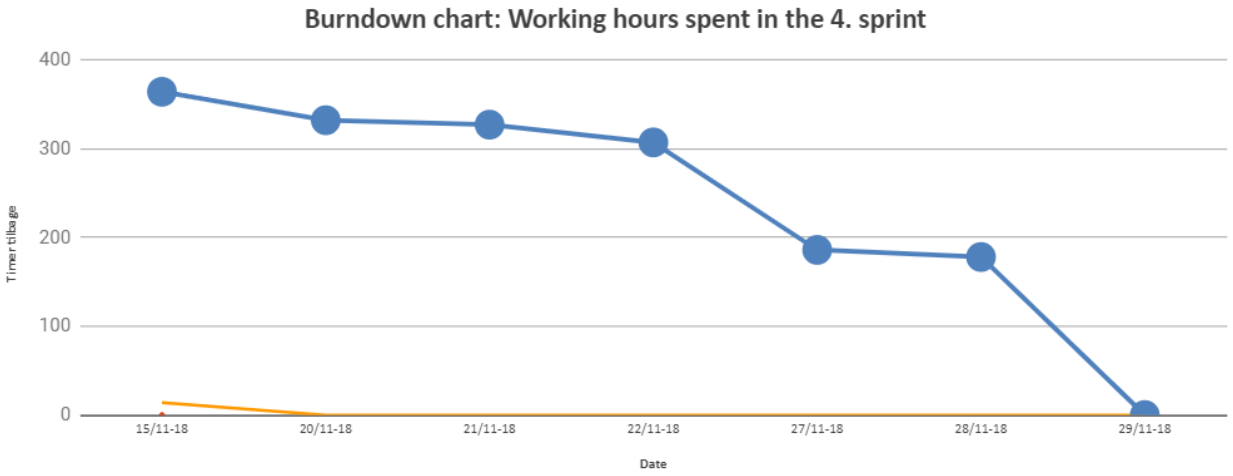
04/11 - 14/11										
Fifth sprint						Start				
					Date:	4/12-18	5/12-18	6/12-18	7/12-18	11/12-18
Requirement	**Category**	**Status**	**PIC**	**MoSCoW**	Time left:	**447**	**321**	**145**	**41**	**0**
Supporting workflows					Speed:	**30**				
Sprint planning	*Planning*	Completed	All	Must have		14	0	0	0	0
Sprint wrap-up	*Planning*	Completed	All	Must have		7	7	7	7	0
Scrum meetings	*Planning*	Completed	All	Must have		8	6	4	2	0
Core workflows										
Report: Abstract, preface	*Documentation*	Completed	All	Must have		6	6	6	1	0
Report: Introduction	*Documentation*	Completed	All	Must have		6	3	1	0	0
Report: Methods	*Documentation*	Completed	All	Must have		6	3	1	0	0
Report: Analysis	*Documentation*	Completed	All	Must have		6	3	3	0	0
Report: Design	*Documentation*	Completed	J+S+PA	Must have		12	2	0	0	0
Report: Implementation	*Documentation*	Completed	R+L+S+PA	Must have		80	44	10	2	0
Report: Testing	*Documentation*	Completed	S+PA	Must have		36	15	5	2	0
Report: CCM Analysis	*Documentation*	Completed	A+J	Must have		60	45	10	2	0
Report: Discussion	*Documentation*	Completed	R+L+S+PA	Must have		70	55	20	2	0
Report: Conclusion	*Documentation*	Completed	All	Must have		80	80	30	2	0
Report: Appendices	*Documentation*	Completed	J	Must have		6	4	2	1	0
Analysis										
Design										
Design class diagram (updated)	*Documentation*	Completed	All	Must have		30	28	26	20	0
Sequence diagrams	*Documentation*	Abandoned		Should have		20	20	20	0	0
Implementation										
Test										

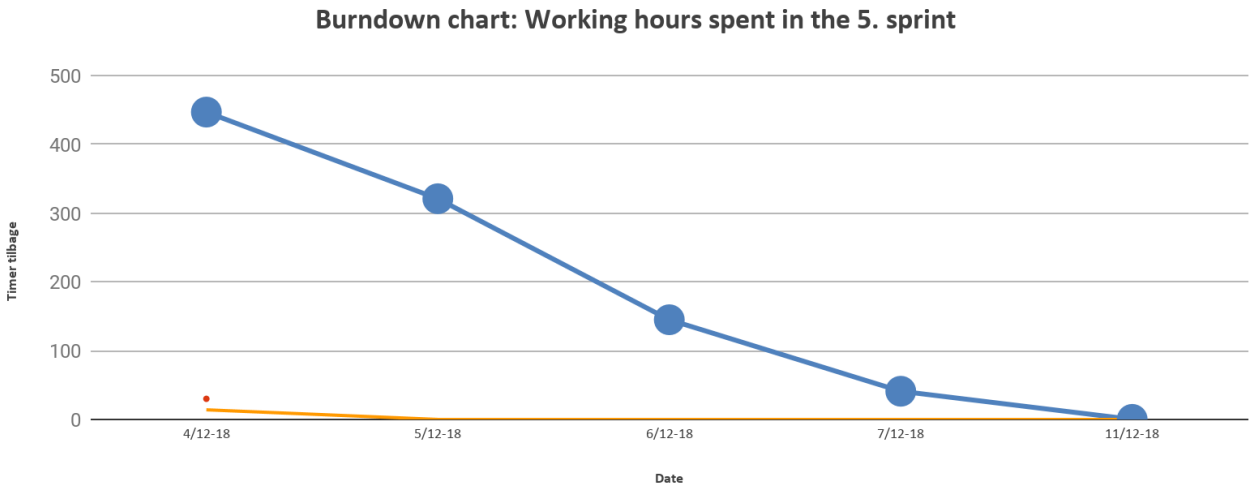
A7 Burn down chart



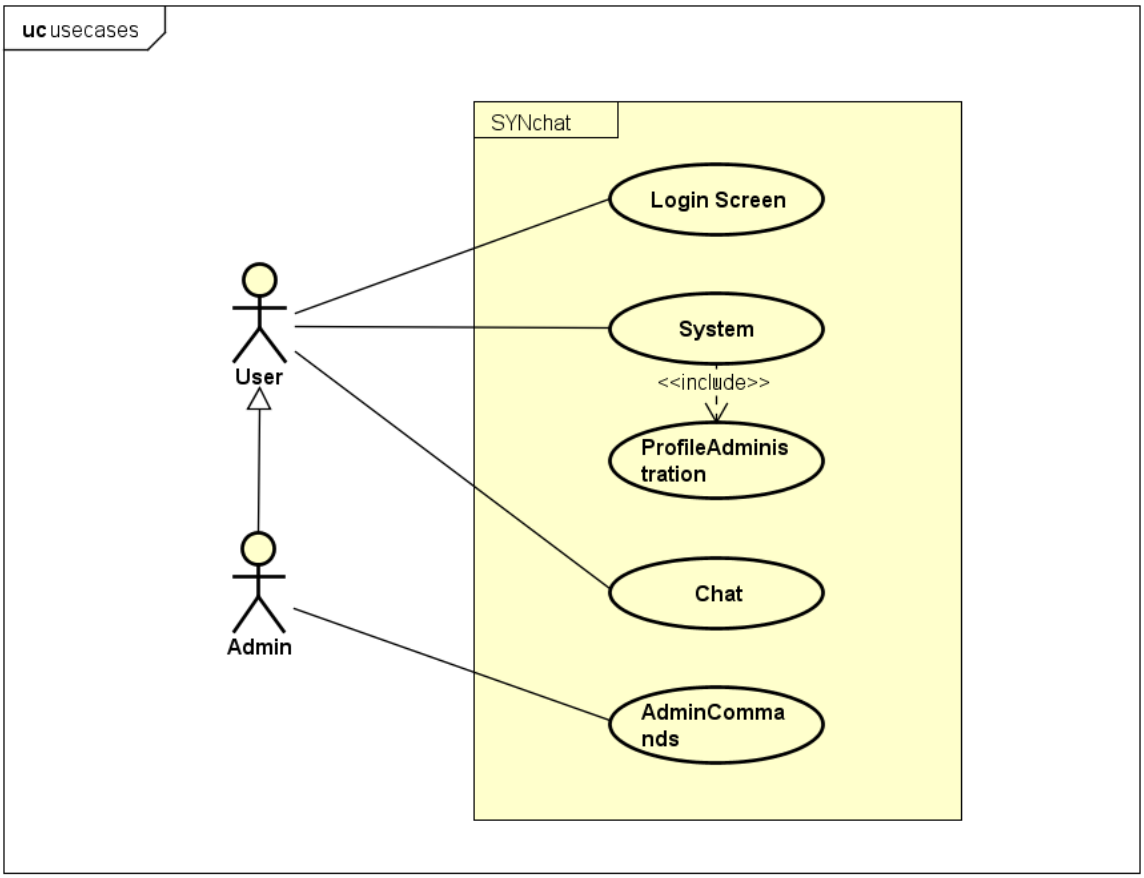


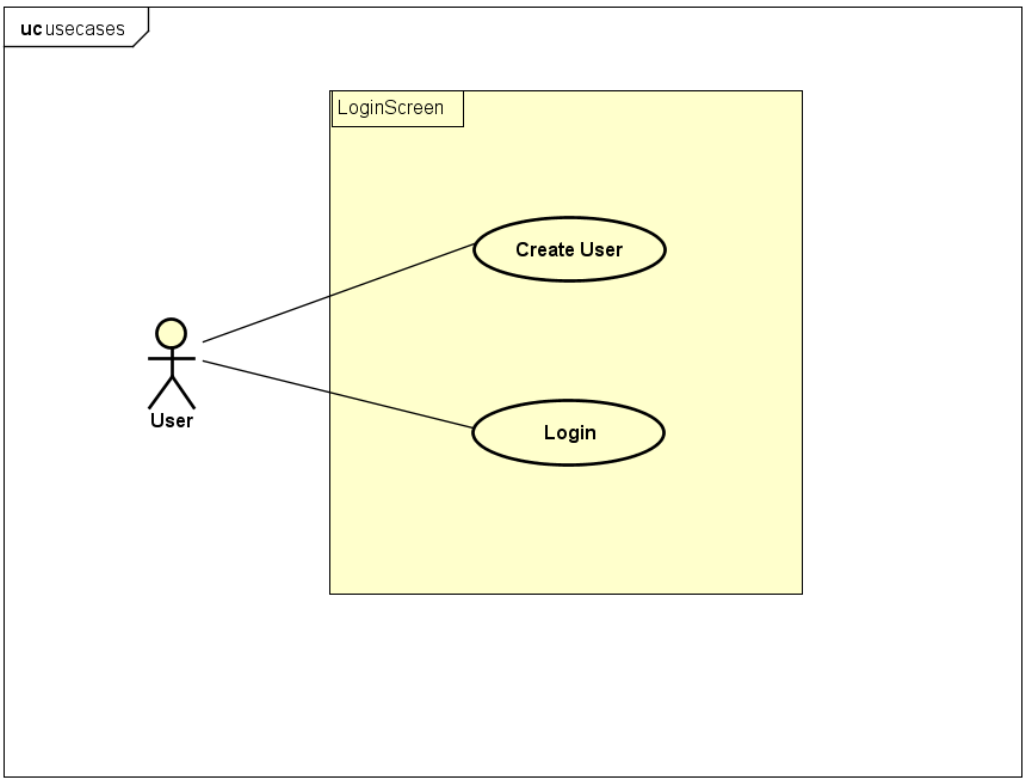


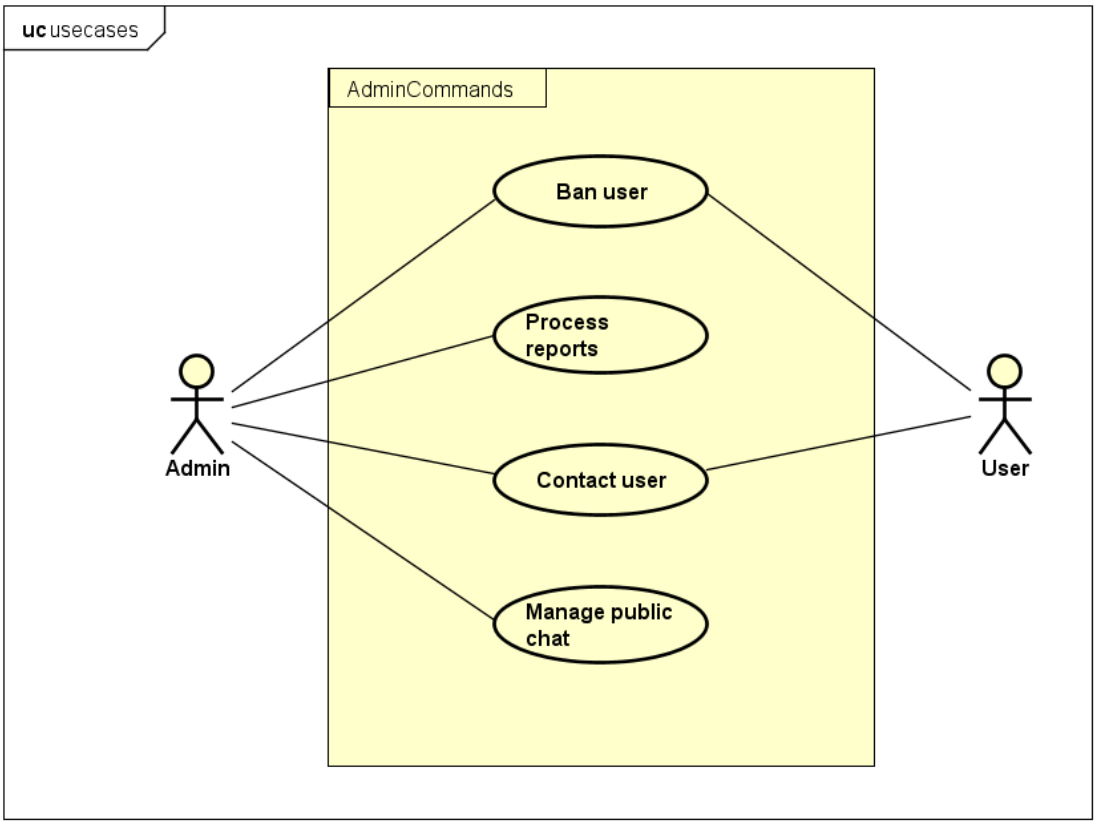

Burndown chart: Working hours spent in the 4. sprint
400
300
200
100
0
Timer tilbage
15/11-18
20/11-18
21/11-18
22/11-18
27/11-18
28/11-18
29/11-18
Date

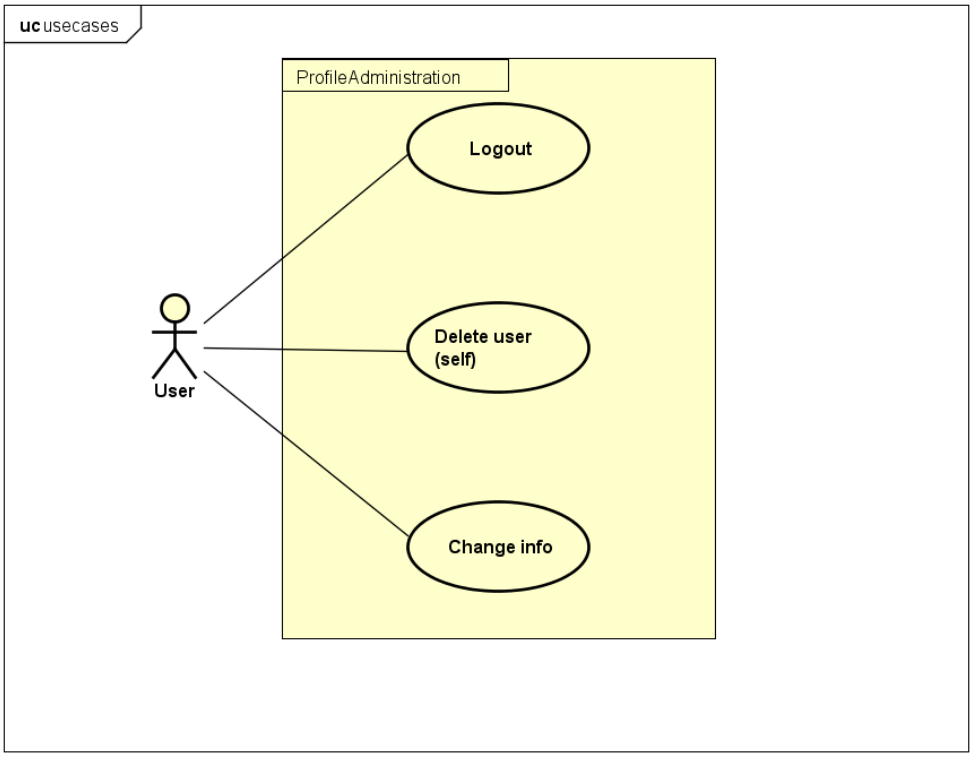

Burndown chart: Working hours spent in the 5. sprint
500
400
300
200
100
0
Timer tilbage
4/12-18
5/12-18
6/12-18
7/12-18
11/12-18
Date

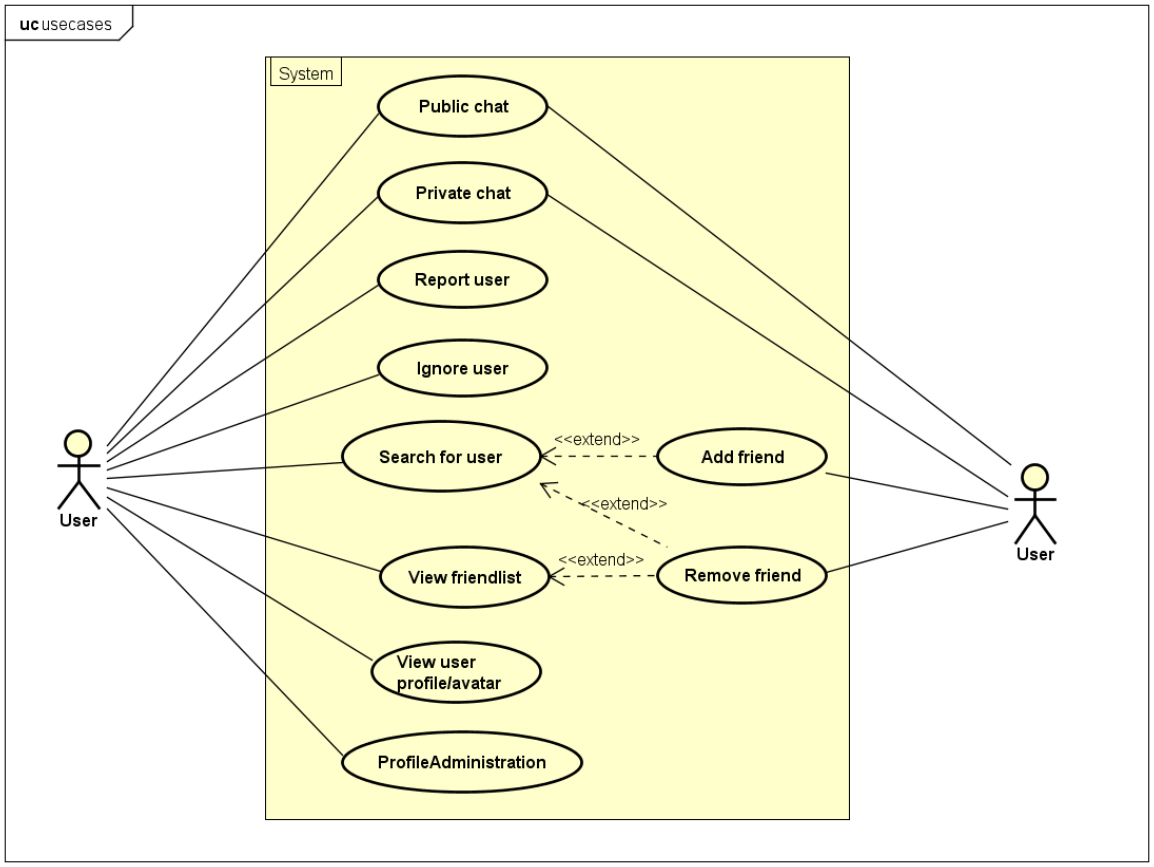
A8 Use cases

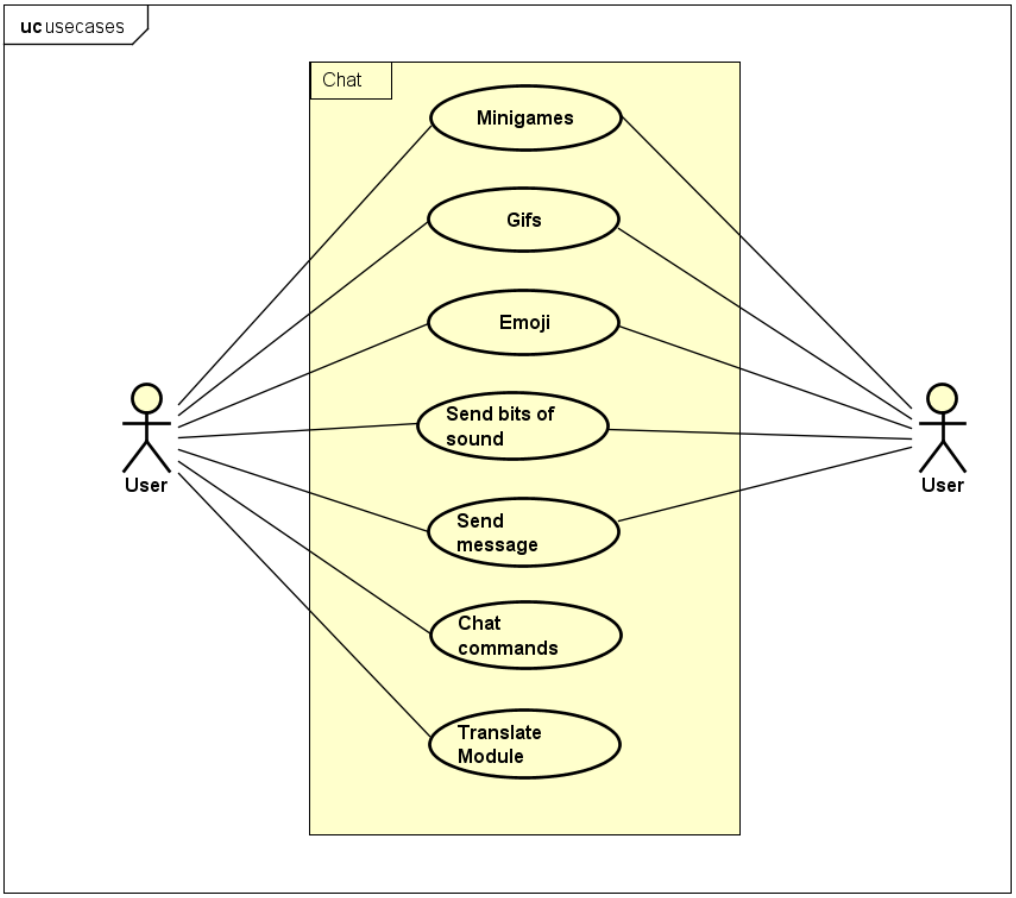

uc usecases
SYNchat
Login Screen
System
<<include>>
ProfileAdminis
tration
Chat
AdminComma
nds
User
Admin


uc usecases
LoginScreen
Create User
Login
User


uc usecases
AdminCommands
Ban user
Process reports
Contact user
Manage public chat
Admin
User


uc usecases
ProfileAdministration
Logout
Delete user (self)
Change info
User


uc usecases
System
Public chat
Private chat
Report user
Ignore user
Search for user
Add friend
<<extend>>
<<extend>>
View friendlist
<<extend>>
Remove friend
View user profile/avatar
ProfileAdministration
User
User


uc usecases
Chat
Minigames
Gifs
Emoji
Send bits of sound
Send message
Chat commands
Translate Module
User
User

A9 Analysis class diagram

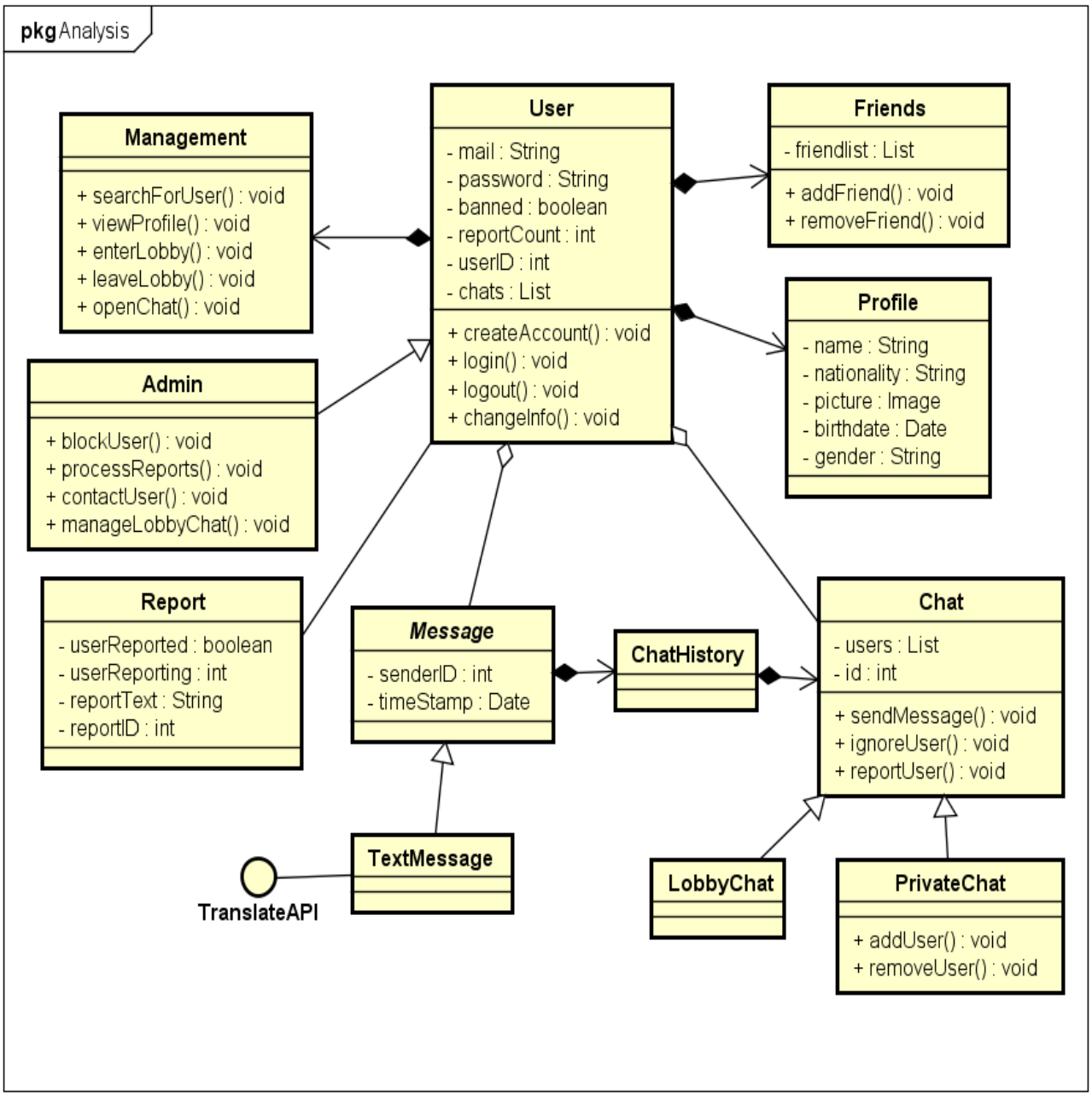
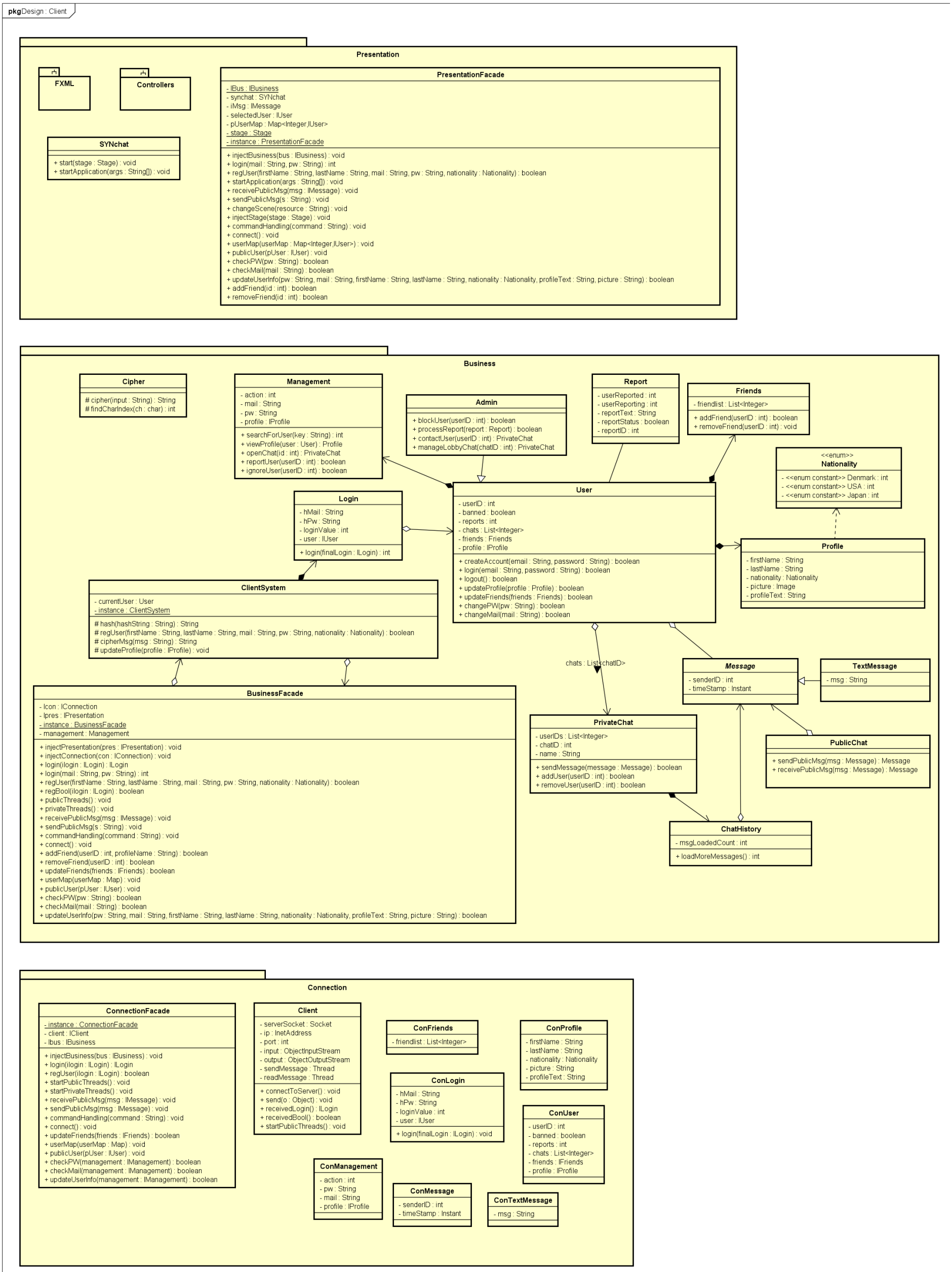
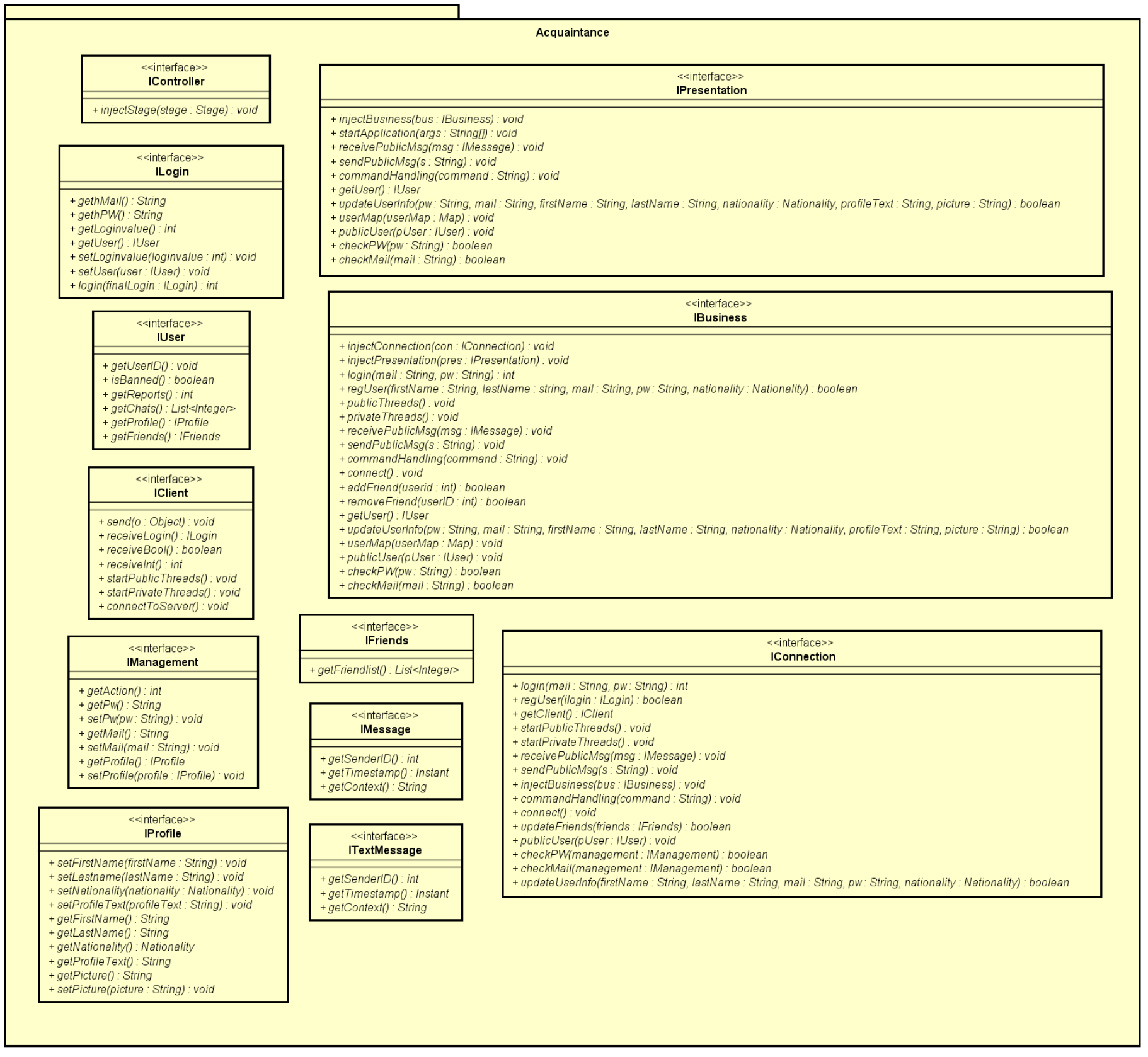
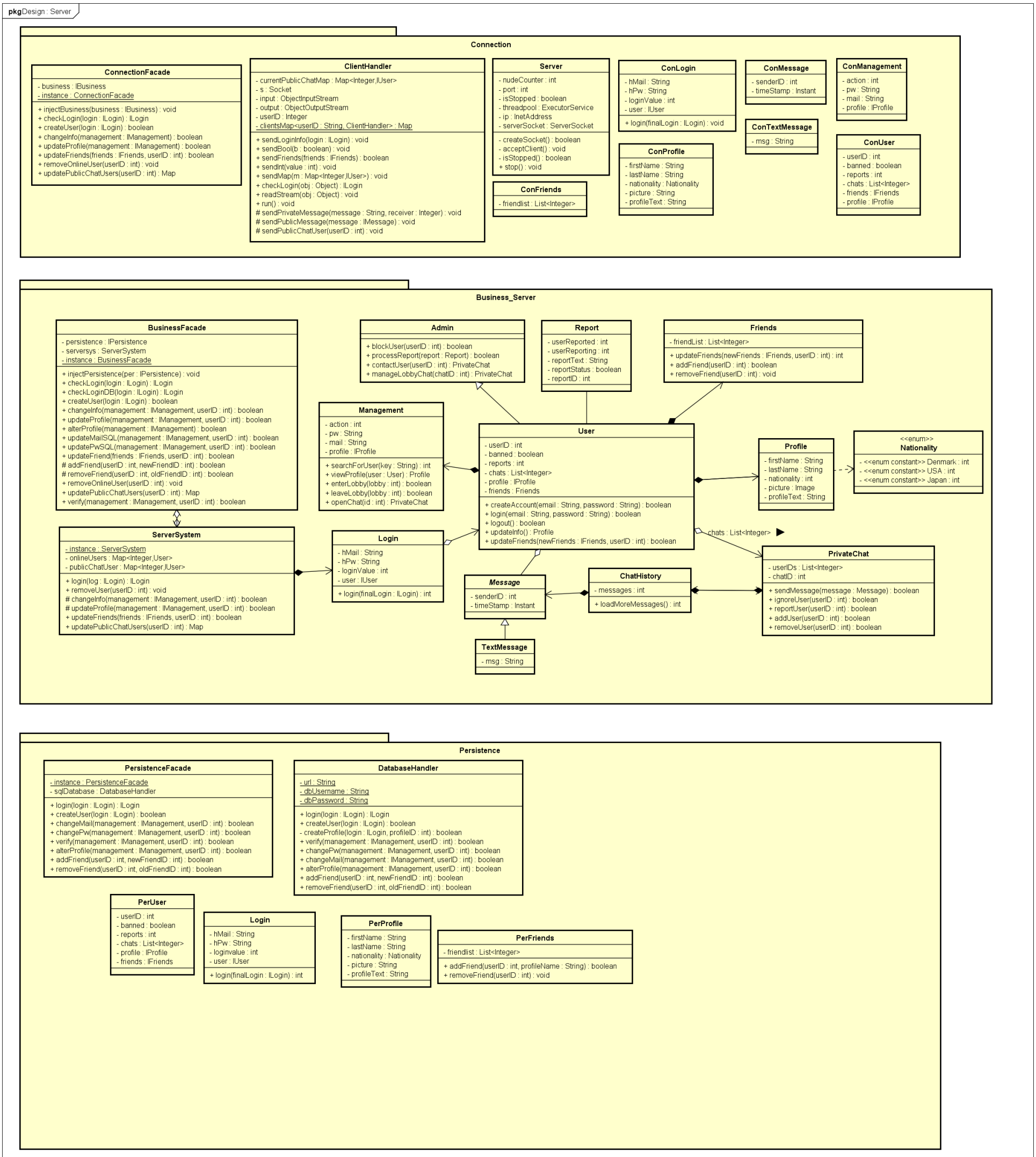


Figure 4 1 Analysis diagram of the system.

A10 Design class diagram




Acquaintance
<<interface>>
IController
+ injectStage(stage : Stage) : void
<<interface>>
ILogin
+ gethMail() : String
+ gethPW() : String
+ getLoginvalue() : int
+ getUser() : IUser
+ setLoginvalue(loginvalue : int) : void
+ setUser(user : IUser) : void
+ login(finalLogin : ILogin) : int
<<interface>>
IUser
+ getUserID() : void
+ isBanned() : boolean
+ getReports() : int
+ getChats() : List<Integer>
+ getProfile() : IProfile
+ getFriends() : IFriends
<<interface>>
IClient
+ send(o : Object) : void
+ receiveLogin() : ILogin
+ receiveBool() : boolean
+ receiveInt() : int
+ startPublicThreads() : void
+ startPrivateThreads() : void
+ connectToServer() : void
<<interface>>
IManagement
+ getAction() : int
+ getPw() : String
+ setPw(pw : String) : void
+ getMail() : String
+ setMail(mail : String) : void
+ getProfile() : IProfile
+ setProfile(profile : IProfile) : void
<<interface>>
IProfile
+ setFirstName(firstName : String) : void
+ setLastname(lastName : String) : void
+ setNationality(nationality : Nationality) : void
+ setProfileText(profileText : String) : void
+ getFirstName() : String
+ getLastName() : String
+ getNationality() : Nationality
+ getProfileText() : String
+ getPicture() : String
+ setPicture(picture : String) : void
<<interface>>
IPresentation
+ injectBusiness(bus : IBusiness) : void
+ startApplication(args : String[]) : void
+ receivePublicMsg(msg : IMessage) : void
+ sendPublicMsg(s : String) : void
+ commandHandling(command : String) : void
+ getUser() : IUser
+ updateUserInfo(pw : String, mail : String, firstName : String, lastName : String, nationality : Nationality, profileText : String, picture : String) : boolean
+ userMap(userMap : Map) : void
+ publicUser(pUser : IUser) : void
+ checkPW(pw : String) : boolean
+ checkMail(mail : String) : boolean
<<interface>>
IBusiness
+ injectConnection(con : IConnection) : void
+ injectPresentation(pres : IPresentation) : void
+ login(mail : String, pw : String) : int
+ regUser(firstName : String, lastName : string, mail : String, pw : String, nationality : Nationality) : boolean
+ publicThreads() : void
+ privateThreads() : void
+ receivePublicMsg(msg : IMessage) : void
+ sendPublicMsg(s : String) : void
+ commandHandling(command : String) : void
+ connect() : void
+ addFriend(userid : int) : boolean
+ removeFriend(userID : int) : boolean
+ getUser() : IUser
+ updateUserInfo(pw : String, mail : String, firstName : String, lastName : String, nationality : Nationality, profileText : String, picture : String) : boolean
+ userMap(userMap : Map) : void
+ publicUser(pUser : IUser) : void
+ checkPW(pw : String) : boolean
+ checkMail(mail : String) : boolean
<<interface>>
IFriends
+ getFriendlist() : List<Integer>
<<interface>>
IMessage
+ getSenderID() : int
+ getTimestamp() : Instant
+ getContext() : String
<<interface>>
ITextMessage
+ getSenderID() : int
+ getTimestamp() : Instant
+ getContext() : String
<<interface>>
IConnection
+ login(mail : String, pw : String) : int
+ regUser(ilogin : ILogin) : boolean
+ getClient() : IClient
+ startPublicThreads() : void
+ startPrivateThreads() : void
+ receivePublicMsg(msg : IMessage) : void
+ sendPublicMsg(s : String) : void
+ injectBusiness(bus : IBusiness) : void
+ commandHandling(command : String) : void
+ connect() : void
+ updateFriends(friends : IFriends) : boolean
+ publicUser(pUser : IUser) : void
+ checkPW(management : IManagement) : boolean
+ checkMail(management : IManagement) : boolean
+ updateUserInfo(firstName : String, lastName : String, mail : String, pw : String, nationality : Nationality) : boolean


pkgDesign : Server
Connection
ConnectionFacade
- business : IBusiness
- instance : ConnectionFacade
+ injectBusiness(business : IBusiness) : void
+ checkLogin(login : ILogin) : ILogin
+ createUser(login : ILogin) : boolean
+ changeInfo(management : IManagement) : boolean
+ updateProfile(management : IManagement) : boolean
+ updateFriends(friends : IFriends, userID : int) : boolean
+ removeOnlineUser(userID : int) : void
+ updatePublicChatUsers(userID : int) : Map
ClientHandler
- currentPublicChatMap : Map<Integer,IUser>
- s : Socket
- input : ObjectInputStream
- output : ObjectOutputStream
- userID : Integer
- clientsMap<userID : String, ClientHandler> : Map
+ sendLoginInfo(login : ILogin) : void
+ sendBool(b : boolean) : void
+ sendFriends(friends : IFriends) : boolean
+ sendInt(value : int) : void
+ sendMap(m : Map<Integer,IUser>) : void
+ checkLogin(obj : Object) : ILogin
+ readStream(obj : Object) : void
+ run() : void
# sendPrivateMessage(message : String, receiver : Integer) : void
# sendPublicMessage(message : IMessage) : void
# sendPublicChatUser(userID : int) : void
Server
- nudeCounter : int
- port : int
- isStopped : boolean
- threadpool : ExecutorService
- ip : InetAddress
- serverSocket : ServerSocket
- createSocket() : boolean
- acceptClient() : void
- isStopped() : boolean
+ stop() : void
ConFriends
- friendlist : List<Integer>
ConLogin
- hMail : String
- hPw : String
- loginValue : int
- user : IUser
+ login(finalLogin : ILogin) : void
ConProfile
- firstName : String
- lastName : String
- nationality : Nationality
- picture : String
- profileText : String
ConMessage
- senderID : int
- timeStamp : Instant
ConTextMessage
- msg : String
ConManagement
- action : int
- pw : String
- mail : String
- profile : IProfile
ConUser
- userID : int
- banned : boolean
- reports : int
- chats : List<Integer>
- friends : IFriends
- profile : IProfile
Business_Server
BusinessFacade
- persistence : IPersistence
- serversys : ServerSystem
- instance : BusinessFacade
+ injectPersistence(per : IPersistence) : void
+ checkLogin(login : ILogin) : ILogin
+ checkLoginDB(login : ILogin) : ILogin
+ createUser(login : ILogin) : boolean
+ changeInfo(management : IManagement, userID : int) : boolean
+ updateProfile(management : IManagement, userID : int) : boolean
+ alterProfile(management : IManagement) : boolean
+ updateMailSQL(management : IManagement, userID : int) : boolean
+ updatePwSQL(management : IManagement, userID : int) : boolean
+ updateFriend(friends : IFriends, userID : int) : boolean
# addFriend(userID : int, newFriendID : int) : boolean
# removeFriend(userID : int, oldFriendID : int) : boolean
+ removeOnlineUser(userID : int) : void
+ updatePublicChatUsers(userID : int) : Map
+ verify(management : IManagement, userID : int) : boolean
Admin
+ blockUser(userID : int) : boolean
+ processReport(report : Report) : boolean
+ contactUser(userID : int) : PrivateChat
+ manageLobbyChat(chatID : int) : PrivateChat
Report
- userReported : int
- userReporting : int
- reportText : String
- reportStatus : boolean
- reportID : int
Friends
- friendList : List<Integer>
+ updateFriends(newFriends : IFriends, userID : int) : int
+ addFriend(userID : int) : boolean
+ removeFriend(userID : int) : void
Management
- action : int
- pw : String
- mail : String
- profile : IProfile
+ searchForUser(key : String) : int
+ viewProfile(user : User) : Profile
+ enterLobby(lobby : int) : boolean
+ leaveLobby(lobby : int) : boolean
+ openChat(id : int) : PrivateChat
User
- userID : int
- banned : boolean
- reports : int
- chats : List<Integer>
- profile : IProfile
- friends : Friends
+ createAccount(email : String, password : String) : boolean
+ login(email : String, password : String) : boolean
+ logout() : boolean
+ updateInfo() : Profile
+ updateFriends(newFriends : IFriends, userID : int) : boolean
Profile
- firstName : String
- lastName : String
- nationality : int
- picture : Image
- profileText : String
<<enum>>
Nationality
- <<enum constant>> Denmark : int
- <<enum constant>> USA : int
- <<enum constant>> Japan : int
chats : List<Integer>
ServerSystem
- instance : ServerSystem
- onlineUsers : Map<Integer,User>
- publicChatUser : Map<Integer,IUser>
+ login(log : ILogin) : ILogin
+ removeUser(userID : int) : void
# changeInfo(management : IManagement, userID : int) : boolean
# updateProfile(management : IManagement, userID : int) : boolean
+ updateFriends(friends : IFriends, userID : int) : boolean
+ updatePublicChatUsers(userID : int) : Map
Login
- hMail : String
- hPw : String
- loginValue : int
- user : IUser
+ login(finalLogin : ILogin) : int
Message
- senderID : int
- timeStamp : Instant
ChatHistory
- messages : int
+ loadMoreMessages() : int
PrivateChat
- userIDs : List<Integer>
- chatID : int
+ sendMessage(message : Message) : boolean
+ ignoreUser(userID : int) : boolean
+ reportUser(userID : int) : boolean
+ addUser(userID : int) : boolean
+ removeUser(userID : int) : boolean
TextMessage
- msg : String
Persistence
PersistenceFacade
- instance : PersistenceFacade
- sqlDatabase : DatabaseHandler
+ login(login : ILogin) : ILogin
+ createUser(login : ILogin) : boolean
+ changeMail(management : IManagement, userID : int) : boolean
+ changePw(management : IManagement, userID : int) : boolean
+ verify(management : IManagement, userID : int) : boolean
+ alterProfile(management : IManagement, userID : int) : boolean
+ addFriend(userID : int, newFriendID : int) : boolean
+ removeFriend(userID : int, oldFriendID : int) : boolean
DatabaseHandler
- url : String
- dbUsername : String
- dbPassword : String
+ login(login : ILogin) : ILogin
+ createUser(login : ILogin) : boolean
- createProfile(login : ILogin, profileID : int) : boolean
+ verify(management : IManagement, userID : int) : boolean
+ changePw(management : IManagement, userID : int) : boolean
+ changeMail(management : IManagement, userID : int) : boolean
+ alterProfile(management : IManagement, userID : int) : boolean
+ addFriend(userID : int, newFriendID : int) : boolean
+ removeFriend(userID : int, oldFriendID : int) : boolean
PerUser
- userID : int
- banned : boolean
- reports : int
- chats : List<Integer>
- profile : IProfile
- friends : IFriends
Login
- hMail : String
- hPw : String
- loginvalue : int
- user : IUser
+ login(finalLogin : ILogin) : int
PerProfile
- firstName : String
- lastName : String
- nationality : Nationality
- picture : String
- profileText : String
PerFriends
- friendlist : List<Integer>
+ addFriend(userID : int, profileName : String) : boolean
+ removeFriend(userID : int) : void

A11 E/R Diagram

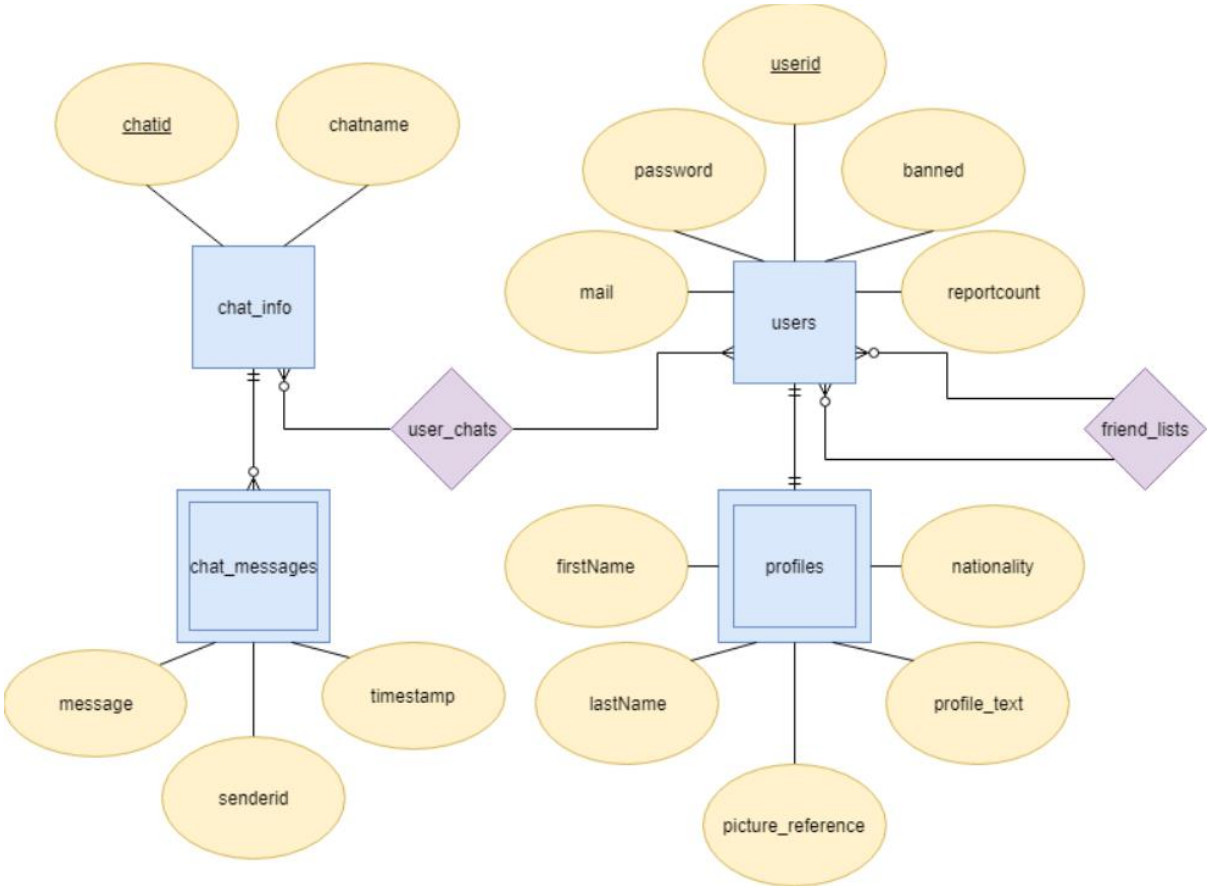
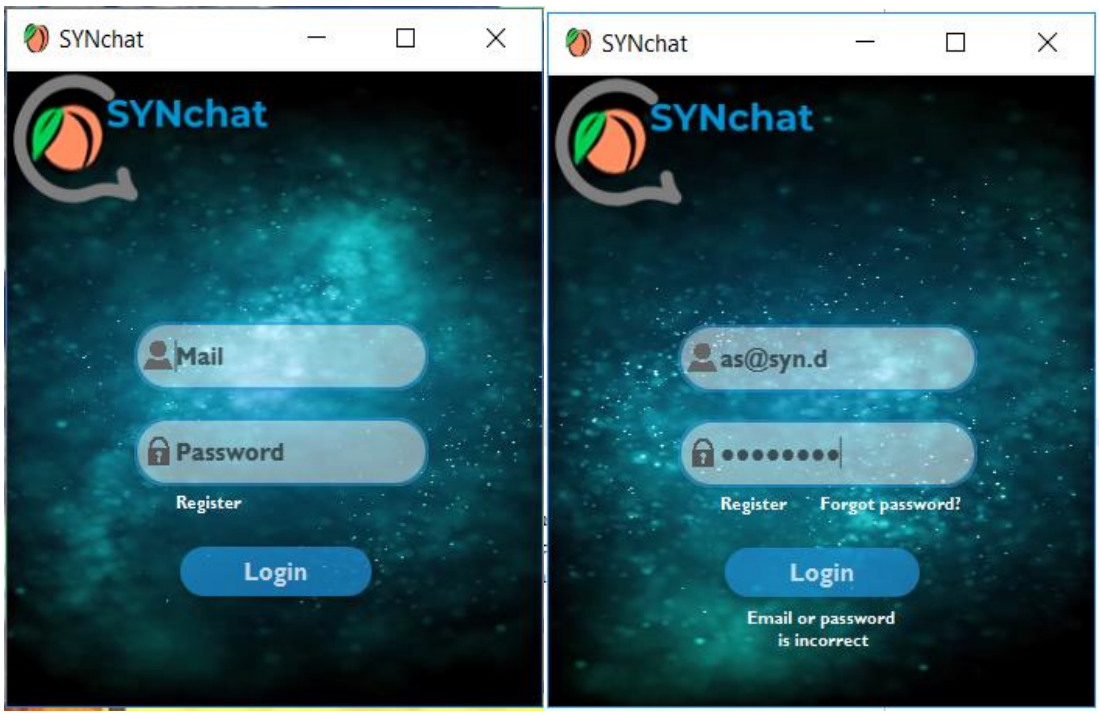


Figure 4.6 1 E/R diagram, visual presentation of entities and relations in database.

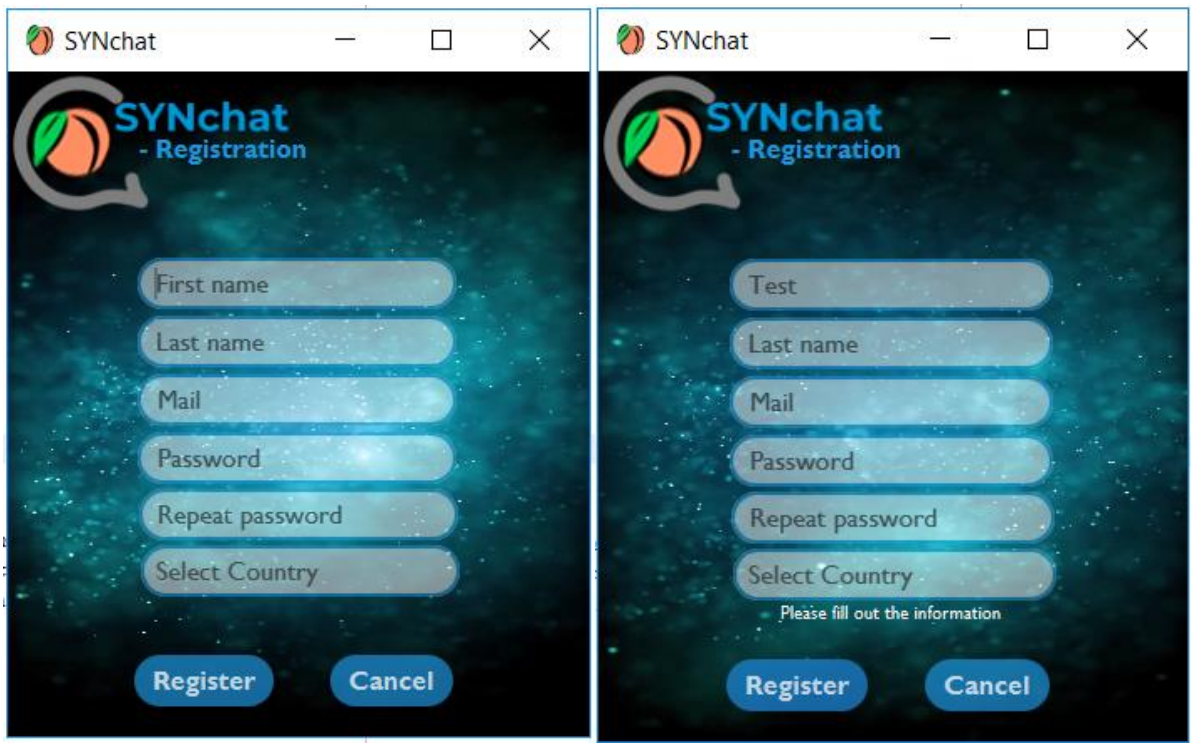
A12 System images

A12.1 Login Screen



At the login screen, error messages will be displayed in scenarios such as if the user missed to fill out or typed in an incorrect mail, missed to fill out or typed in the incorrect the password, or both.

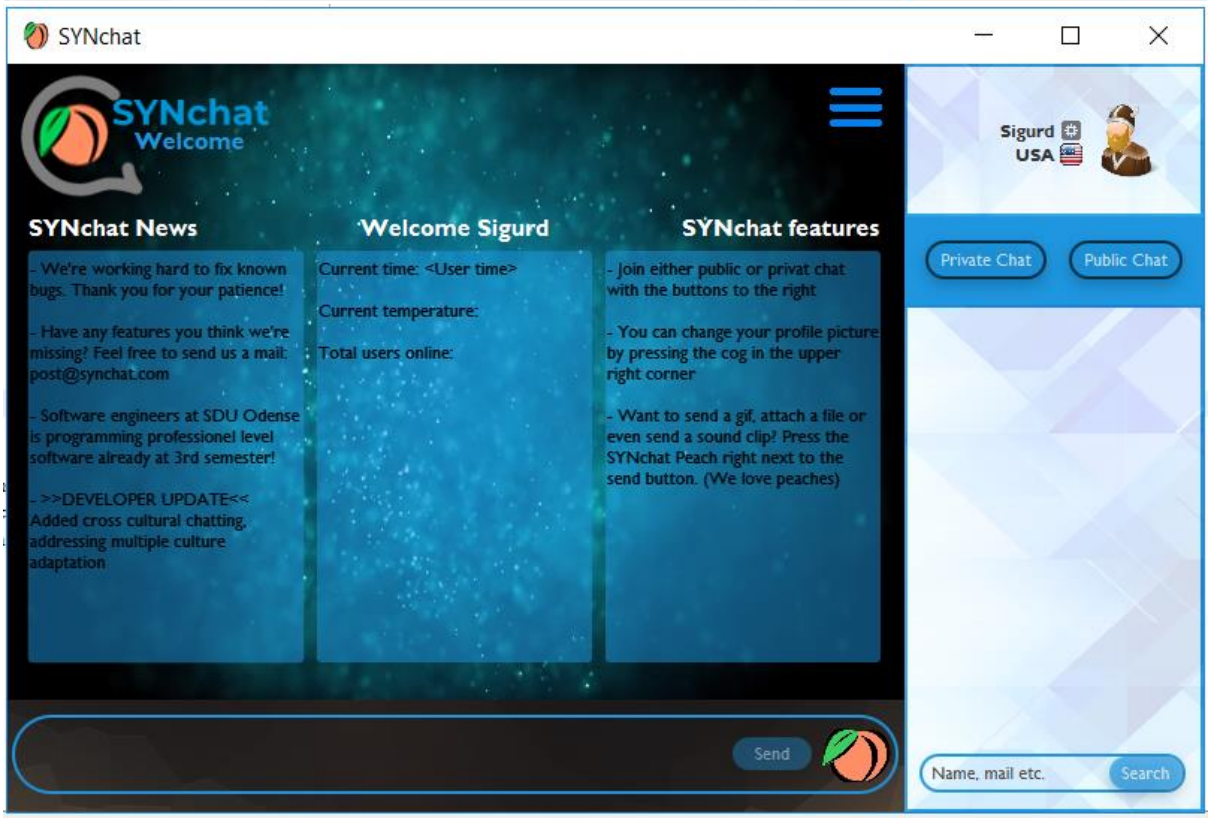
A12.2 Register User

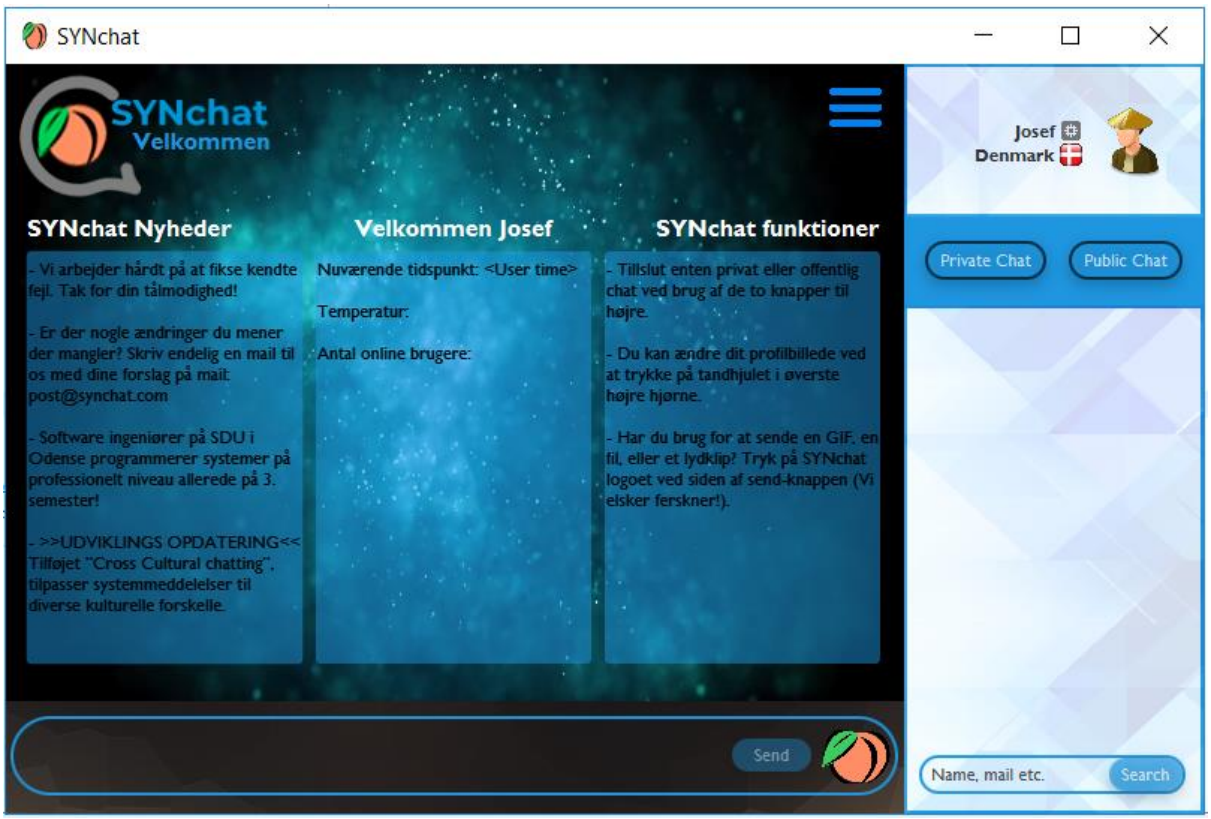


Error messages at the registration would tell the user to fill out all information.

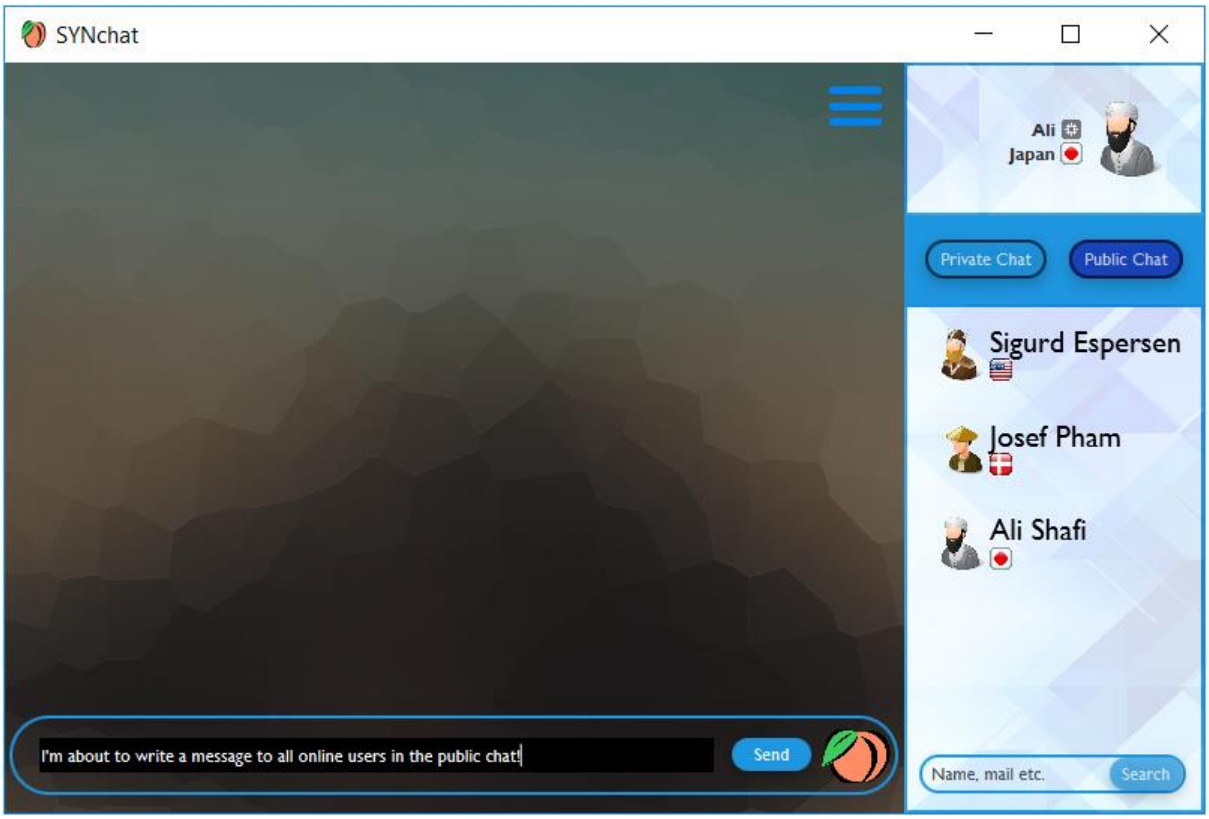
A12.3 Welcome Screen

Welcome screen of three different users of different nationality. The language would change according to the chosen nationality.

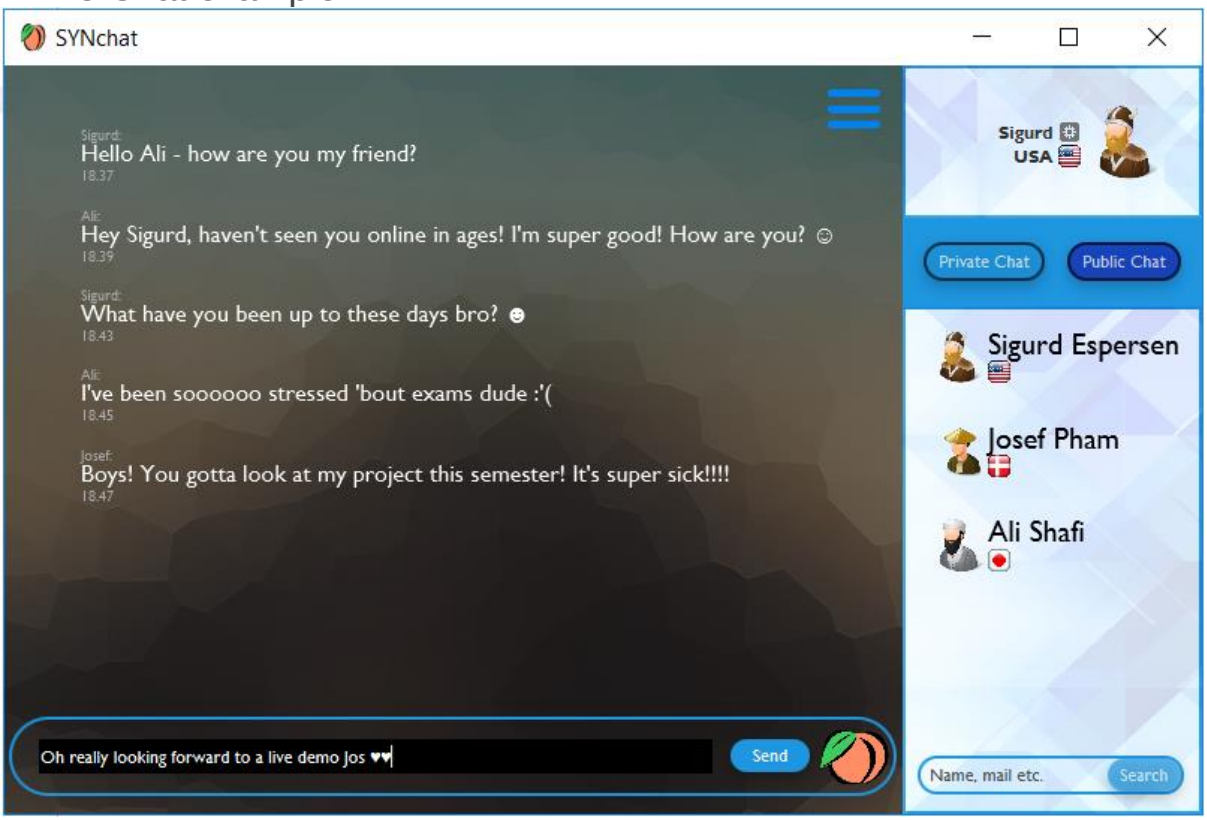


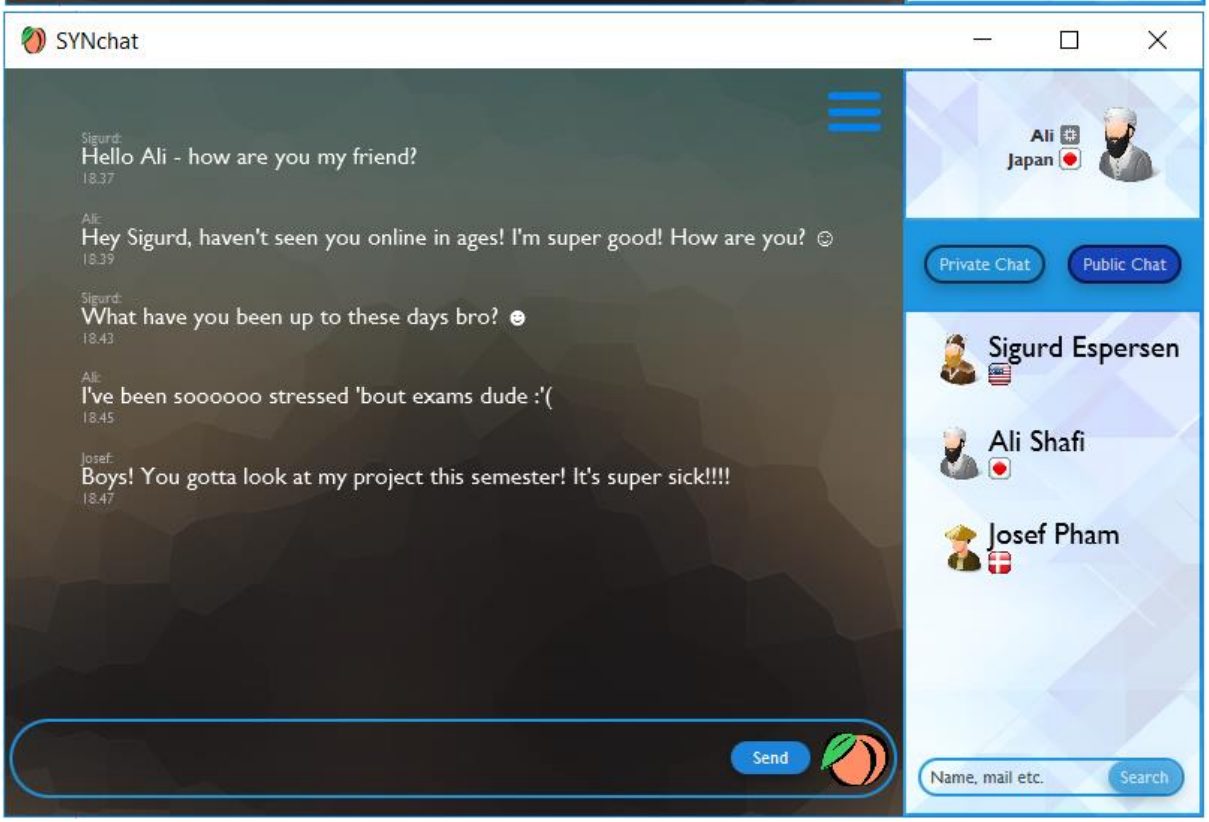


A12.4 Public Chat



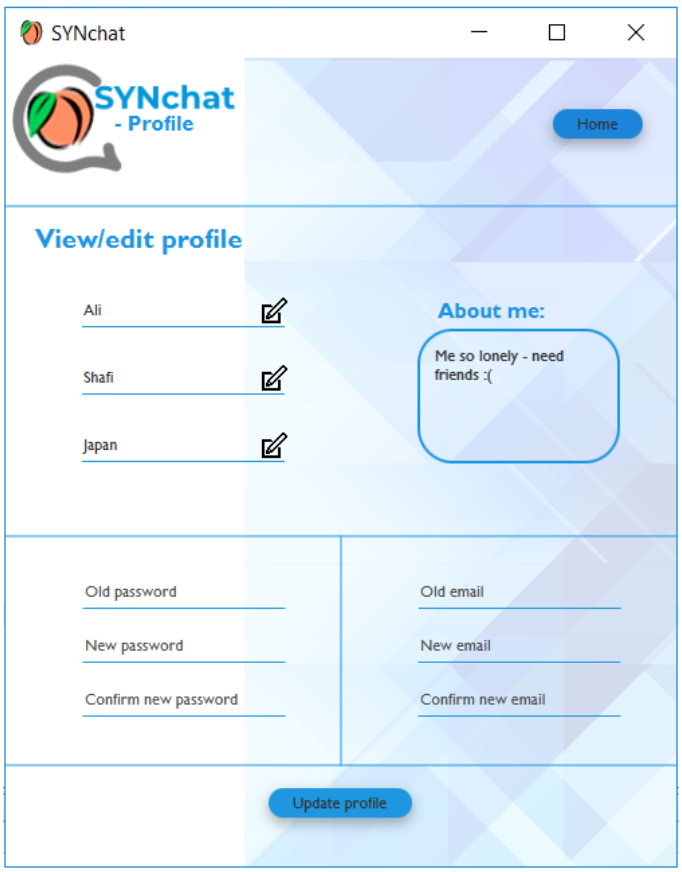
A12.5 Chat example





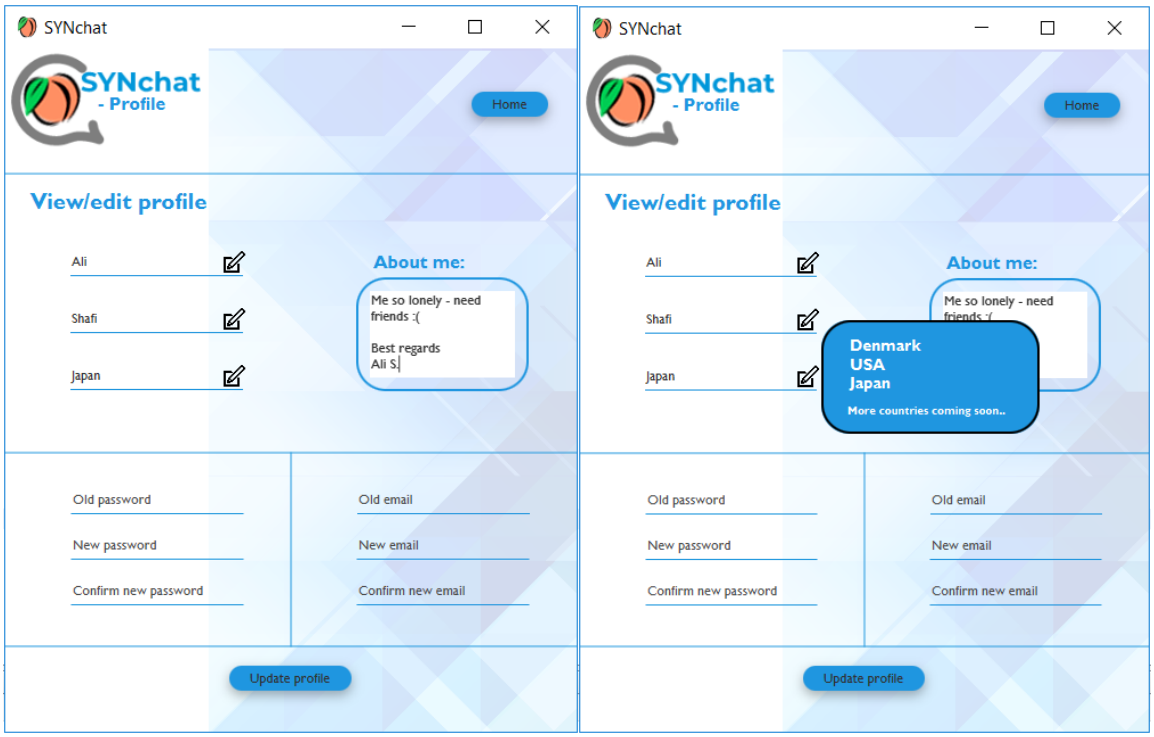
A12.6 View Profile

When you click at the icon beside your avatar, you will get the options to either change profile picture (avatar) or view your profile (the next image below)



A12.7 Edit Profile

In this window you can edit various information on your profile: name, nationality, password, email, or profile text.



The icon beside the avatar gives you the options to either view the profile, or change the profile picture. When choosing to change the avatar, you can scroll through around 12 pictures, and further customize the profile.

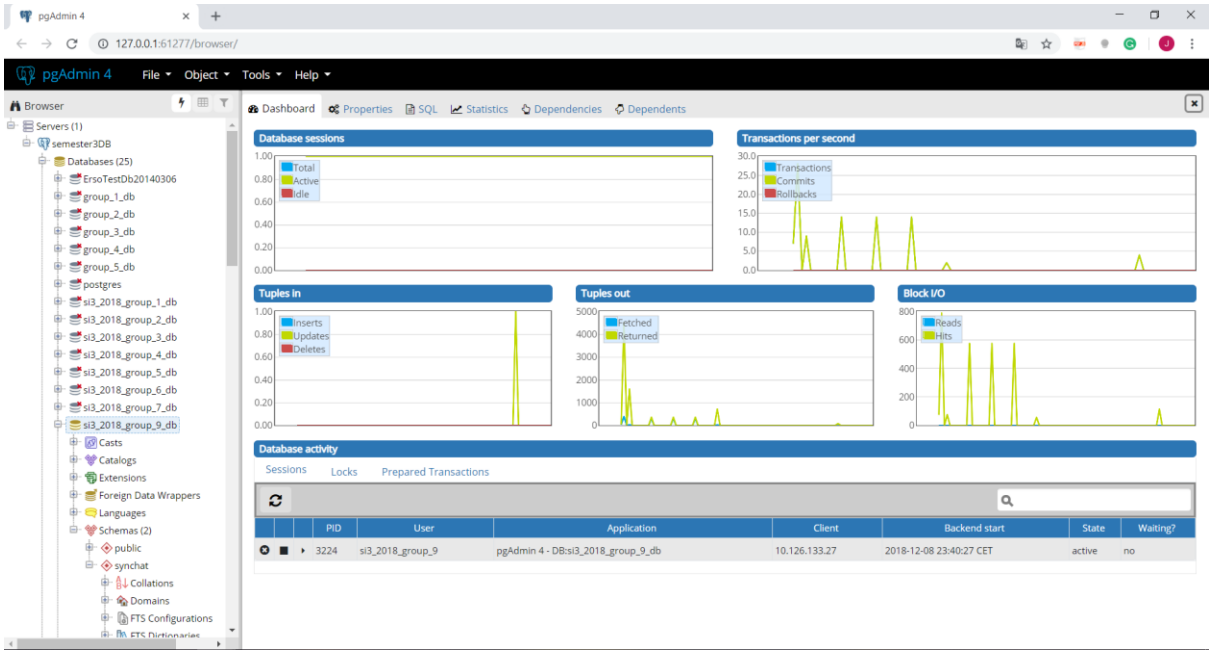
A13 pgAdmin

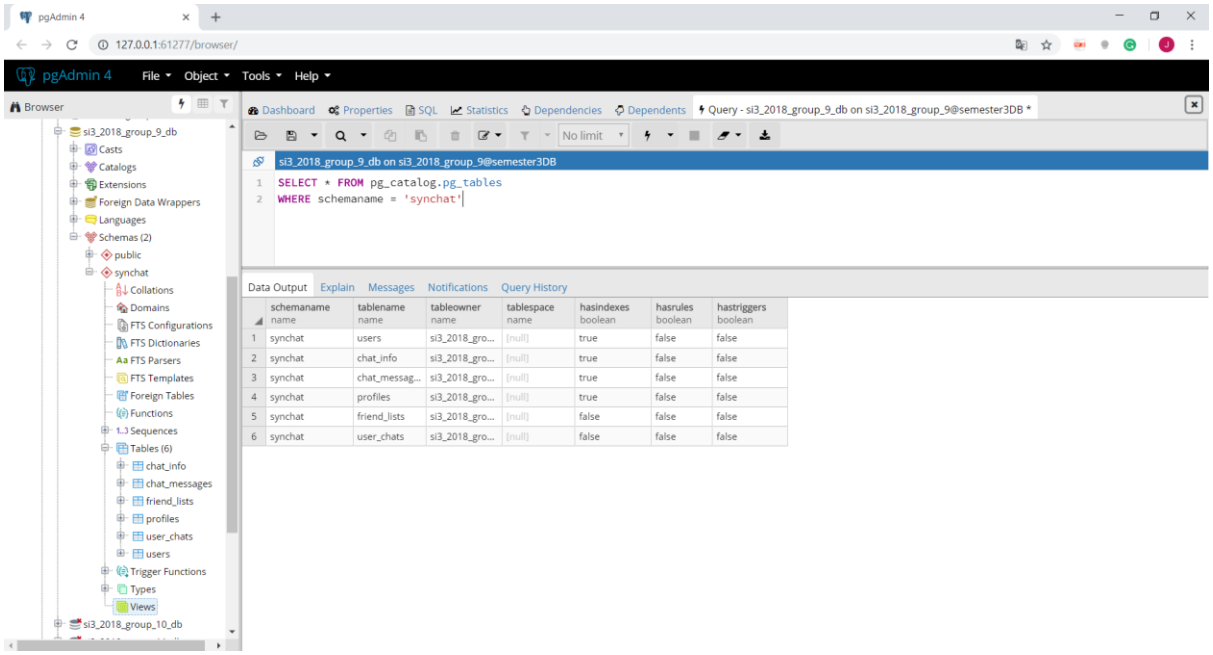
Postresql database viewer tek-mmmi-db0a.tek.c.sdu.dk

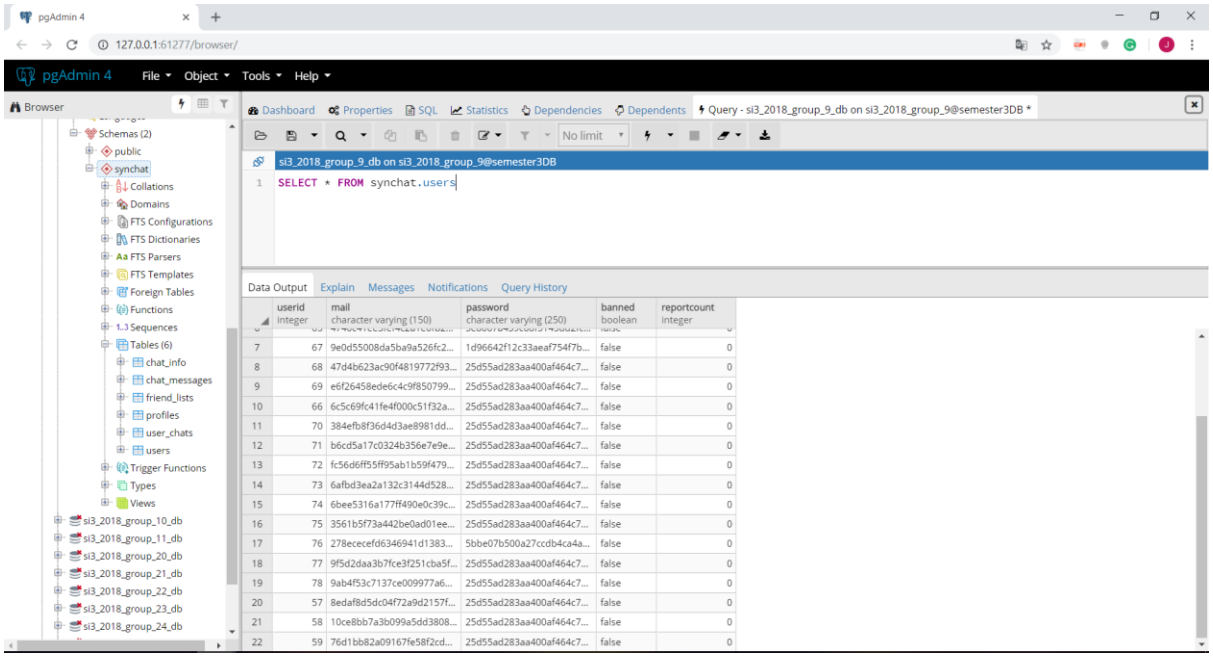
Group ID: si3_2018_group_9_db

Password: copt22*viols

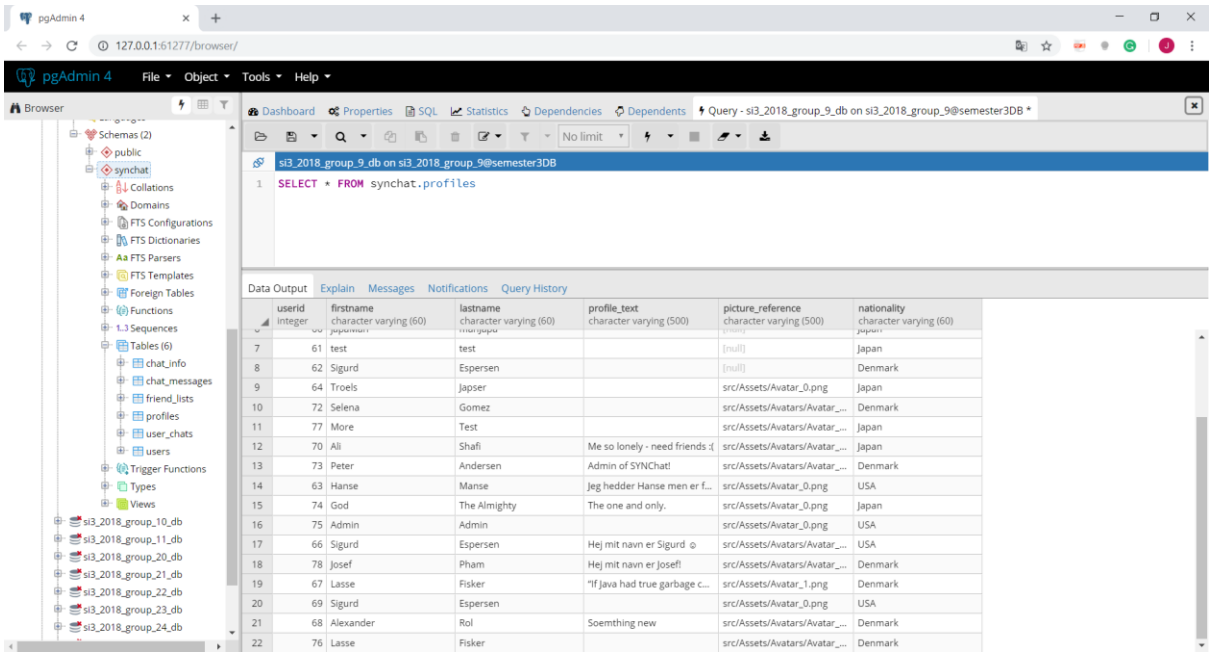
Below is screen dumps from pgAdmin, where you can see the database activity, all the schemas and tables in the database, and finally all users and profiles stored in the database.







	userid integer	mail character varying (150)	password character varying (250)	banned boolean	reportcount integer
7	67	9e0d55008da5ba9a526fc2...	1d96642f12c33aeaf754f7b...	false	0
8	68	47d4b623ac90f4819772f93...	25d55ad283aa400af464c7...	false	0
9	69	e6f26458ede6c4c9f850799...	25d55ad283aa400af464c7...	false	0
10	66	6c5c69fc41fe4f000c51f32a...	25d55ad283aa400af464c7...	false	0
11	70	384efb8f36d4d3ae8981dd...	25d55ad283aa400af464c7...	false	0
12	71	b6cd5a17c0324b356e7e9e...	25d55ad283aa400af464c7...	false	0
13	72	fc56d6ff55ff95ab1b59f479...	25d55ad283aa400af464c7...	false	0
14	73	6afbd3ea2a132c3144d528...	25d55ad283aa400af464c7...	false	0
15	74	6bee5316a177ff490e0c39c...	25d55ad283aa400af464c7...	false	0
16	75	3561b5f73a442be0ad01ee...	25d55ad283aa400af464c7...	false	0
17	76	278ececefd6346941d1383...	5bbe07b500a27ccdb4ca4a...	false	0
18	77	9f5d2daa3b7fce3f251cba5f...	25d55ad283aa400af464c7...	false	0
19	78	9ab4f53c7137ce009977a6...	25d55ad283aa400af464c7...	false	0
20	57	8edaf8d5dc04f72a9d2157f...	25d55ad283aa400af464c7...	false	0
21	58	10ce8bb7a3b099a5dd3808...	25d55ad283aa400af464c7...	false	0
22	59	76d1bb82a09167fe58f2cd...	25d55ad283aa400af464c7...	false	0

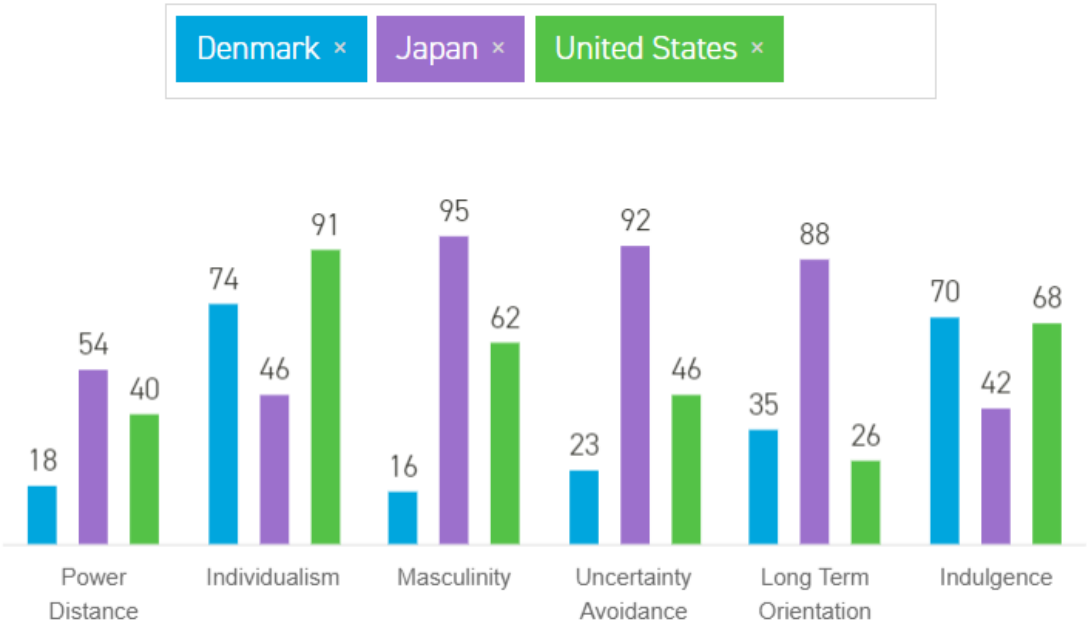


	userid integer	firstname character varying (60)	lastname character varying (60)	profile_text character varying (500)	picture_reference character varying (500)	nationality character varying (60)
7	61	test	test		[null]	Japan
8	62	Sigurd	Espersen		[null]	Denmark
9	64	Troels	Japser		src/Assets/Avatar_0.png	Japan
10	72	Selena	Gomez		src/Assets/Avatars/Avatar_...	Denmark
11	77	More	Test		src/Assets/Avatars/Avatar_...	Japan
12	70	Ali	Shafi	Me so lonely - need friends :(	src/Assets/Avatars/Avatar_...	Japan
13	73	Peter	Andersen	Admin of SYNChat!	src/Assets/Avatars/Avatar_...	Denmark
14	63	Hanse	Manse	Jeg hedder Hanse men er f...	src/Assets/Avatar_0.png	USA
15	74	God	The Almighty	The one and only.	src/Assets/Avatar_0.png	Japan
16	75	Admin	Admin		src/Assets/Avatar_0.png	USA
17	66	Sigurd	Espersen	Hej mit navn er Sigurd ☺	src/Assets/Avatars/Avatar_...	USA
18	78	Josef	Pham	Hej mit navn er Josef!	src/Assets/Avatars/Avatar_...	Denmark
19	67	Lasse	Fisker	"If Java had true garbage c...	src/Assets/Avatar_1.png	Denmark
20	69	Sigurd	Espersen		src/Assets/Avatar_0.png	USA
21	68	Alexander	Rol	Soemthing new	src/Assets/Avatars/Avatar_...	Denmark
22	76	Lasse	Fisker		src/Assets/Avatars/Avatar_...	Denmark

B External Appendices

B1 Geert Hofstede

The Hofstede model of national culture consists of six dimensions. The cultural dimensions represent independent preferences for one state of affairs over another that distinguish countries (rather than individuals) from each other. Following countries, Denmark, Japan and USA have been analyze using Hofstede's model, and can be visualised in the diagram below. The following dimensions are also described below: (Geert Hofstede. (2018). The 6 dimensions model of national culture by Geert Hofstede.)



POWER DISTANCE INDEX (PDI): Power Distance is the extent to which the less powerful members of organizations and institutions (like the family) accept and expect that power is distributed unequally.

INDIVIDUALISM VERSUS COLLECTIVISM (IDV): Individualism is the extent to which people feel independent, as opposed to being interdependent as members of larger wholes.

MASCULINITY VERSUS FEMININITY (MAS): Masculinity is the extent to which the use of force in endorsed socially. In a feminine society, the genders are emotionally closer. Competing is not so openly endorsed, and there is sympathy for the underdog.

UNCERTAINTY AVOIDANCE INDEX (UAI): Uncertainty avoidance deals with a society's tolerance for uncertainty and ambiguity.

LONG TERM ORIENTATION VERSUS SHORT TERM NORMATIVE ORIENTATION (LTO): Long-term orientation deals with change. In a long-time-oriented culture, the basic notion about the world is that it is in flux, and preparing for

the future is always needed. In a short-time-oriented culture, the world is essentially as it was created, so that the past provides a moral compass, and adhering to it is morally good.

INDULGENCE VERSUS RESTRAINT (IND): Indulgence is about the good things in life. In an indulgent culture it is good to be free. Doing what your impulses want you to do, is good. Friends are important and life makes sense. In a restrained culture, the feeling is that life is hard, and duty, not freedom, is the normal state of being.

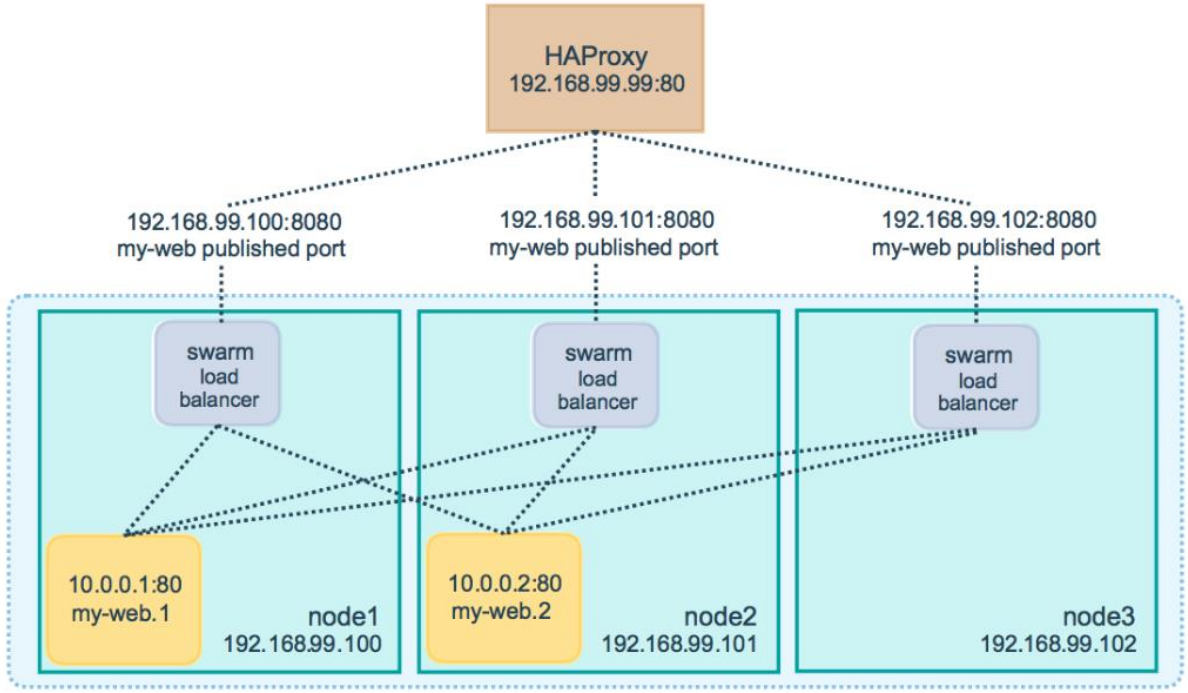
B2 Docker swarm availability table

# of managers	failure tolerance	required for quorum
1	0	1
2	0	2
3	1	2
4	1	3
5	2	3
6	2	4
7	3	4

[1]

[1] "Lecture 11 - Distributed Systems" - Mathias Neerup et al. - OPN slides from lecture 12

B3 External load balancing in Docker



2

[2] https://docs.docker.com/engine/swarm/ingress/#using-the-routing-mesh