
Dokumentation for Marto

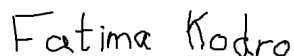
Forfattere:	Alexander Lichtenstein Davidsen, Daniel Pat Hansen, Fatima Kodro, Frederik Kastrup Mortensen, Martin Haugaard Andersen, Søren Bech og Søren Schou Mathiasen
Antal sider:	121
Lokation:	Aarhus Universitet
Vejleder:	Jesper Rosholm Tørresø



Alexander Lichtenstein Davidsen
201608479, IKT



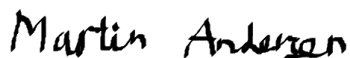
Daniel Pat Hansen
201601915, IKT



Fatima Kodro
201609565, IKT



Frederik Kastrup Mortensen
201607221, IKT



Martin Haugaard Andersen
201605036, IKT



Søren Bech
201604784, IKT



Søren Schou Mathiasen
201605264, IKT

May 29, 2018

Indholdsfortegnelse

1	Indledning	2
1.1	Arbejdsfordeling	2
1.2	Termliste	2
2	Kravspecifikation	3
2.1	Kravspecifikation indledning	3
2.2	Systembeskrivelse	3
2.3	UserStories	3
2.4	Aktør beskrivelse	5
3	Arkitektur	6
3.1	Internet kommunikation	8
3.2	Kommunikation mellem server og database	9
4	Design, implementering & test	16
4.1	MartUI	16
4.2	Server	77
4.3	Database	97
5	Integrationstest	108
6	Accepttestspecifikation	109
6.1	Funktionelle krav	109
6.2	Ikke-funktionelle krav, samt resterende funktionelle krav	120
	Bibliography	121

1 Indledning

Rundt om i verden bliver der hele tiden fundet nye interesse-muligheder, hvilket medfører at der er mange mennesker med meget forskellige hobbyer. Det bliver derfor hele tiden sværere at finde mennesker med samme hobbyer som én selv.

Marto er en chat app, der har med formål at bringe mennesker sammen, der har samme interesser. Uanset om man elsker en bestemt sportsgren, et bestemt videospil eller har en anden hobby, hvor man søger andre mennesker at nyde denne hobby med, kan dette arrangeres på appen. Brugeren kan tilføje nogle tags til sin profil, som kan være interesser og hobbyer. Brugeren kan herefter finde andre brugere med samme tags, og begynde at lære dem at kende.

Marto minder om andre chat programmer, hvor at brugeren kan logge ind med et brugernavn og adgangskode vha. en GUI. Herefter bliver brugeren navigeret ind til selve programmet, hvor det er muligt at redigere sin profil, samt møde andre brugere.

Med udgangspunkt i brugerens behov, vil der blive opstillet en række user stories, som vil beskrive interaktionen mellem brugeren og systemet. Ud fra disse user stories vil der blive oprettet en kravspecifikation, der beskriver en række krav til appen.

1.1 Arbejdsfordeling

Navn	Område
Alexander Lichtenstein Davidsen	Database
Daniel Pat Hansen	Server
Fatima Kodro	GUI
Frederik Kastrup Mortensen	Server
Martin Haugaard Andersen	Server
Søren Bech	Database
Søren Schou Mathiasen	GUI

Tabel 3: Tabel over arbejdsfordeling

1.2 Termliste

Marto er selve produktet.

Tags er en tekst, en bruger kan tilføje til sin profil, som kan være hobbyer og interesser.

UserName er et brugernavn, der bruges for at logge ind, og det er dette brugernavn der identificerer brugerne. Et brugernavn skal være unikt.

2 Kravspecifikation

2.1 Kravspecifikation indledning

Dette dokument omhandler kravspecifikationerne. Kravspecifikationen tager udgangspunkt i kundens krav til projektet, hvordan de vil have det skal se ud og de funktionelle krav. Kunden sætter altså rammerne for projektet som skal følges nøje. Kravspecifikationen er den primære kommunikationsform med kunden, hvor kundens krav skal accepteres og derefter skal kunden acceptere de tolerancer der stilles for projektet. Så snart begge er enige, så kan næste fase påbegyndes.

2.2 Systembeskrivelse

Dette system er opbygget af en grafisk brugergrænseflade der fungerer som klient, en server og en database. Server og database befinder sig på samme computer. Den grafiske brugergrænseflade er måden brugeren tilgår Marto. Brugeren bevæger sig rundt i brugergrænsefladen og bruger dets funktioner. Klienten sender her requests til serveren, der efter behov tilgår databasen og melder tilbage til klienten. Serveren står her for alt kommunikation mellem klient og database, samt fra klient til klient.

2.3 UserStories

I dette afsnit findes den række userstories der tilsammen beskriver brugs scenariet for systemet. Userstories er kombineret med MosCow modellen. Her vægtes vores usecases efter skal/vil/kunne, som er vægtet fra højst til mindst henholdsvis.

2.3.1 Chat User stories

- **Sendte tekstbesked:** Som bruger **skal** jeg kunne sende en teksbesked for at skrive med andre brugere.
- **Modtage tekstbesked:** Som bruger **skal** jeg kunne modtage en teksbesked for at kunne læse hvad andre brugere skriver til mig.
- **Se tidligere beskeder:** Som bruger **vil** jeg kunne se tidligere beskeder med andre brugere, for at have en historik over samtaler.
- **Sendte emojis:** Som bruger **vil** jeg kunne sende emojis for hurtigt og nemt at vise mit humør.
- **Modtage emojis:** Som bruger **vil** jeg kunne modtage emojis for at kunne se andre brugeres humør.

- **Sende billeder:** Som bruger **skal** jeg kunne sende billeder for at hurtigt at vise andre brugere mine billeder.
- **Modtage billeder:** Som bruger **skal** jeg kunne modtage billeder for at kunne se andre brugeres billeder.
- **VOIP(Voice Over Internet Protocol):** Som bruger **vil** jeg kunne tale med andre brugere over internettet for at kunne kommunikere med andre brugere uden tekst.
- **Gruppechat:** Som bruger **skal** jeg kunne oprette en gruppechat for nemt at kunne kommunikere med flere brugere på samme tid.

2.3.2 Profil User stories

- **Log ind:** Som bruger **skal** jeg kunne logge ind med mit brugerdefinerede brugernavn og password, for at få adgang til applikationens funktionalitet.
- **Opret profil:** Som bruger **skal** jeg gerne kunne oprette en profil med et unikt brugernavn og et password, der giver mig mulighed for at logge ind på applikationen.
- **Fjern profil:** Som bruger **vil** jeg gerne kunne slette min profil, så mit brugernavn igen bliver ledigt.
- **Søg efter profilnavn:** Som bruger **skal** jeg gerne kunne søge efter en profil, baseret på brugernavn, så jeg kan se profilsiden på denne profil.
- **Søg efter tag:** Som bruger **skal** jeg gerne kunne søge efter en profil, baseret på et tag, så jeg kan se profilsiden på denne profil.
- **Tilføj ven:** Som bruger **skal** jeg gerne kunne tilføje en ven, gennem deres profilside, så jeg kan kommunikere med denne profil.
- **Se venneliste:** Som bruger **skal** jeg gerne kunne se mine tilføjede venner.
- **Slet ven:** Som bruger **skal** jeg gerne kunne slette en ven jeg har tilføjet, så kommunikation ikke længere er mulig.
- **Rediger profil billede:** Som bruger **vil** jeg gerne kunne ændre mit profilbillede, så min profil bliver mere personlig.

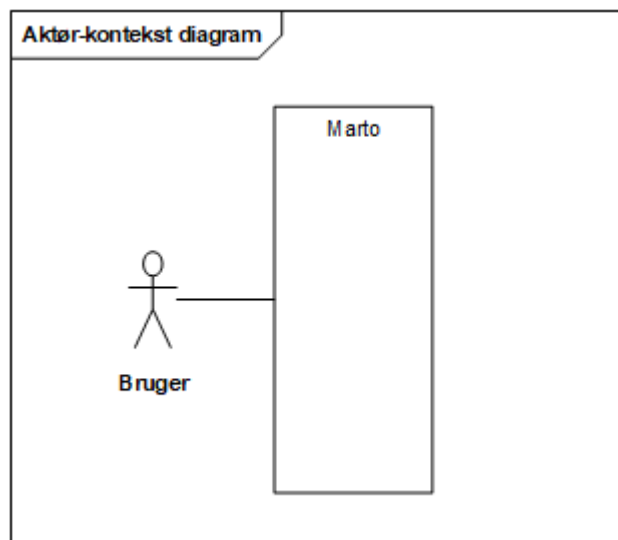
- **Rediger profil tag:** Som bruger **skal** jeg kunne tilføje og ændre min profils tags, så andre brugere kan finde min profil.
- **Rediger profil beskrivelse:** Som bruger **vil** jeg kunne tilføje og ændre min profil beskrivelse, så min profil bliver mere personlig.
- **Rediger profilkoden:** Som bruger **vil** jeg kunne med rimelig sikkerhed ændre min kode til min egen profil.

2.3.3 DecisionMaker

- **Tilfældighed:** Som bruger **skal** jeg kunne bruge rouletten til at tilfældigt træffe en beslutning ud af de brugerangivne muligheder.
- **Format:** Som bruger **skal** jeg kunne indtaste mine muligheder i tekstformat.
- **Layout:** Som bruger **vil** jeg kunne se rouletten med alle mulighederne, hvoraf den valgte tydeligt fremgår.

2.4 Aktør beskrivelse

Systemet har én primær aktør, som er brugeren. Brugeren skal kunne have adgang til Marto vha. en brugergrænseflade, som skal køre på en Computer. På figur 1 ses aktør kontekst diagrammet for Marto systemet. Alle moduler som systemet er opbygget af, indgår i Marto blokken. Der er derfor ingen sekundære aktører.

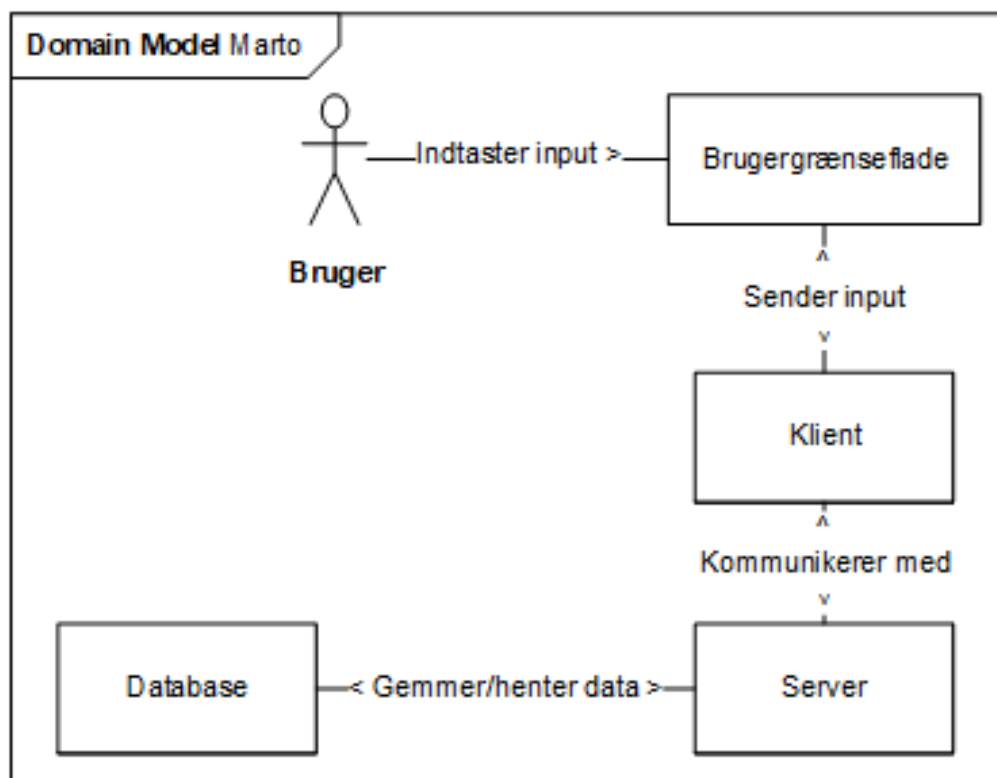


Figur 1: Aktør kontekst diagram for Marto

3 Arkitektur

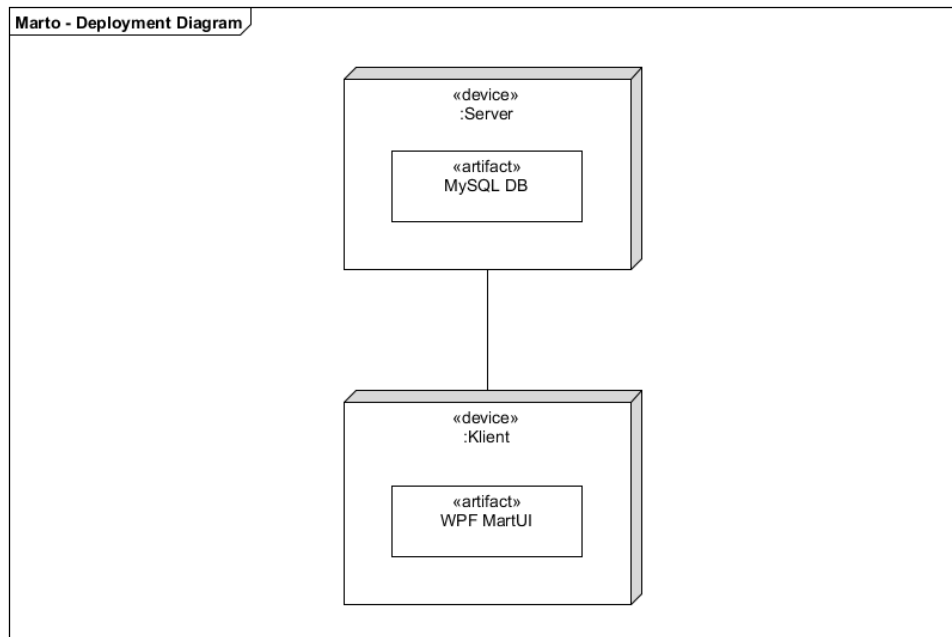
I dette afsnit bliver arkitekturen for systemet beskrevet. Arkitekturen har til formål at give et grundlag for udviklingen af systemet. Det giver et godt indblik i, hvordan det overordnede system ser ud, samt hvilke moduler der bruges for at opbygge systemet i en sammenhæng.

For at få et godt overblik over hele systemet, bliver der lavet en domænemodel. På figur 2 ses et domænemodel for det samlede projekt. Aktøren her er brugeren, og brugeren har her adgang til brugergrænsefladen, som sender/modtager input til/fra klienten. Klienten kommunikerer her sammen med serveren, som herefter går ind i databasen, for f.eks. at finde ud af om nogle bestemte oplysninger er gemt i databasen.



Figur 2: Domænemodel for Marto

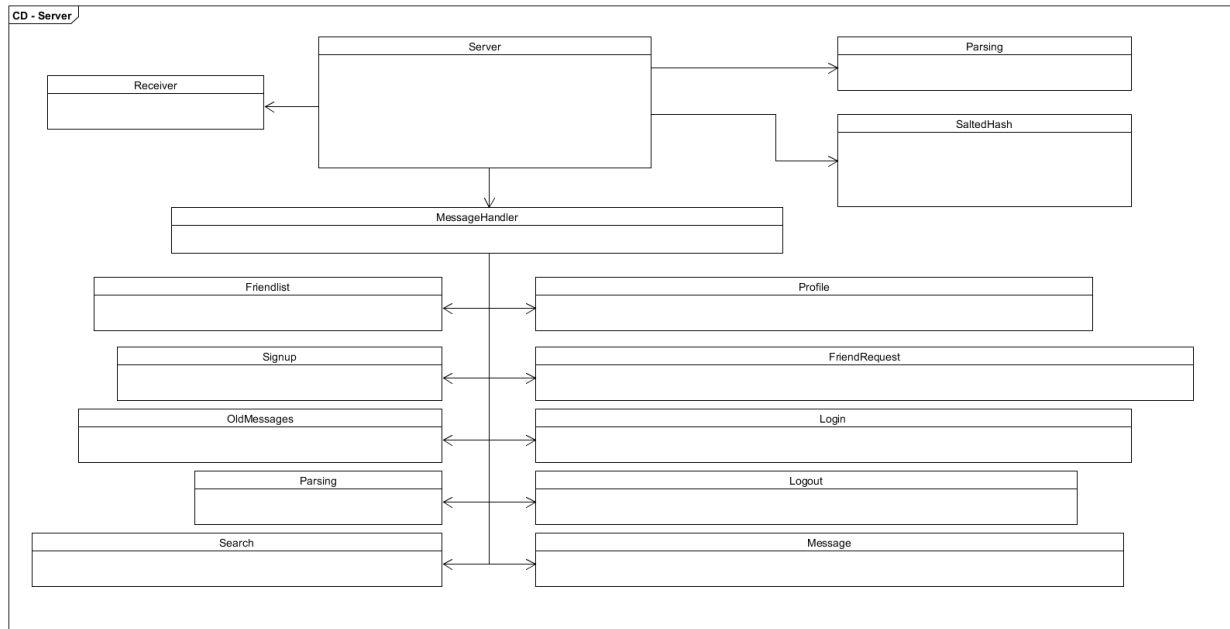
På figur 3 ses et deployment diagram over Marto systemet. Det ses her at Marto består af en server og en klient. Klienten har den grafiske brugergrænseflade, MartUI og serveren har MySQL databasen.



Figur 3: Deployment Diagram

3.1 Internet kommunikation

Til kommunikation frem og tilbage mellem klient og server, skal der bruges en række klasser, til at håndtere de forskellige events der kan forekomme fra klient til server.

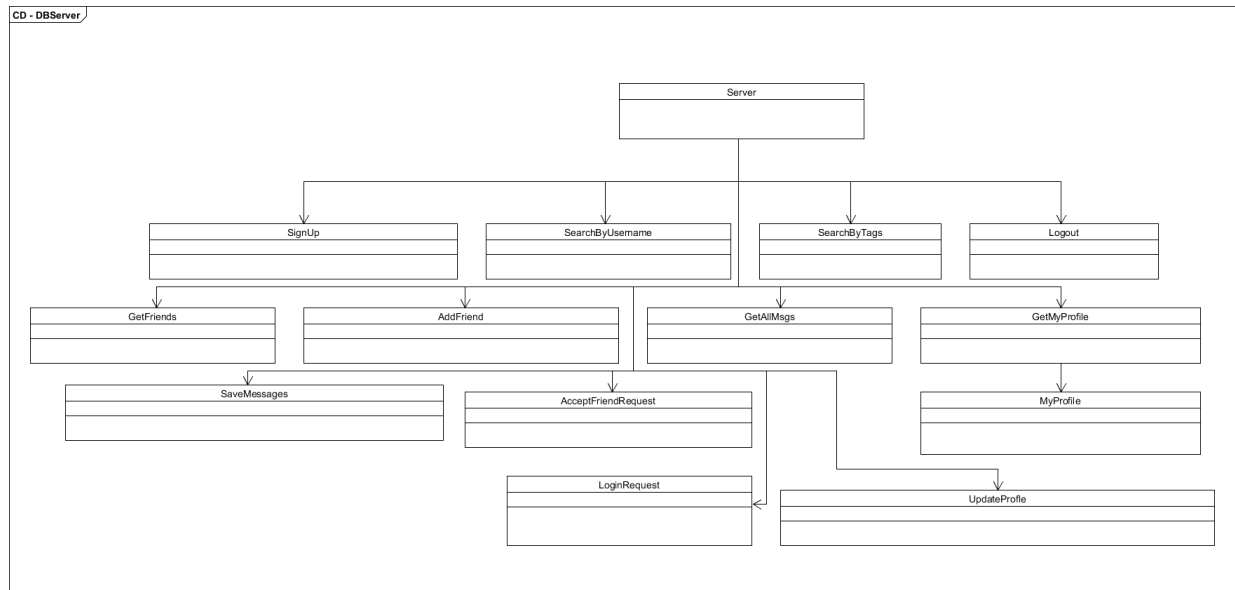


Figur 4: Klasse diagram over server

Øverst har vi klassen server, som vil være den primære klasse. Denne klasse sørger for at oprette serveren med det vores SSL-certifikat, samt håndtere nye klient forbindelser der tilsluttes serveren. MessageHandler klassen, skal håndtere alle de typer requests og data transfers, der forekommer mellem klient og server. Klassen sender er undladet i klassediagrammet af overskueligheds årsager. Sender klassen bliver brugt af Friendlist, Friendrequest, Login, Signup, Search, Profile, Oldmessages, Messages, Messagehandler.

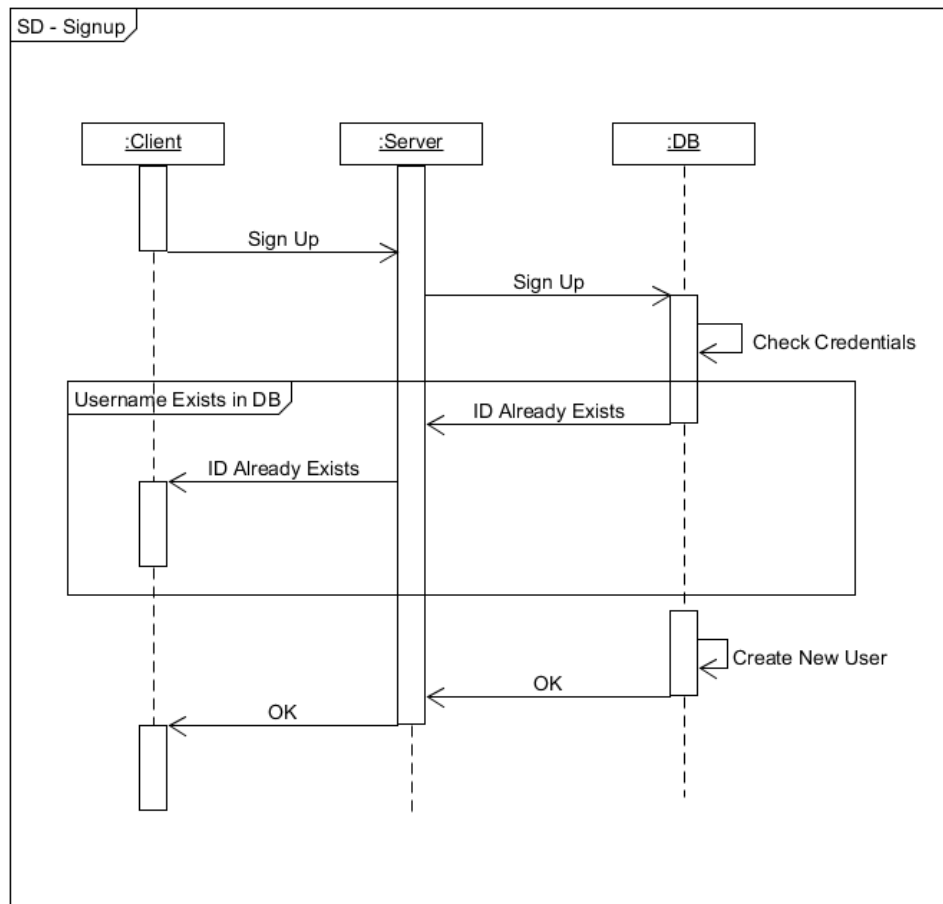
3.2 Kommunikation mellem server og database

For at gemme det data vi har opstillet i vores krav, skal der implementeres en række klasser til håndtering af dette. Besked handleren på serveren, benytter sig af disse klasser til at gemme og hente relevant data, på baggrund af hvilket request der modtages fra klienten. På figur 5 ses der et klassediagram over de klasser, serveren bruger til at kommunikere med databasen. Server klassen står tom for overskuelighedens skyld, da den består af mange klasser der vil gøre klassediagrammet for stort. Der refereres her i stedet til klassediagrammet i internet kommunikations afsnittet.



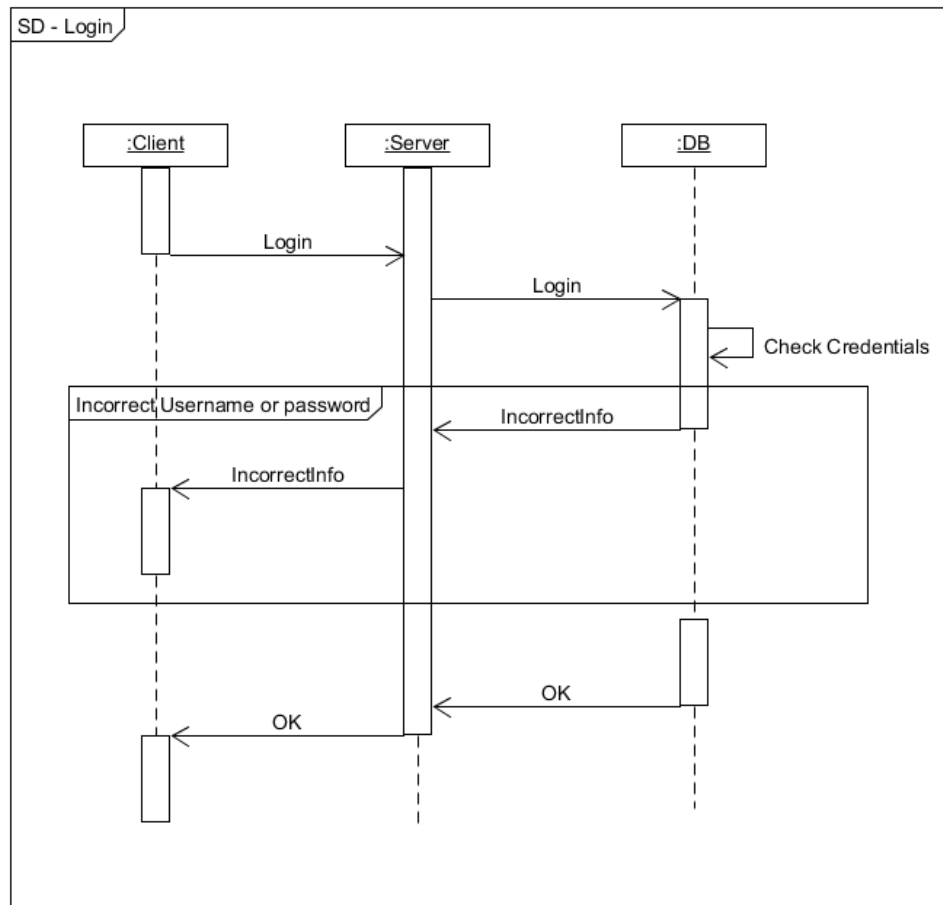
Figur 5: Klassediagram over DB kommunikationen fra server

På figur 6 ses hvordan klassen SignUp skal fungerer i sammenhæng med resten af systemet. Klienten indtaster de ønskede login information, brugernavn og kodeord. Serveren kigger i databasen om det modtagne brugernavn allerede eksisterer i databasen. Hvis det gør, modtager brugeren en fejlmeddelse, der siger at brugernavnet allerede eksisterer. Hvis brugernavnet ikke er optaget, vil brugeren blive oprettet i databasen og brugeren er nu klar til at logge ind.



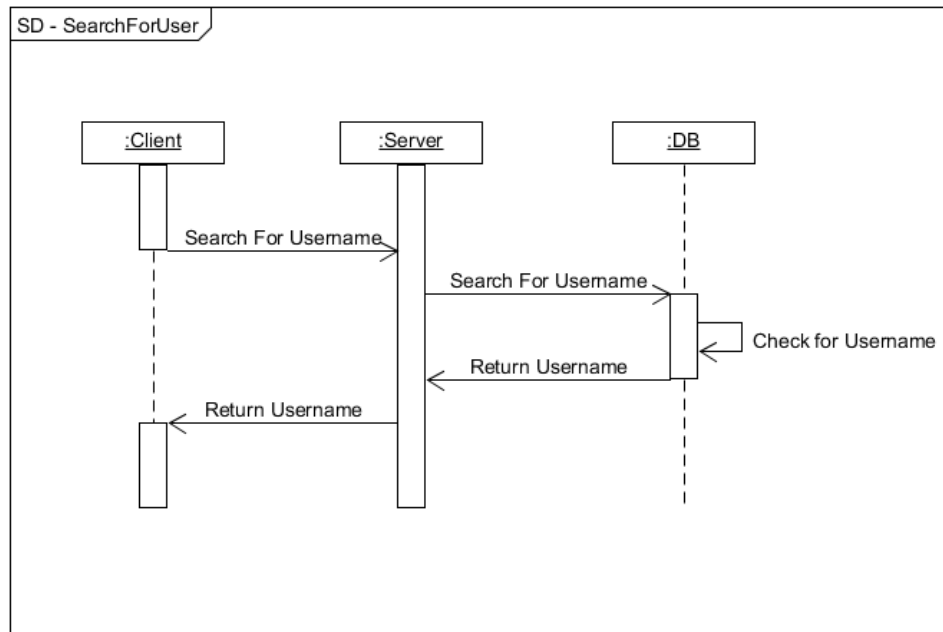
Figur 6: Sekvensdiagram over oprettelse af bruger

På figur 7 ses et sekvensdiagram over login processen. Den minder en del om signup sekvensen. Bruger indtaster sine login informationer og serveren tjekker i databasen om disse information stemmer overens, med en bruger i databasen. Hvis de ikke gør meldes der fejl til brugeren og brugeren må prøve igen. Stemmer informationerne overens, logges brugeren ind på systemet med de informationer.



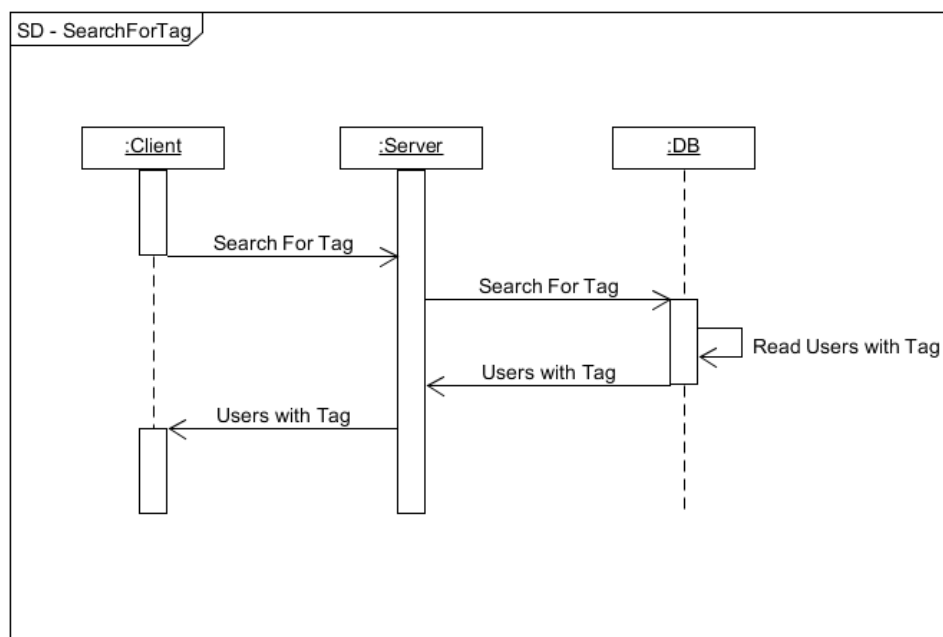
Figur 7: Sekvensdiagram over login

På sekvensdiagrammet over SearchForUser på figur 8 klassen, ses det at en given bruger indtaster navnet på en anden bruger. Hvis brugernavnet ikke eksisterer i systemet, vil brugeren ikke blive viderestillet til en anden profil. Hvis brugeren derimod eksisterer i databasen, vil den oprindelige bruger blive viderestillet til den anden brugers profil.



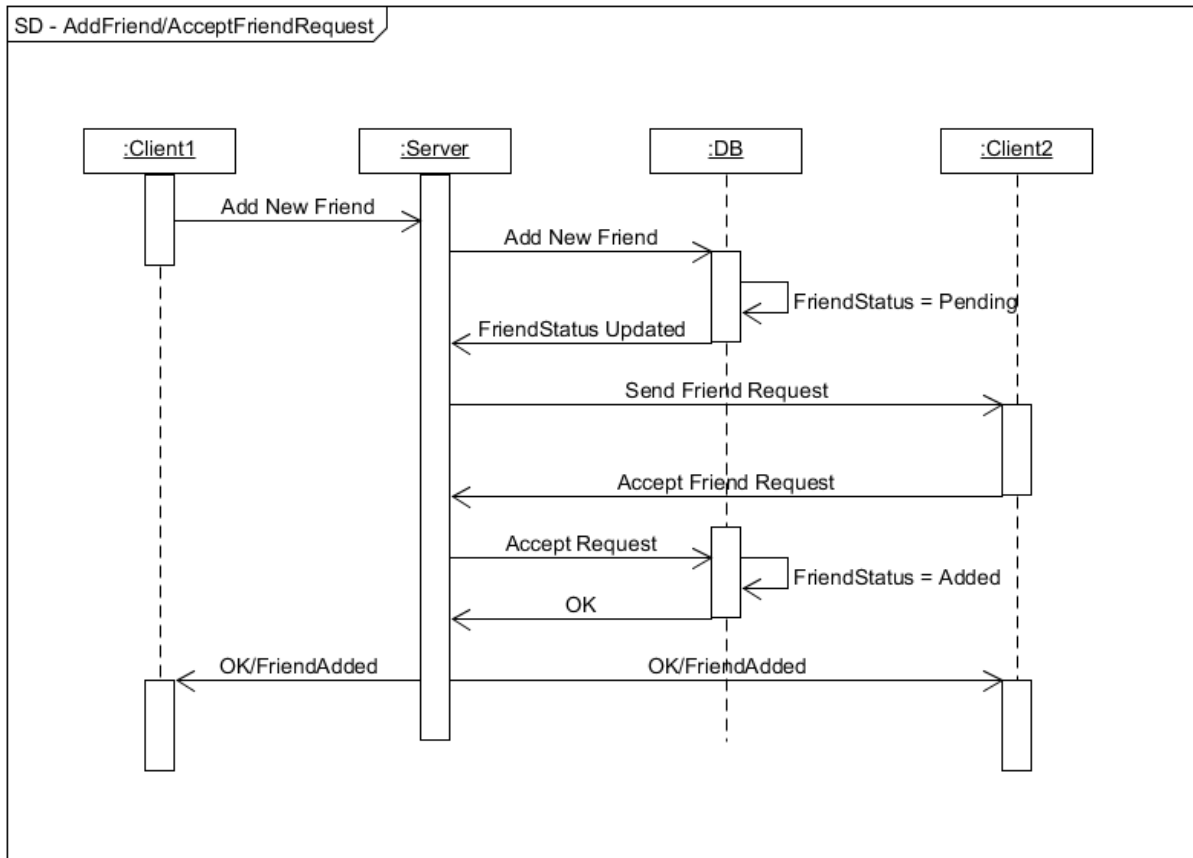
Figur 8: Sekvensdiagram over search for user

På nedenstående figur 9, kan man se search for tag funktionaliteten. Her søger brugeren på en tag frem for et brugernavn og får returneret alle brugere, med det relevante tag.



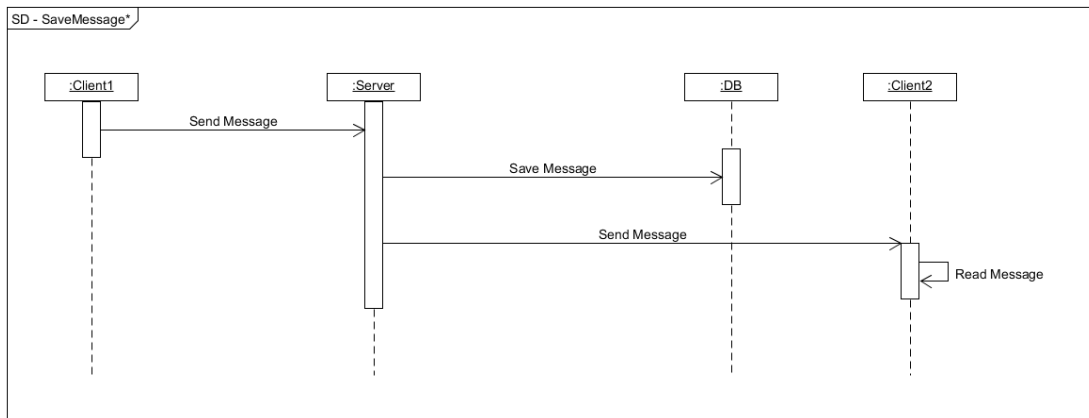
Figur 9: Sekvensdiagram over search for tag

På figur 10 ses sekvensdiagrammet over AddFriend og AcceptFriendRequest. Klient 1 anmoder en anden bruger, klient 2, om at blive venner. Dette skriver serveren i databasen og venskabsstatussen ændres til pending. Herefter sender serveren en venneanmodning til klient 2 om at der modtaget en venneanmodning. Brugeren accepterer her anmodningen og dette gemmes i databasen. Her opdateres venskabsstatussen til Added. Begge klienter modtager herefter at de nu er venner.



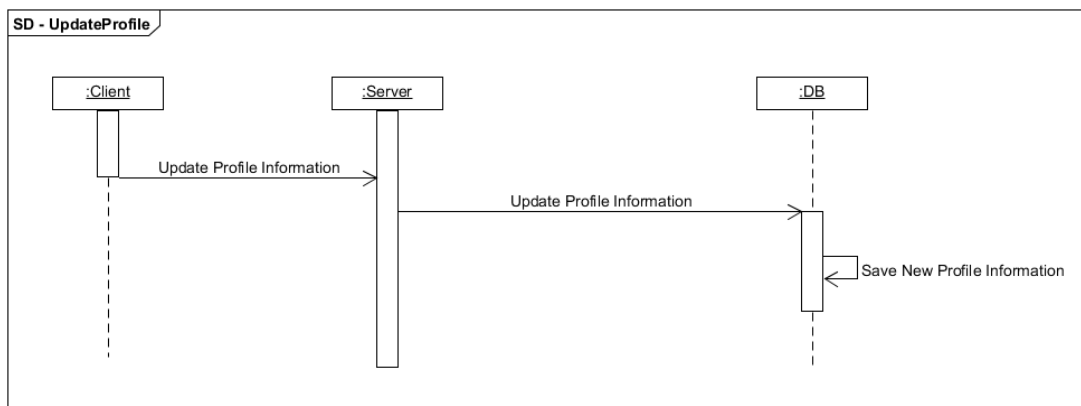
Figur 10: Sekvensdiagram addfriend og acceptfriendrequest

SaveMessage klassen bruges når der bliver skrevet til en besked fra en klient til en anden. Når klient 1 skriver til klient 2, vil beskeden først blive gemt i databasen, hvorefter den bliver sendt videre til klient 2.



Figur 11: Sekvensdiagram over save message

UpdateProfile kaldes når en bruger ændre i sit bruger information. Bruger information er defineret som brugerens profil beskrivelse eller tags. Serveren går ind og finder brugeren i databasen og overskriver de gamle informationer med de nye.



Figur 12: Sekvensdiagram over updateProfile

4 Design, implementering & test

4.1 MartUI

MartUI er den grafiske brugerflade på Marto. Det er herfra brugerne oprettes, logger ind, kommunikerer og meget mere. Selve applikationen er skrevet i C# WPF og er designet med MVVM (Model View ViewModel). Analyse, implementering og test af MartUI er beskrevet i nedenstående kapitler.

4.1.1 Analyse

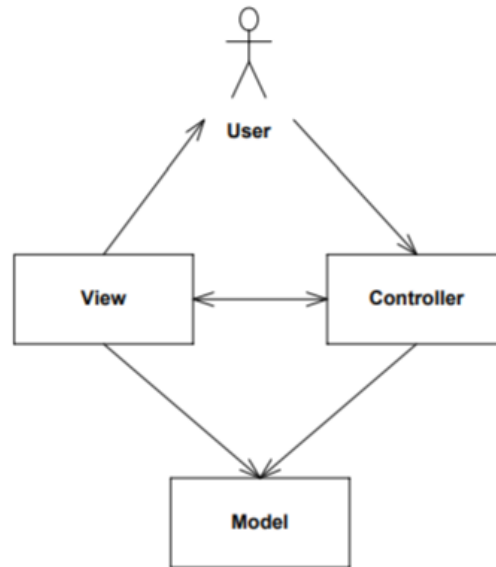
Kilde til analyseafsnittet er [4]. For at fremme forståelsen af hvordan MartUI skulle implementeres, blev visse ting analyseret inden implementeringen startede. Der er her især lagt vægt på at vælge et fornuftigt design pattern. Et godt design pattern er nødvendigt for at fremme videreudvikling af systemet, samt forbedre testbarheden. Følgende afsnit analyserer forskellige design patterns.

4.1.1.1 Indledning

Et af kravene til projektet er, at der skal udarbejdes en brugergrænseflade. Brugergrænsefladen udarbejdes i WPF. En WPF-applikation kan implementeres ved at definere arrangementen af controllers på skærmen, hvortil der kan bindes events til forskellige controllers. Når brugeren ændrer en værdi vil det kalde et event, som vil blive behandlet i code-behind. På denne måde interagerer brugeren med systemet. Dette er godt til små og simple applikationer, men der er en del problemer ved større applikationer. Man kan ikke genanvende forretningslogikken og det er ikke muligt at teste med automatiske test. Et andet meget stort problem er, at når grænsefladen vokser bliver den meget svær at overskue og det er nemt at foretage fejl. Måden hvorpå man kan rette op på dette problem er at bruge et design pattern. Relevante design pattern er MVP, MVC og MVVM. Disse patterns tager udgangspunkt i at opdele brugergrænsefladen i forskellige dele som styrer henholdsvis direkte brugerinput, forretningslogik og linket herimellem.

4.1.1.2 MVC (Model-View-Controller)

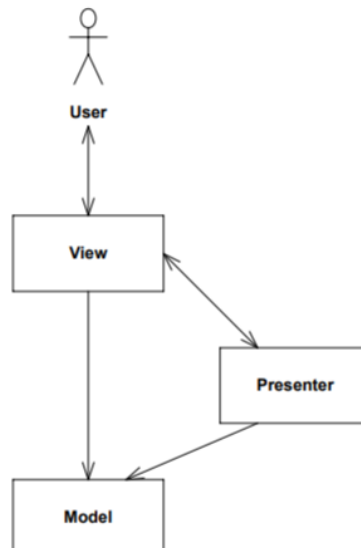
Dette design pattern består af Model, View og Controller, som der ses på figur 13. Brugeren ser View, som præsenterer modellen til brugeren. Brugeren kan bruge Controller til at interagere med systemet. Disse to komponenter er interfacet som brugeren ser og anvender. Model beskriver et objekt som indeholder data. Modellen er ikke afhængig af noget. Den indeholder logikken for at opdatere View hvis data ændres. Controlleren kan manipulere data i modellen. Ved at anvende observer-pattern, kan en værdi ændres i modellen som vil afspejles i View. Der er på denne måde ingen direkte interaktion mellem Model og View. Controller og View kan komme til at tale direkte med hinanden hvis man gerne vil ændre farven af en widget, skifte skærm eller andre View-relaterede ting som ikke har noget med modellen at gøre.



Figur 13: Diagram over MVC design pattern [4]

4.1.1.3 MVP (Model-View-Presenter)

I moderne GUI frameworks er det View som håndterer initial interaktion men delegerer med det samme til controlleren. View i denne model er en struktur af controllers som ikke har nogen opførsel. Den aktive reaktion til bruger input befinder sig i Presenter objektet. Et diagram over MVP ses på figur 14.



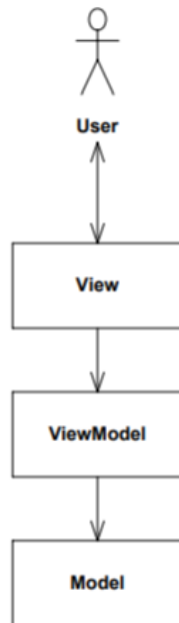
Figur 14: Diagram over MVP design pattern [4]

Her er der to måder at implementere modellen på: Passiv og Supervising view. I Supervising View

håndterer View simpel mapping til modellen og controlleren håndterer input og kompleks view logik. Dette kan gøres gennem data binding men er sværere at teste pga flere afhængigheder. I Passive View har View ingen direkte associering med modellen. Dette kan kræve meget kode som blot skal passere data fra modellen til view. Presenter kan laves om til en Presentation Model, hvor View observerer på denne, så View selv skal opdatere sig selv. Dette minimerer koblingen fra presenter til view.

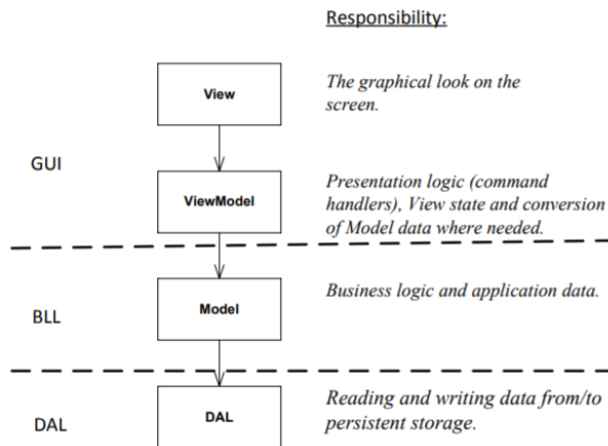
4.1.1.4 MVVM (Model-View-ViewModel)

MVVM er en specialiseret udgave af Presentation Model Pattern som anvendes i WPF. MVVM kan anvende specifikke funktioner, især data binding, i WPF for at bedre kunne opdele strukturen. Ved at anvende MVVM kan man fjerne al code-behind fra View laget. Dette betyder at designeren kan skrive View laget i XAML og ikke behøve at skrive noget C# som skrives af applikationudviklere. View forbindes til View-Model gennem data binding og sender kommandoer til View-Model, men View-Model kender ikke til View. View-Model interagerer med Model gennem properties og metodekald og kan modtage events fra Model. Modellen kender ikke til View-Model. En illustration af modellen ses på figur Dette betyder at der er meget lav kobling og alle lagene er godt separeret. Dette er specielt nyttigt at anvende i WPF, idét det anvender data binding og kommandoer. View-laget skal blot kende til en view-model gennem DataContext.



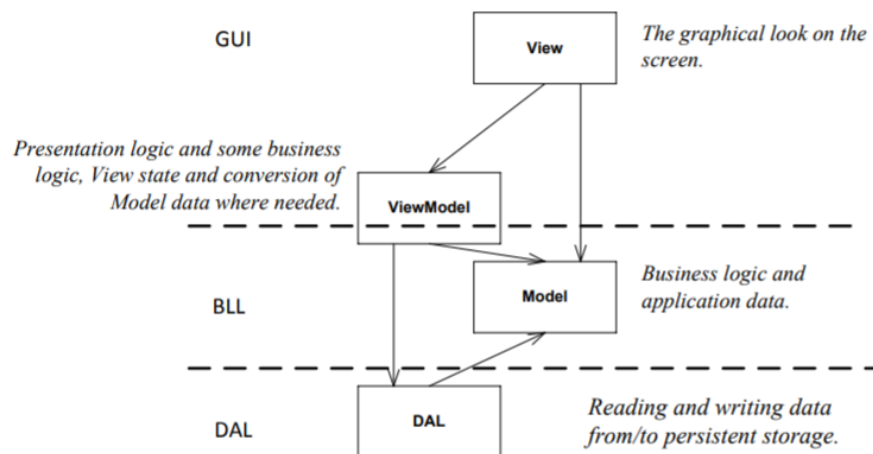
Figur 15: Diagram over MVVM design pattern [4]

Man vil opdele MVVM i flere lag, hvor selve GUI defineres i view og viewmodellen, og business logik findes i modellen. Det sidste lag og nederste lag er database laget. Dette illustreres på figur 16.



Figur 16: Diagram over det ideelle MVVM design pattern [4]

Da det er svært altid at opretholde MVVM design pattern, bliver de forskellige lag ofte blandet sammen. Viewmodel kommer til at blive en del af BLL og view kender til modellen. I MartUI er der forsøgt at følge MVVM, men ender også med at komme til at ligne denne model.



Figur 17: Diagram over det typiske MVVM design pattern [4]

4.1.1.5 Konklusion af design valg

Ulempen ved design patterns er at det kræver meget mere kode at skrive i forhold til en simple applikation som ikke bruger et design pattern. Dog opvejer design patterns dette idét at de har den egenskab at opdele lagene og forøge læsbarheden og vedligeholdelse. MVVM er specielt designet til at anvendes i WPF, og derfor også det oplagte valg i forhold til de øvrige kandidater.

Da MVVM er valgt som design pattern, har det efterfølgende været nødvendigt, at finde en måde hvorpå der kan kommunikeres på tværs af Views og ViewModels i WPF. Her er events trådt i spil. Alt dette blive der gravet dybere i i nedenstående afsnit.

4.1.1.6 Klient valg

Der er mange måder at lave en klient på, den kan være baseret på en terminal eller GUI. Det vigtigste for en klient er at den laver den såkaldte korrekte Client stream som kunne være enten UDP eller TCP.

4.1.1.7 Klient valg uddybbet

Valget blev TCPClienten og senere bliver omformatteres til en SSLstream.

I starten blev et terminalprogram lavet, som koden bærer præg af. Senere hen med større overvejelser om at dette terminal program kunne blive til en class library .dll som kunne bruges af GUI i form af et slags library, til at blive integreret ind i GUI som så ikke kunne blive laves om.

GUIs kompleksitet gjorde at det ville være nemmere at bruge dele af terminal koden og lavet det om til en del af GUI i form af events.

Derfor under nøje overvejning blev klienten lavet til en midstation der håndtere beskederne der kommer fra serveren og GUI.

4.1.2 Essentiel kode

I dette afsnit bliver essentiel kode forklaret. Dette betyder termer som bliver nævnt og anvendt igennem dokumentationen af MartUI bliver forklaret. Centrale principper som der er blevet brugt i implementeringen af MartUI bliver også forklaret i dette afsnit.

4.1.3 Views

'Views' er det interface som brugeren kan interagere med. Denne kode er primært skrevet i XAML med en smule ekstra kode skrevet i C# som ikke har plads i en viewmodel. Dette kan være kode som håndterer borders, dvs. hvis brugeren dobbeltklikker på kanten af et vindue eller lignende. Der er lavet et view for hver type af "skærm", som brugeren kan se. Dvs. dette kan være login-skærmen eller profil-skærmen. En referering til "login view" vil altså være det interface som brugeren kan se og interagere med.

4.1.4 Kommandoer (commands)

[6] Kommandoer (herefter commands) er en måde som man kan forbinde knapper og andre elementer til viewmodel. Viewmodel kan implementere en funktionalitet, fx. "log ind", som en bruger af applikationen kan initiere i viewet. Når brugeren vælger at trykke på en knap, vil knappen udløse en command i viewmodellen. Dette er en del af at afkoble viewmodel og view, da view blot skal forbinde sig til en command, og har ikke brug for at vide hvilken implementering der ligger bag. Anvender man simple events fra XAML til C# som WPF bygger på, vil det give en større kobling. Systemet skal holde øje med event handlers, så de kan ske når eventet bliver kaldt. Et andet problem med events er, at et event skal defineres i code-behind filen af en XAML fil, hvilket man helst gerne vil undgå ved at bruge MVVM.

4.1.5 Events

[7] Det er relativt nemt at forbinde en viewmodel til et view og få en lav kobling ved at anvende data binding. Et problem opstår dog hvis man vil have to forskellige viewmodels til at kommunikere med hinanden, da der netop er en lav kobling, så de kender ikke til hinanden.

Der er flere måder at gøre det på. Man kan anvende Mediator Pattern, som betyder at alle viewmodels kan sende til en samlet station, som vil kende til viewmodels. Mediatoren kan derefter distribuere beskederne rundt til viewmodels og på denne måde lade viewmodels kommunikere med hinanden. Der er forskellige frameworks som anvender dette. En anden måde er at bruge events. Dette betyder at man kan definere et event og lade en viewmodel "subscribe" på disse events. For at sende information fra en viewmodel til en anden kan en viewmodel "publish" et event som en viewmodel subscriber på. Dette kræver ikke meget kode, især hvis man vælger at bruge Microsofts' PRISM's EventAggregator klasse, som håndterer events på en elegant måde. Hvis flere views skal have et event, er dette også nemt implementeret; der skal blot subscribes på dette event fra den givne viewmodel.

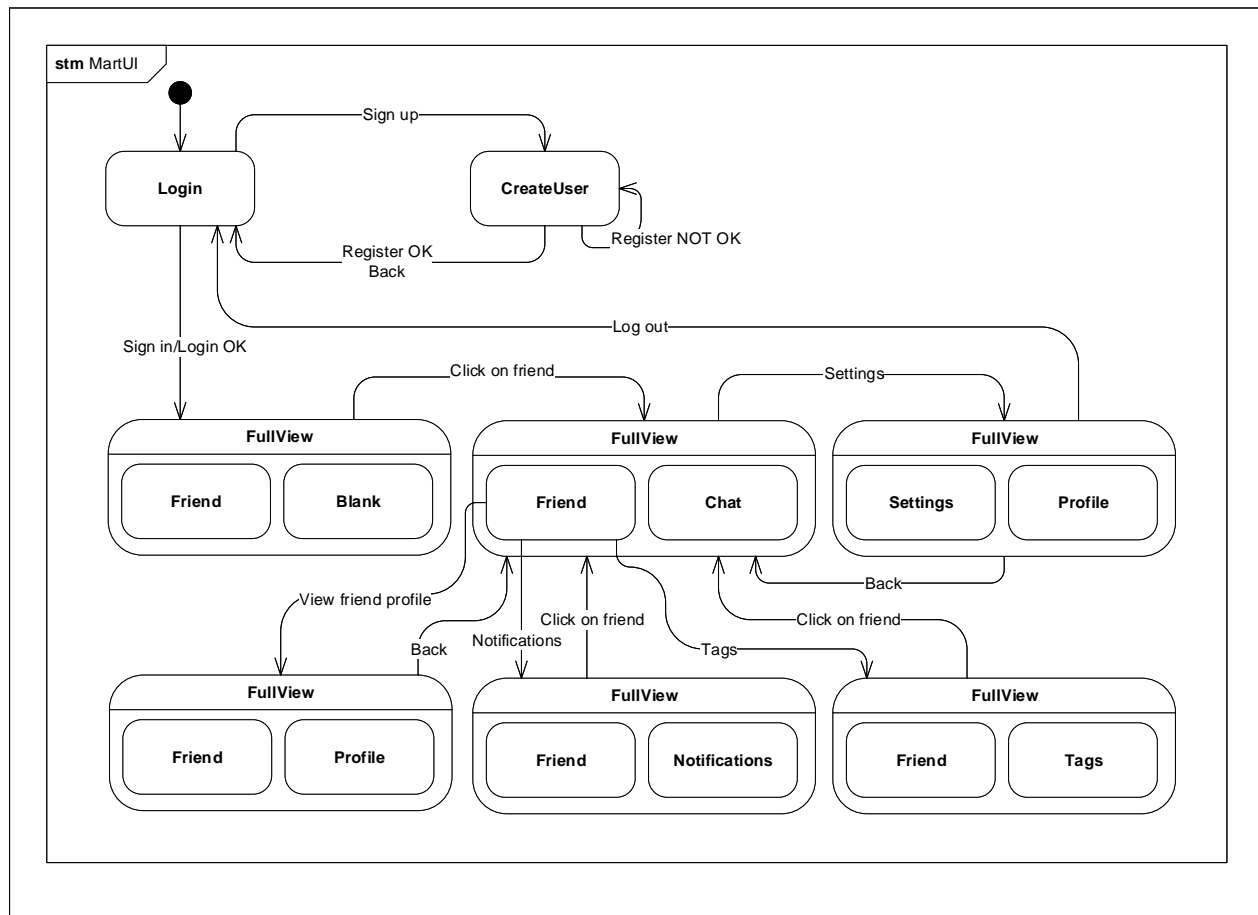
Både mediator og events kan anvendes. Mediator kræver at en samlet station håndterer events, mens events er frit tilgængelige. Da events ser mere håndterbare ud og ikke kræver en samlet station, er der valgt at arbejde videre med dette.

4.1.6 Design og implementering

MartUI er opbygget således at, brugeren nemt kan navigere rundt uden at læse lange manualer. Det er noget der generelt er vigtigt for brugergrænseflader, især som konkurrencen stiger, og kun de bedste bliver brugt. År brugeren anvender Marto, skulle de derfor gerne opleve at alt fra oprettelse af en bruger og tilføje andre brugere som ven, til at kommunikere med dem bør være ret intuitivt.

4.1.6.1 Overordnet design

Da der er anvendt MVVM, er koden opdelt i views og viewmodels. Dette betyder at der laves en klasse for hver view (hvor klassen er en viewmodel). Det er ikke overskueligt at lave et klassediagram over dette, men en tilstandsmaskine som beskriver, hvordan man navigerer mellem views er vist på figur 18.



Figur 18: Tilstandsmaskine over MartUI

Når brugeren tilgår MartUI, bliver brugeren ført til login. Herefter kan brugeren vælge at trykke på Sign up eller Sign in. Trykker brugeren Sign in, kan brugeren vælge at oprette en bruger ved at trykke Register. Serveren vil her validere om brugeren kan registreres. Hvis brugeren registreres korrekt, bliver der navigeret tilbage til Login hvor brugeren kan logge ind. Efter login kan brugeren se to views som er Friend og Blank. En tilstand some indeholder flere tilstande er her angivet som "Fullview". Dette betyder at man er inde i to tilstande, her Friend og Blank. Friend er vennelisten mens Blank er en tom start-side ved siden af vennelisten. Brugeren kan her vælge at trykke på en ven hvormed chatten åbnes for vennen. Dette betyder at der vises nu to views: Friend og Chat for en ven. Man kan trykke på Settings knappen som navigerer brugeren til Profile, som er brugeren profil, samt Settings, hvor brugeren kan logge ud, slette sin profil eller skifte sit kodeord. De sidste to elementer er ikke implementeret. Settings kan dog altid tilgås fra ethvert view, da det er en knap som altid kan ses på samme måde som man altid kan trykke på "x" i højre hjørne. Brugeren kan gå tilbage. Inde fra friend kan man højreklikke på en ven og se vennens profil. Man kan også tilgå sine notifikationer eller søge efter tags ved at klikke på henholdsvis på notifikation eller tags knappen.

De forskellige dele af MartUI er beskrevet i følgende afsnit, hvor implementering og design af dette

er beskrevet.

4.1.6.2 MVVM (PRISM)

[8] [9] Som tidligere nævnt er der valgt at anvende MVVM. Det er muligt at implementere MartUI uden et framework, men da der er forskellige værktøjer til rådighed, vil det give mere mening at bruge allerede udviklede værktøjer. Microsoft's PRISM er blevet anvendt i en grad. Event Aggregator anvendes til at oprette og publish/subscribe events, og PRISM's "SetProperty()" metode. Dette er det samme som "INotifyPropertyChanged", som notificerer klienter om at en property værdi har ændret sig. Dette er nødvendigt for at få det tilsvarende view til at opdatere. Følgende kodeudsnit viser et eksempel på de to former.

SetProperty:

```
SetProperty(ref _myData, value);
```

INotifyPropertyChanged:

```
if (_myData != value)
{
    myData = value;
    NotifyPropertyChanged();
}
```

NotifyPropertyChanged kræver ekstra linjer kode, da der ikke ønskes at notificere observers, hvis værdien ikke har ændret sig. Herefter sættes værdien og der notificeres. Dette er indkapslet i PRISM's SetProperty(), som selv håndterer at tjekke om værdien har ændret sig og herefter notificere samt sætte værdien.

Dette er nødvendigt at bruge når man arbejder med views - de er nødt til at vide når værdier ændrer sig i viewmodel. Det er desuden muligt at bruge ekstra værktøjer fra PRISM, bl.a. at man ikke behøver at definere datacontexten af en view/viewmodel i et MainView, men blot at definere dem ved startup. Dette er dog svært at debugge, da mange af metoderne er komplekse og kræver meget dokumentation for at forstå. Derfor er der valgt at lave en blanding af almindelig MVVM uden frameworks sammen med PRISM.

```
public class PubSubEvent<TPayload> : EventBase
```

Der er forsøgt at følge MVVM så godt som muligt, men til tider er dette svært at implementere helt korrekt i specielle situationer, og er derfor blevet forbigået. Det er heller ikke et krav at følge MVVM 100% men det øger kvaliteten af koden.

4.1.6.2.1 Events

PRISM's events er blevet anvendt. PubSubEvent, som er defineret i EventBase anvendes. Eventet er defineret som:

```
public class PubSubEvent<TPayload> : EventBase
```

For at anvende dette event skal man blot definere et event og lade den arve fra PubSubEvent med en mulig payload. Dette kan fx være:

```
public class GetFriendList : PubSubEvent<string>
{ }
```

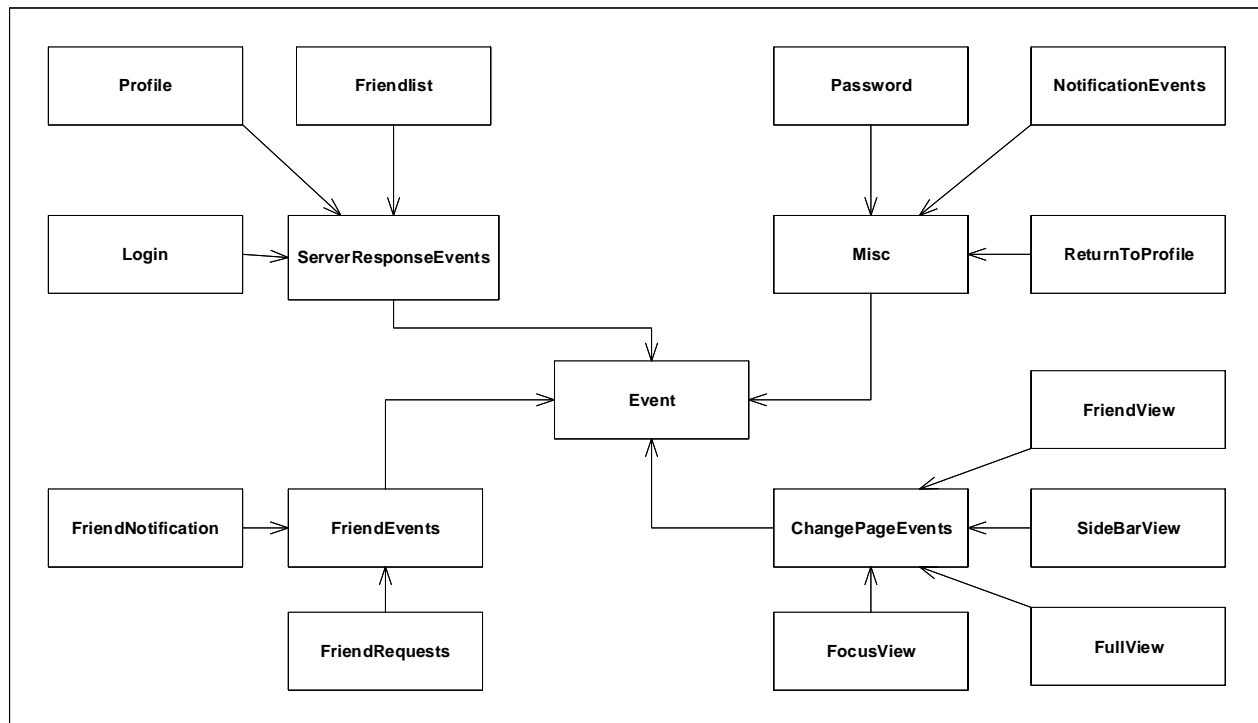
Dette vil være et simpelt event som der kan subscribes og publishes ved hjælp af en EventAggregator. Er en eventaggregator defineret i en viewmodel, kan eventet publishes som:

```
_eventAggregator.GetEvent<GetFriendList>.Publish(friends);
```

En anden viewmodel kan blot subscribe på dette med en Subscribe metode i stedet for Publish, hvor parameteren er en funktion som der delegeres til.

Ulempen ved denne måde at implementere MartUI er, at der kommer til at være mange events, da der skal være et event (en klasse) ved hver informationsdeling eller view-skift, som kan blive uoverskueligt i længden.

Da der er mange events i MartUI, er det nødvendigt med et overordnet diagram. Figur 19 viser et overblik over nogle af de events som der er oprettet.



Figur 19: Overblik over diverse events der bruges i MartUI

Der er brugt events til at håndtere Friends hvormed der er lavet FriendNotification events og FriendRequest events. Under ServerResponseEvent håndteres events som bliver sendt, når der modtages respons fra serveren. Misc er forskellige events som bruges i specielle situationer, bl.a. håndtering af password og notificationevents. Password forklares i et senere afsnit. ChangePageEvents anvendes til at navigere rundt i systemet, og er de events man oftest vil anvende.

4.1.6.2.2 Commands

En command defineres ved ICommand interfacet. Her kan man bruge to metoder, CanExecute() og Execute(), hvilket betyder at en betingelse kan være opfyldt før man kan få lov til at eksekvere en kode. Dvs. man kan deaktivere en knap eller en anden kontrol hvis betingelsen ikke er opfyldt.

PRISM's DelegateCommand er anvendt, som implementerer ICommand interfacet. En simpel deklaration af en Command som anvender DelegateCommand kunne være:

```
ICommand ChangePageToFriends = new DelegateCommand(ChangeToFriends);
```

ChangeToFriends er en simpel metode. Dette kan også være en anonym metode defineret med et lambda expression. En command og et event vil tit være koblet sammen hvis der skal skiftes view:

```
ICommand ChangePageToFriends = new DelegateCommand(() =>
    eventAggregator.GetEvent<GetFriendList>.Publish(friends));
```

Dette eksempel på en command er et af byggestenene i MartUI.

4.1.6.3 MyData

MyData-klassen er klassen som mange viewmodels vil tilgå. Denne klasse er lavet statisk: klassen i sig selv er ikke statisk, men en instans af MyData inde i MyData er statisk, samt er der en statisk `GetInstance()` som returnerer instansen:

```
private static MyData _instance;
// Return an instance (singleton!)
public static MyData GetInstance()
{
    return _instance ?? (_instance = new MyData());
}
```

Herefter vil klassen ligne en normal model med et brugernavn, kodeord, tags, beskrivelse og et billede, som der ses på kodeudsnittet:

```
private Uri _image;
public Uri Image
{
    get => _image ?? (_image = new
Uri("pack://application:,,,/Images/ProfilePicPlaceholder.PNG"));
    set => SetProperty(ref _image, value);
}

private string _description;
public string Description
{
    get => _description ?? (_description = "Describe yourself!");
    set => SetProperty(ref _description, value);
}
```

Skal andre klasser bruge denne klasse, skal de blot kalde `GetMyInstance()`.

4.1.6.4 Main

Alle viewmodels implementerer interfacet `IViewModel`:

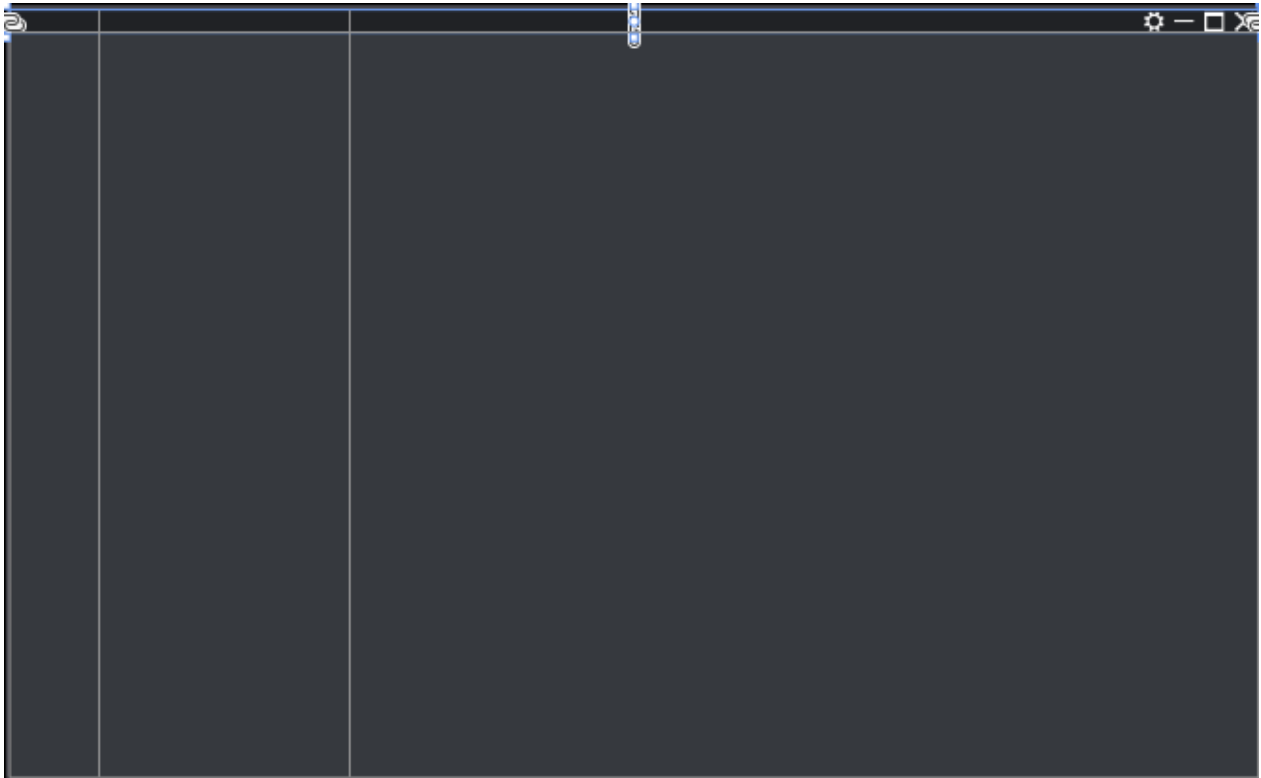
```
public interface IViewModel
{
    string ReferenceName { get; }
}
```

Dette gør det muligt at indsætte dem i en liste og pille dem ud når de skal bruges.

Main-klassen er den klasse, som der bliver startet op. Herinde bliver alle de forskellige views indsat. En DataContext bliver sat for hver view/viewmodel par, så Main ved, hvordan de hænger sammen. Et udsnit af dette ses nedenfor:

```
<Window.Resources>
  <DataTemplate DataType="{x:Type friend:FriendViewModel}">
    <friend:FriendView DataContext="{Binding}" />
  </DataTemplate>
  <DataTemplate DataType="{x:Type login:LoginViewModel}">
    <login:LoginView DataContext="{Binding}" />
  </DataTemplate>
  <DataTemplate DataType="{x:Type createuser:CreateUserViewModel}">
    <createuser:CreateUserView DataContext="{Binding}" />
  </DataTemplate>
  ...
  ...
</Window.Resources>
```

De 4 knapper i hjørnet (settings, minimize, maximize, close) bliver også defineret herinde. Det mere interessant omkring Main er dog hvordan den er opstillet. Der er lavet fire forskellige "zoner" på Main. Disse zoner er opdelt som der ses på 20. Der er en bar til venstre, som står for at håndtere gruppechat. Baren til højre for denne bar kommer til at håndtere vennerne og chatten med dem. Den sidste, og største firkant til højre kommer til at håndtere chatten, profil, og andre vigtige ting.



Figur 20: Opsætning af Main-vinduet

Måden hvorpå dette sættes inde i XAML er meget simpel. En ContentControl sættes, hvormed Content sættes lig med et sæt af properties inde i MainViewModel.

```
<Grid Grid.Column="0" Grid.ColumnSpan="3" Grid.Row="1" Width="Auto">  
    <ContentControl Content="{Binding FullView}" />  
</Grid>
```

```
<Grid Grid.Column="0" Grid.Row="1">  
    <ContentControl Content="{Binding SideBarView}" />  
</Grid>
```

```
<Grid Grid.Column="1" Grid.Row="1" >  
    <ContentControl Content="{Binding FriendListView}" />  
</Grid>
```

```
<Grid Grid.Column="2" Grid.Row="1" Width="Auto">  
    <ContentControl Content="{Binding FocusView}" />  
</Grid>
```

Inde i MainViewModel er de fire properties defineret. Dette er en IViewModel:

```
public IViewModel FocusView
{
    get => _focusView;
    set => SetProperty(ref _focusView, value);
}

public IViewModel FriendListView
{
    get => _friendListView;
    set => SetProperty(ref _friendListView, value);
}
...
...
```

En liste af IViewModel er defineret. Da alle viewmodels implementerer IViewModel, kan de indsættes i denne liste. I constructoren bliver nogle af viewmodels indsat i listen. Dette er nødvendigt da nogle af dem subscriber på events, som bliver sendt af fx. login eller create user. Her sker der også en subscription på de 4 events som håndterer skift af view, kaldet ChangeFullPage, ChangeFocuspage, ChangeFriendPage og ChangeSideBarPage. Det sidste event håndterer et blinkende ikon inde i FriendView. Listen sættes til at være lig med det første element som er lig med LoginViewModel. I de fire første events bliver der medsendt en viewmodel, som fortæller, hvilket view der skal skiftes til.

```
public MainViewModel()
{
    // Make one method that does these
    _eventAggregator.GetEvent<ChangeFullPage>().Subscribe(ChangeFullView);
    _eventAggregator.GetEvent<ChangeFocusPage>().Subscribe(ChangeFocusView);
    _eventAggregator.GetEvent<ChangeFriendPage>().Subscribe(ChangeFriendView);
    _eventAggregator.GetEvent<ChangeSideBarPage>().Subscribe(ChangeSideBarView);
    _eventAggregator.GetEvent<NotificationReceivedEvent>().Subscribe(Notify);

    ViewList.Add(new LoginViewModel());
    ViewList.Add(new FriendViewModel());
    ViewList.Add(new ChatViewModel());
    ViewList.Add(new ProfileViewModel());
    ViewList.Add(new FriendNotificationViewModel());
    ViewList.Add(new SettingsViewModel());

    FullView = ViewList[0];
}
```

Følgende kodeudsnit er metoderne som bliver kaldt. Her bliver modellerne sat.

```
private void ChangeFriendView(IViewModel model)
{
    FriendListView = GetTrueModel(model);
}

private void ChangeFocusView(IViewModel model)
{
    FocusView = GetTrueModel(model);
}
...
...
```

Dog, da der altid sendes en ny model med, skal der tjekkes om modellen findes i forvejen i listen:

```
private IViewModel GetTrueModel(IViewModel model)
{
    if (model == null) return null;

    bool isFound = false;

    foreach (var view in ViewList)
    {
        if (model.ReferenceName == view.ReferenceName)
        {
            isFound = true;
            model = view;
            break;
        }
    }
    if (!isFound)
        ViewList.Add(model);

    return model;
}
```

Dette kodeudsnit itererer gennem ViewList og tjekker referencenavnene. Bliver modellen fundet, sættes view lig modellen og den indsættes ikke i listen. Findes den ikke, indsættes den i listen. Herefter returneres den. Dette sikrer at der ikke indsættes ens viewmodels ind i listen, men det betyder at der oprettes mange modeller som ikke anvendes og skal samles af garbage collectoren.

Settings knappen styres inde fra denne viewmodel:

```
public void ShowSettings()
{
    _eventAggregator.GetEvent<ReturnToProfile>().Publish();
    _eventAggregator.GetEvent<ChangeFriendPage>()
        .Publish(new SettingsViewModel());
    _eventAggregator.GetEvent<ChangeFocusPage>()
        .Publish(new ProfileViewModel());
}
```

Dette skifter blot til Profile og Settings. ReturnToProfile er en måde at vise profilen for brugeren. Dette bliver yderligere forklaret i Profile.

4.1.6.5 Helpers

Dette afsnit beskriver ekstra klasser, som anvendes i projektet, men ikke er views eller viewmodels.

4.1.6.5.1 Bootstrapper

Denne klasse overrider CreateShell(), der returnerer en instans af default typen i containeren. Dvs. at MainView bliver returneret. InitializeShell() sætter den nuværende application til at vise MainWindow. Under app.xaml.cs instantieres denne klasse og der kaldes .Run. Dette sikrer sig at initieringen af services sker korrekt. Man kunne have indsat de forskellige views i containeren i stedet for i en liste af IViewModel, men dette er der valgt ikke at gøre pga. tidligere nævnte grunde.

4.1.6.5.2 DatabaseDummy

Denne klasse er kun brugt til at teste at CreateUser og Login fungerer uden en server, og har ikke yderligere relevans for projektet.

4.1.6.5.3 TokenizingControl

Denne kontrol konverterer tekst fra CreateUser's tag input til at blive til grønne knapper. Denne kode er inspireret af [10], og der vil ikke gå meget i dybden med denne. Den tjekker for hver indtastning om der er skrevet noget, og vil danne en token, hvis TokenMatcher bliver kaldt. TokenMatcher er sat i code-behind til at returnere en tekst, når der trykkes på mellemrum. Ved at identificere hvor dette sker, kan man indsætte et event som returnerer tagget, som en anden viewmodel kan subscribe på. Det samme kan gøres ved slet af tags. En "TagControl" klasse er defineret, som kan håndtere dette:

```
GetEventAggregator.Get().GetEvent<ChangingTagsInCreate>()
    .Publish(new TagControl(true, txt));
```

Sættes parameteren lig true skal der indsættes et nyt tag.

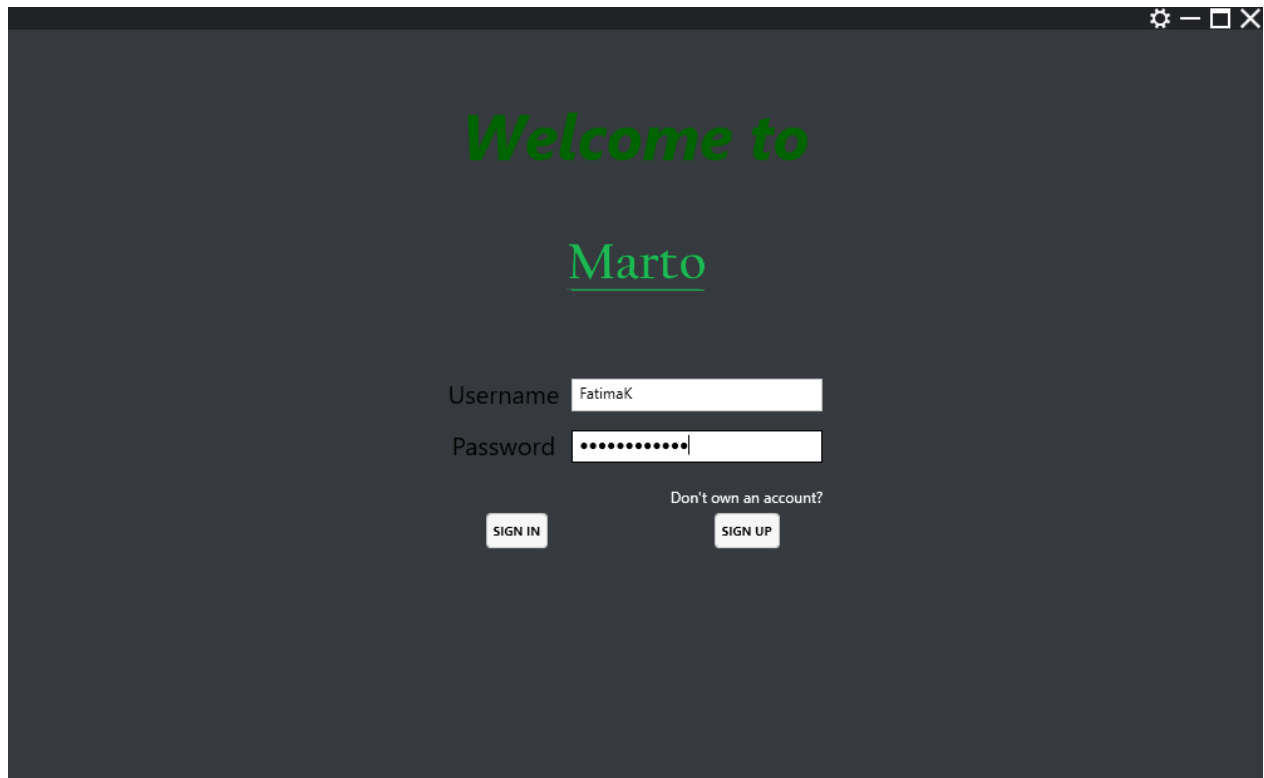
Det er muligt at tilføje en `ItemsSource` property, i stedet for at tilføje dette event. Det er ikke set som det vigtigste i projektet og er derfor nedprioriteret.

4.1.6.6 Ressourcer

Ressourcerne er alle billeder og ikoner som der anvendes. "Marto" billedet og ikonet er defineret under "Misc" og "Resources". Der anvendes en pakke af ikoner som er defineret under `App.xaml` under en liste af `ResourceDictionaries`.

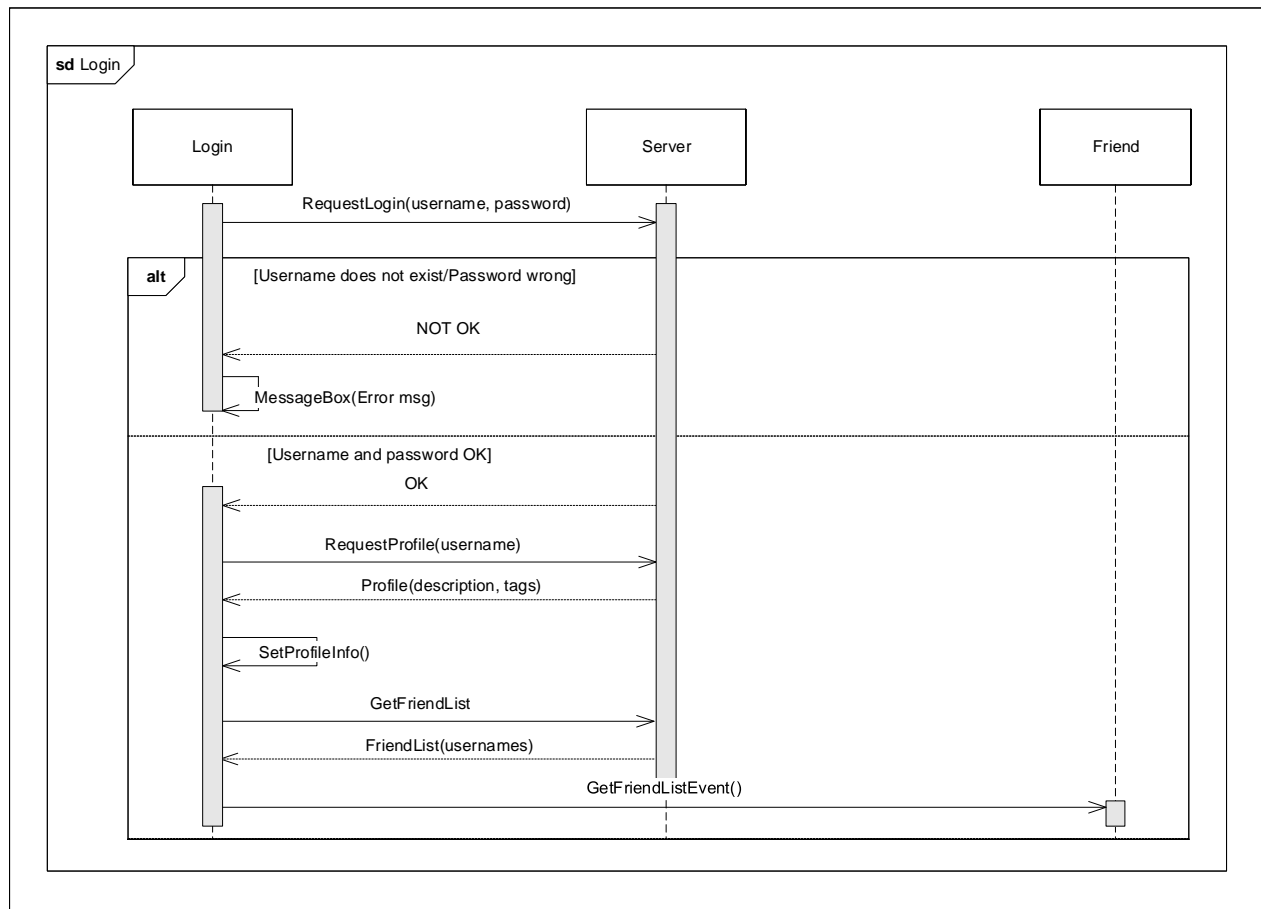
4.1.6.7 Login

På figur 21 ses et billede af login-skærmen af MartUI. Dette er en simpel skærm hvor brugeren kan indtaste sit brugernavn og kodeord i de to felter og trykke på "Sign in" for at logge ind. Er brugeren ikke oprettet i forvejen, kan der oprettes en bruger ved at trykke på "Sign up". På billedet er der indtastet "FatimaK" som brugernavn og et kodeord som ikke kan ses.



Figur 21: Billede af login i MartUI

På 22 ses sekvensdiagrammet over hvordan login fungerer. Brugeren indtaster sit brugernavn og kodeord og trykker på "Sign in" knappen. Dette sender en forespørgsel til serveren om at logge brugeren ind. Lykkedes dette bliver der modtaget profilinfo og venneinfo om brugeren fra serveren til klienten (som her er Login-blokken). For at få vennelisten opdateret, bliver listen sendt videre til "Friend"-skærmen som håndterer vennerne, samt får alle gamle beskeder når dette modtages.



Figur 22: Sekvensdiagram over login

På figur 23 er der ikke indtastet et brugernavn eller kodeord, som gør at knappen ikke kan trykkes på.



Figur 23: "Sign in" knappen kan ikke trykkes på

4.1.6.7.1 Implementering

For at kunne kommunikere med andre modeller, subscriber denne viewmodel på en del events:

```
public LoginViewModel()
{
    _eventAggregator = GetEventAggregator.Get();
    _eventAggregator.GetEvent<PasswordChangedInLogin>().Subscribe(paraPass =>
        Password = paraPass);
    _eventAggregator.GetEvent<LogoutPublishLoginEvent>()
        .Subscribe(HandleLogoutPublishLogin);
    _eventAggregator.GetEvent<LoginResponseEvent>().Subscribe(HandleLogin);
    _eventAggregator.GetEvent<GetProfile>().Subscribe(ProfileInfo);
    _eventAggregator.GetEvent<GetFriendList>().Subscribe(FriendListInfo);
}
```

Disse events i login bliver published af bl.a. code-behind filen og når der modtages data fra serveren.

Implementeringen af Login består i at lave et Grid i XAML og indsætte de forskellige elementer. Der bruges selvfølgelig bindings og commands til at styre kommunikationen til viewmodel. For at kunne observere på Username og Password er der ikke oprettet en underliggende model, som man normalt vil gøre med MVVM. Dette vil man normalt gøre hvis man vil tilgå en model, men i dette tilfælde kan PRISM's `.ObserveProperty()` ikke observerer komplekse properties: dvs. den kan ikke opdage hvis en property under en anden property er ændret. Username kan dog stadig sættes ved

at bruge MyData klassen. Her anvendes MyData klassen til at set/get MyData (UserData er sat lig instansen af MyData):

```
public string Username
{
    get => UserData.Username;
    set
    {
        if (UserData.Username == value) return;
        UserData.Username = value;
        RaisePropertyChanged();
    } // if username != value, notify
}
```

"Password" er lidt mere kompleks idét der er forsøgt at skjule kodeordet. Normalt vil man ikke gemme et kodeord i plain-tekst, men i dette tilfælde bliver det gemt midlertidigt på klienten. Dette gøres for at kunne få lov til at observere på denne property. Hvis man ville undgå at gemme kodeordet, ville det være nødvendigt at spørge serveren om, om et kodeord er indtastet og derefter få lov til at logge ind. For at formindske forespørgsler til serveren er dette ikke blevet gjort, men det er noget som kan gøres. Der forekommer dog stadig sikkerhedshåndtering af kodeordet når kodeordet sendes over serveren til databasen.

For at få lov til at observere på en PasswordBox (som der anvendes i XAML-filen), er det nødvendigt at anvende et event eller en anden form for videregivelse af kodeordet, da det ikke er muligt at binde til en PasswordBox af sikkerhedsmæssige årsager. Det er ikke et stort problem i dette tilfælde, da kodeordet bliver håndteret af serveren i denne prototype, men er slet ikke acceptabelt i et færdigt produkt. Dette betyder at et event er blevet defineret i code-behind filen, som sender kodeordet i et event hver gang den ændres af brugeren:

```
private void PassBx_OnPasswordChanged(object sender, RoutedEventArgs e)
{
    GetEventAggregator.Get().GetEvent<PasswordChangedInLogin>()
        .Publish(PassBx.Password);
}
```

LoginViewModel, som har en property, Password, kan derefter subscribe på dette event og sætte kodeordet:

```
_eventAggregator.GetEvent<PasswordChangedInLogin>().Subscribe(paraPass =>
    Password = paraPass);
```

Den command som håndterer "Login" knappen kan derefter observerer på de to properties:

```
public ICommand LoginCommand => _loginCommand ?? (_loginCommand =
```

```
new DelegateCommand(LoginExecute, LoginCanExecute)
.ObservesProperty(() => Username)
.ObservesProperty(() => Password));
```

LoginExecute() er en metode som tjekker om brugernavnet og kodeordet er sat, samt om det er over 1 karakter langt.

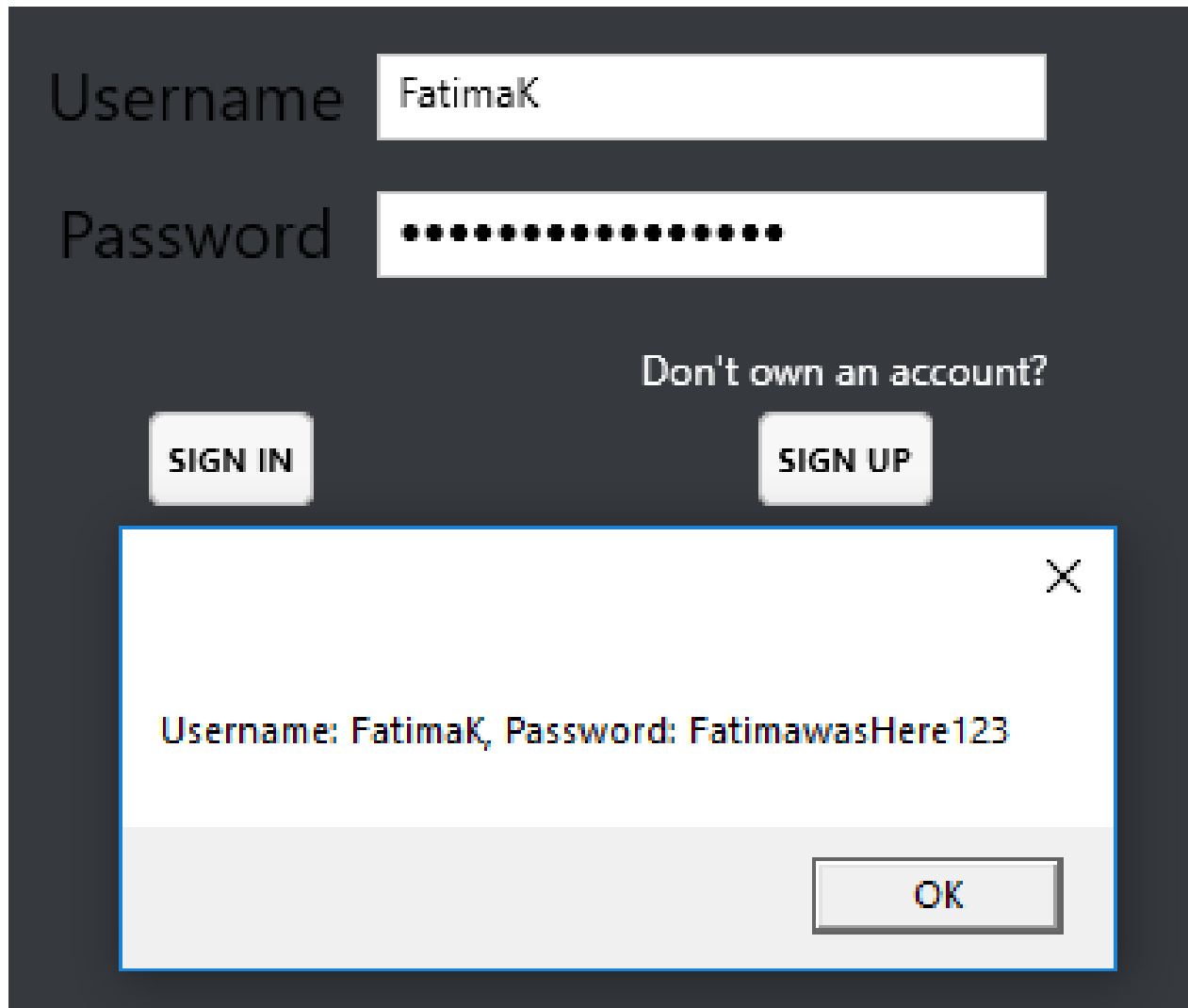
Der kommunikeres med serveren ved hjælp af et event, som der sendes, når klienten modtager profilen ved login. Følgende kodeudsnit anvendes til at håndtere respons:

```
private void HandleLogin(string response)
{
    switch (response)
    {
        case "OK":
            _eventAggregator.GetEvent<SendMessageToServerEvent> ()
            .Publish(Constants.RequestProfile
            + Constants.GroupDelimiter
            + Username);
            break;
        case "NOK":
            MessageBox.Show("Cannot login! Wrong username or password!");
            break;
    }
}
```

Når profilen modtages bliver strengen splittet op og håndteret, hvorefter vennelisten hentes og sendes videre.

4.1.6.7.2 Test

Der er blevet foretaget nogle få test til at tjekke om brugernavnet og kodeordet bliver sat. Dette kan nemt tjekkes ved at indsætte en MessageBox som viser de to informationer når der trykkes på "Sign in". Dette ses på figur 24.

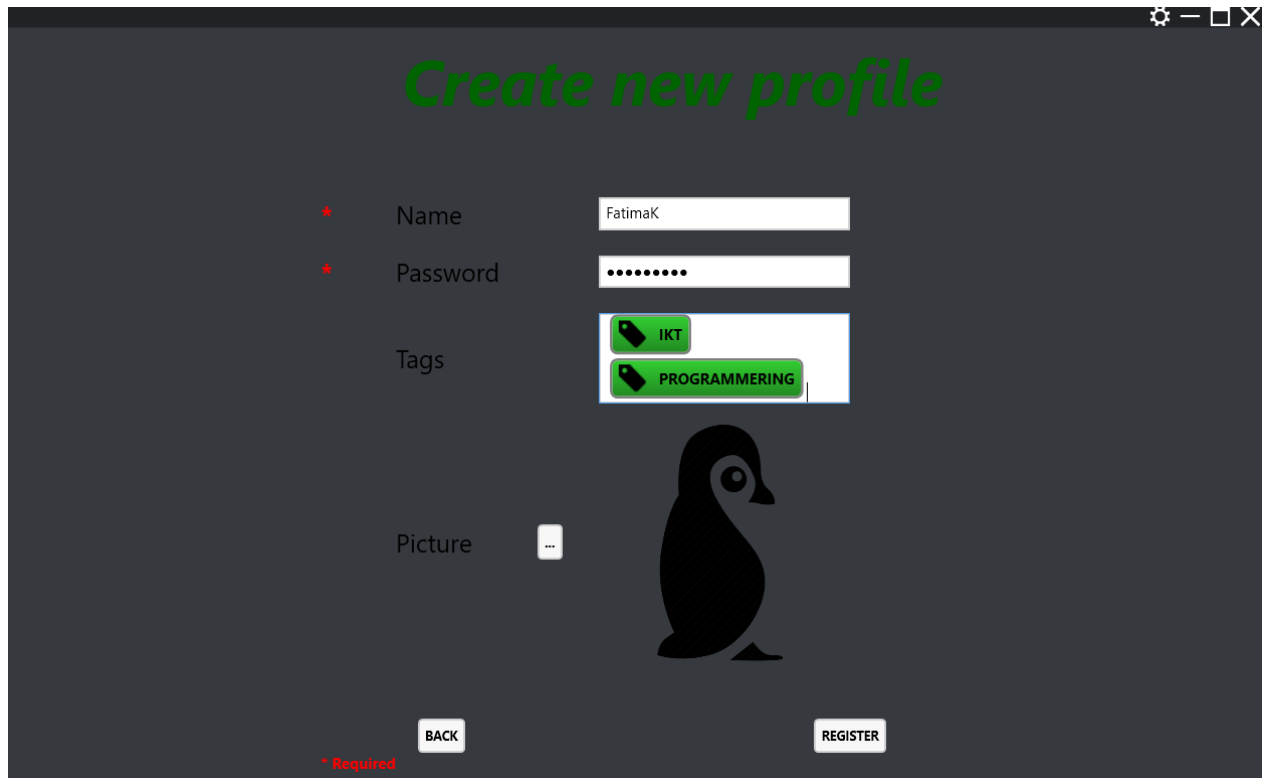


Figur 24: Test af login med en message box

4.1.6.8 Create User

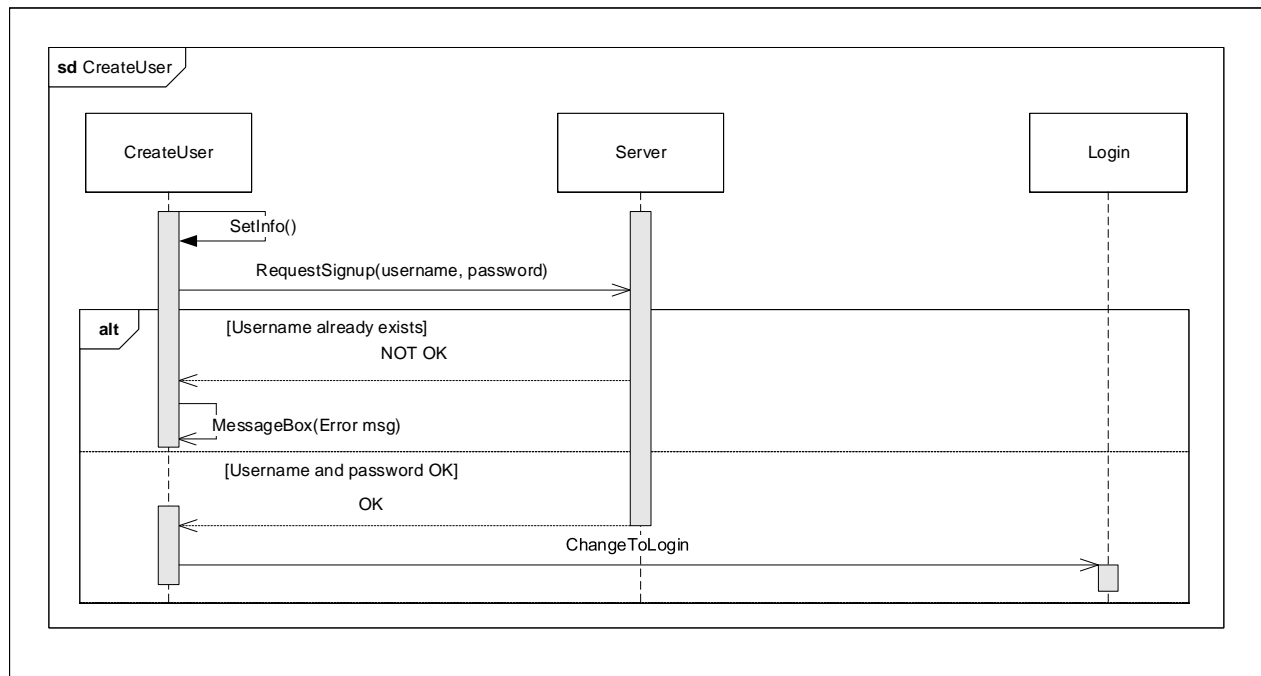
4.1.6.8.1 Design

På figur 25 ses layoutet af "Create User". Brugeren kan indtaste et brugernavn og et password. Passwordet kan ikke ses af brugeren og det er ikke muligt at kopiere. Disse er begge nødvendige for at oprette en bruger. Tags og billedet er optionelt. Tags indskrives ved at skrive et tag efterfulgt af et tryk på 'Enter' knappen. Dette vil vise tagget som en grøn firkant med tagget. Billedet kan browses ved at trykke på "..." som åbner en fildialog hvor brugeren kan vælge et billede. Det er muligt at gå tilbage til login-skærmen ved at trykke på "Back". For at oprette brugeren trykkes der på "Register". På figuren bliver brugernavnet sat til FatimaK, tags til "IKT" og "PROGRAMMERING" og billedet indsættes. Kodeordet vises som punkter.



Figur 25: Billede af opret bruger i MartUI

På figur 26 ses sekvensdiagrammet over oprettelse af en ny bruger. Brugeren indsætter et brugernavn og password. Serveren modtager i denne prototype et brugernavn og password, men kan udvides til at modtage tags og billedet. En forespørgsel om at oprette brugeren sendes til serveren. Der modtages et svar på om brugeren er oprettet, eller om der opstod en fejl. Der opstår en fejl hvis brugeren allerede eksisterer i databasen. Bliver brugeren korrekt oprettet skiftes der til login-skærmen hvor brugeren kan logge ind.



Figur 26: Sekvensdiagram over oprettelse af en ny bruger

På 27 ses at "Register"-knappen ikke kan trykkes på, når der ikke er indtastet et username og password. Dette giver noget ekstra brugervejledning.

The image shows a registration form on a dark background. It has four main input areas: 'Name' with a red asterisk, 'Password' with a red asterisk, 'Tags', and 'Picture'. The 'Picture' field includes a small icon with three dots and a larger placeholder image of a landscape. At the bottom, there are two buttons: 'BACK' and 'REGISTER'. The 'REGISTER' button is greyed out, indicating it is disabled. A red asterisk followed by the word 'Required' is located at the bottom left of the form.

Figur 27: "Register" kan ikke trykkes på

4.1.6.8.2 Implementering

Implementeringen af Create User minder lidt om login, idét den også skal tjekke om Name og Password er sat, hvorefter det er muligt at trykke på "Register". Det er også muligt at indsætte tags og vælge et billede, men dette sendes ikke med til serveren på denne prototype. For at gøre det muligt at kommunikere med serveren, subscribes der på nogle events:

```
public CreateUserViewModel()  
{  
    //Subscriptions  
    _eventAggregator.GetEvent<PasswordChangedInCreate>().Subscribe(para =>  
        Password = para);  
    _eventAggregator.GetEvent<SignupResponseEvent>()  
        .Subscribe(CheckSignup);  
    _eventAggregator.GetEvent<ChangingTagsInCreate>().Subscribe(ModifyTags);  
}
```

```
}
```

SignUpResponseEvent bliver sendt når der modtages noget relevant fra serveren. ChangingTagsInCreate sendes fra TokenizingControl, som håndterer de grønne tag-bokse. Dette betyder også, at det kun er muligt at fjerne det sidste tag, og ikke et vilkårligt tag:

```
public void ModifyTags(TagControl tag)
{
    if (!tag.Command)
    {
        if (UserData.Tags.Any())
            UserData.Tags.RemoveAt(UserData.Tags.Count - 1);
    }
    else
    {
        UserData.Tags.Add(tag.Tag);
    }
}
```

Det er også muligt at vælge et billede. Trykkes der på "..." popper der en dialog op som lader brugeren vælge et vilkårligt billede.

```
private void ChoosePicture()
{
    OpenFileDialog dialog = new OpenFileDialog
    {
        Title = "Select a picture",
        Filter = "Image files (*.png;*.jpeg)|*.png;*.jpeg|All files (*.*)|*.*"
    };

    if (dialog.ShowDialog() == true)
        UserData.Image = new Uri(dialog.FileName);
}
```

Når brugeren vælger at trykke på "Register" bliver der sendt en forespørgsel til serveren om at oprette brugeren:

```
string msg = Constants.Signup + Constants.GroupDelimiter +
            Username + Constants.GroupDelimiter +
            Password;

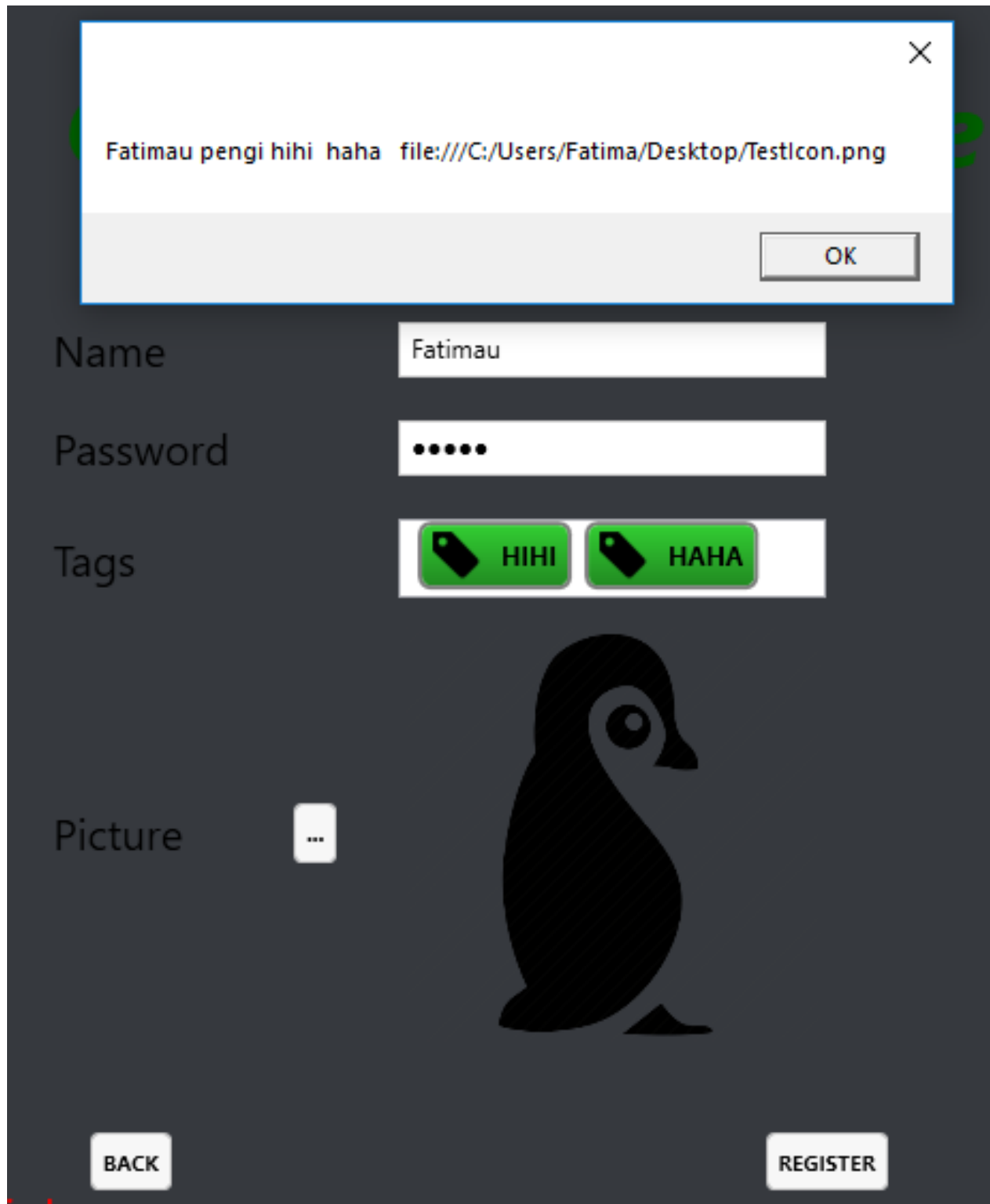
_eventAggregator.GetEvent<SendMessageToServerEvent>().Publish(msg);
```

Følgende kodeudsnit håndterer respons fra serveren. Der modtages enten "SOK" eller "SNOK" for "Sign up OK/Not OK". Lykkedes det at oprette brugern bliver brugeren navigeret til login-siden.

```
private void CheckSignup(string response)
{
    switch (response)
    {
        case "SOK":
            _eventAggregator.GetEvent<ChangeFullPage>().Publish(new
                LoginViewModel());
            MessageBox.Show($"Welcome to the club, {UserData.Username}! You can
                now log in");
            break;
        case "SNOK":
            MessageBox.Show("Username " + Username + " already exists! Choose
                something else");
            break;
    }
}
```

4.1.6.8.3 Test

En simpel test er foretaget ved at indsætte data og få en MessageBox til at udskrive informationen når der trykkes på "Register". Brugernavnet er Fatimau, adgangskode er pengi, de to tags er hihi og haha, mens filstien identificeres på billedet. Dette ses på figur 28.

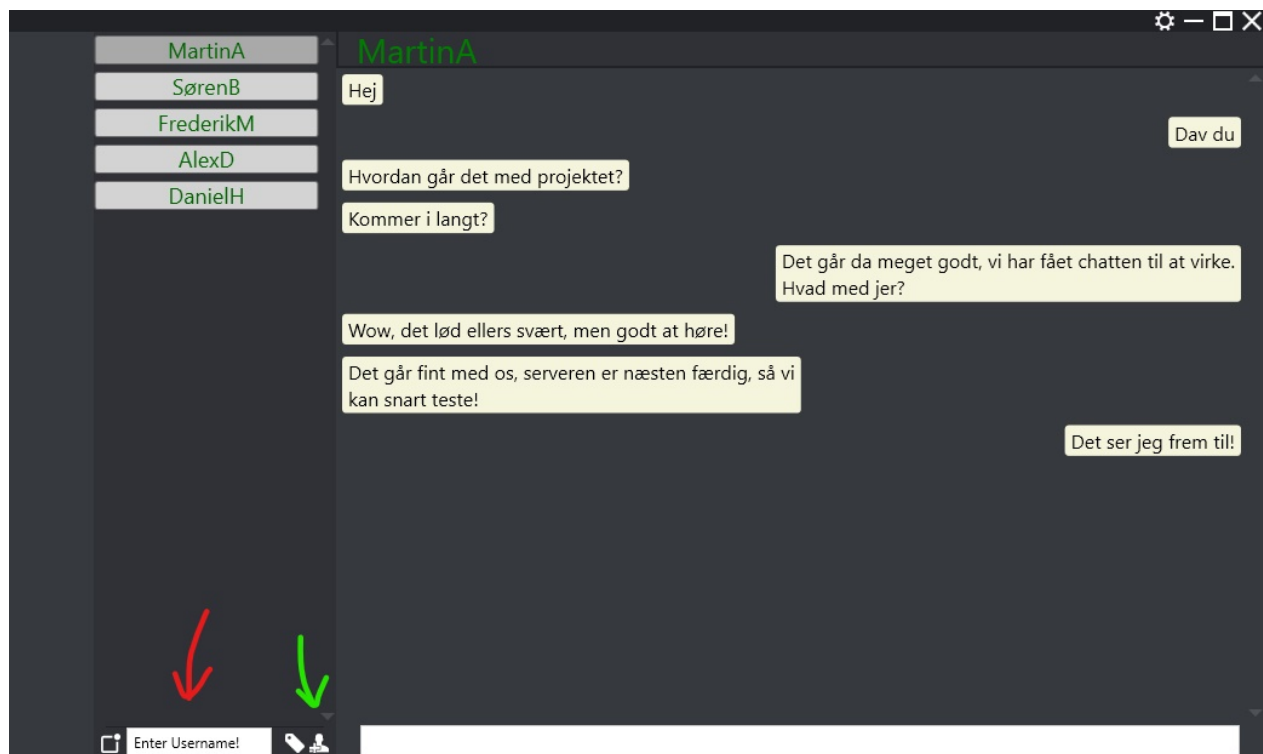


Figur 28: Test af Create User, som viser en MessageBox med det korrekte info indtastet

4.1.6.9 Friend

Det første brugeren ser, når de er logget ind, er en venneliste, som initielt er tom, men som hurtigt opfyldes ved hjælp af en "Add Friend" eller tilføj ven funktionalitet, som er implementeret ved hjælp af et klik på en knap som frembringer en tekstboks, hvori brugeren kan indtaste vennens navn og trykke enter, hvilket sender en venneanmodning. Når brugeren har tilføjet en, eller flere, ven(ner) kan de kommunikere med dem, ved at trykke på vennens navn i vennelisten, og skrive i den chat der fremstår på højre side. Det er også muligt at slette venner fra vennelisten, ved at højreklikke på dem og vælge "Remove" fra menuen. Til sidst kan man tilgå venners profil, ved at vælge "View Profile" fra samme drop down menu.

Figur 29 viser et billede af MartUI med knappen til at åbne tekstboksen til at tilføje venner, og selve tekstboksen, markeret. Der ses også at vennelisten ikke er tom, og at brugeren er i færd med at skrive til "MartinA".

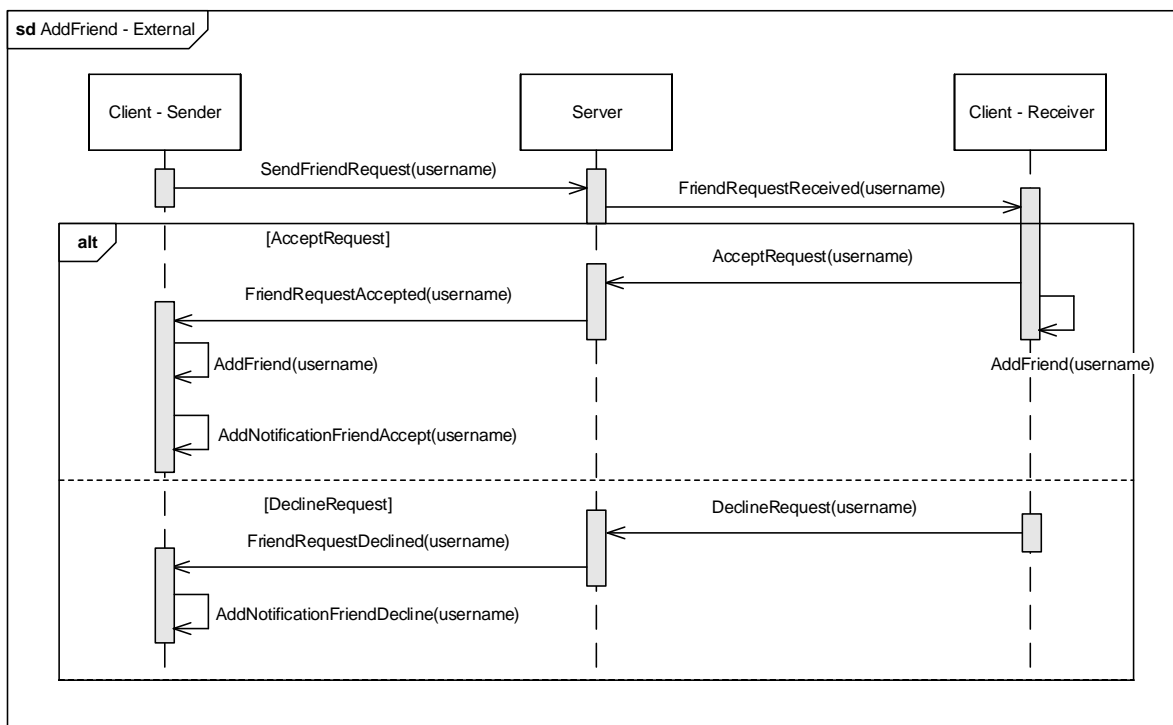


Figur 29: Billede af Chat, vennelist og tilføj ven i MartUI

Som det ses står der: "Enter Username!" i tekstboksen, indtil brugeren trykker på boksen, hvorpå teksten slettes, og brugeren kan indtaste det brugernavn de ønsker at ansøge. Det er først når brugeren trykke på enter, at ansøgningen vil blive sendt til serveren.

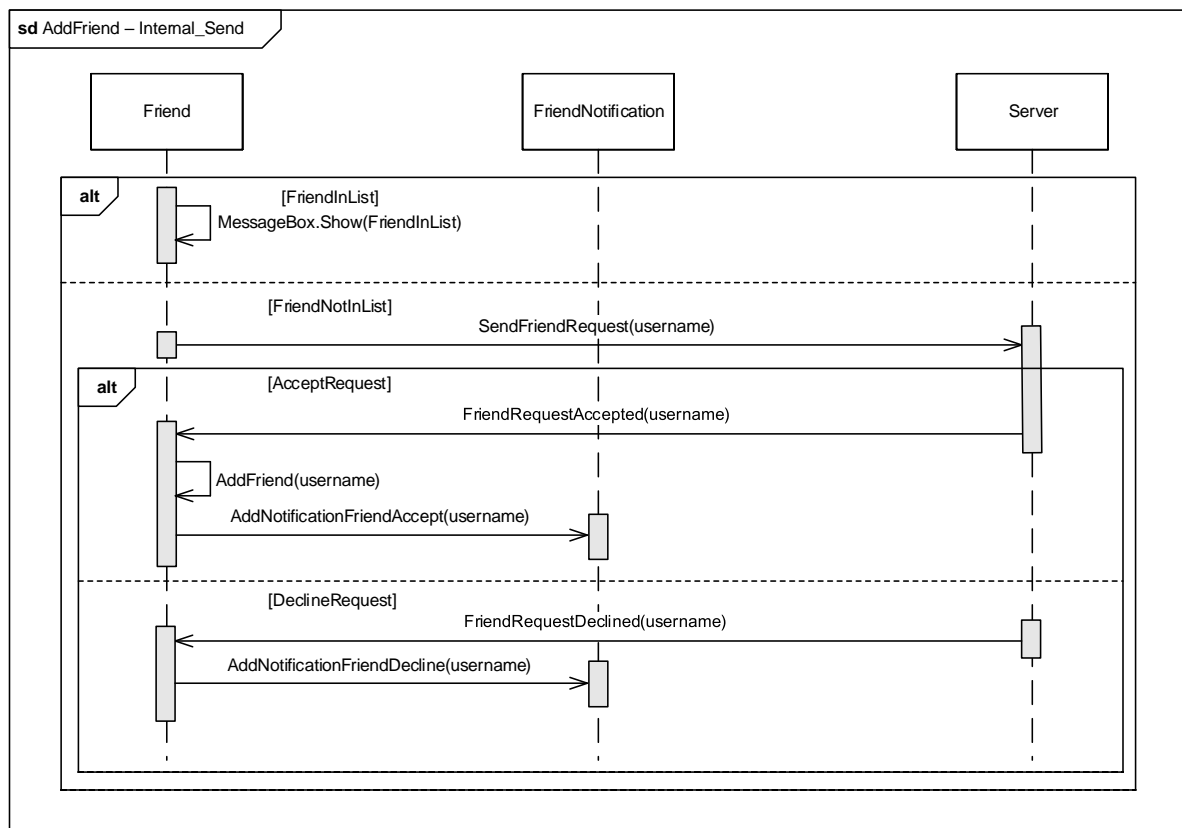
4.1.6.9.1 Design

Selve kommunikationen mellem to klienter og serveren, når der sendes en venneanmodning, kan ses på figur 30. Dette er en meget grov udgave af hvad der sker, uden at tage forbehold til intern funktionalitet i klienter eller på serveren. Der ses at afsenderen sender en venneanmodning, hvorpå modtageren har mulighed for at acceptere eller afvise anmodningen. Hvis modtageren accepterer, tilføjes afsenderen til modtagerens venneliste, og omvendt tilføjes modtageren til afsenderens venneliste. Ydermere tilføjes en notifikation hos afsenderen om at modtageren har accepteret venneanmodningen. Afviser modtageren derimod venneanmodning, får afsenderen blot en notifikation om at venneanmodning til modtageren er blevet afvist.



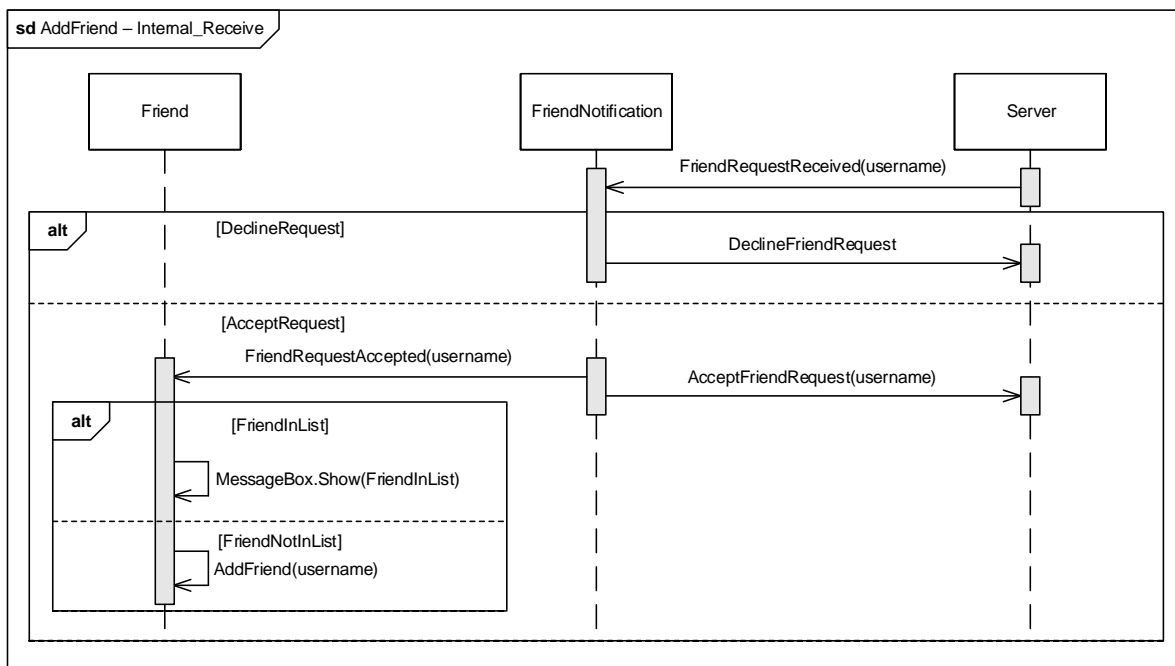
Figur 30: Sekvensdiagram over tilføj ven, ekstern kommunikation i fokus

For at fremme forståelsen for hvad der foregår internt i MartUI når en besked sendes til serveren, henvises der til figur 31. Der ses især her, at der laves et ekstra tjek, for at se om den ven, man forsøger at anmode, ikke allerede er på ens venneliste. Er vennen allerede på listen, vil der, frem for at tilføje en kopi, poppe en lille beskedboks op der informerer om at vennen allerede er på vennelisten.



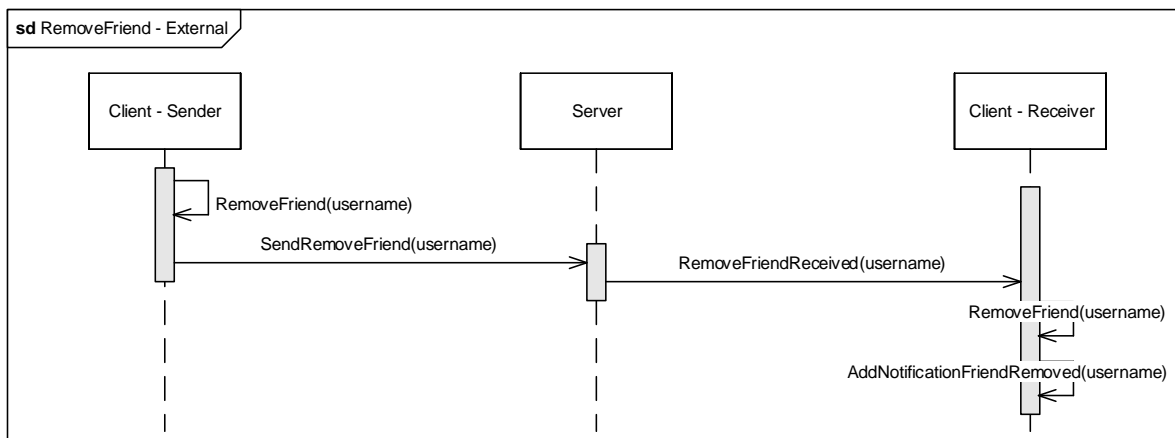
Figur 31: Sekvensdiagram over tilføj ven, intern send i fokus

På figur 32 ses der, at det samme gælder som for figur 31, netop at brugeren ikke kan acceptere en venneanmodning, hvis vennen allerede er på ens venneliste.



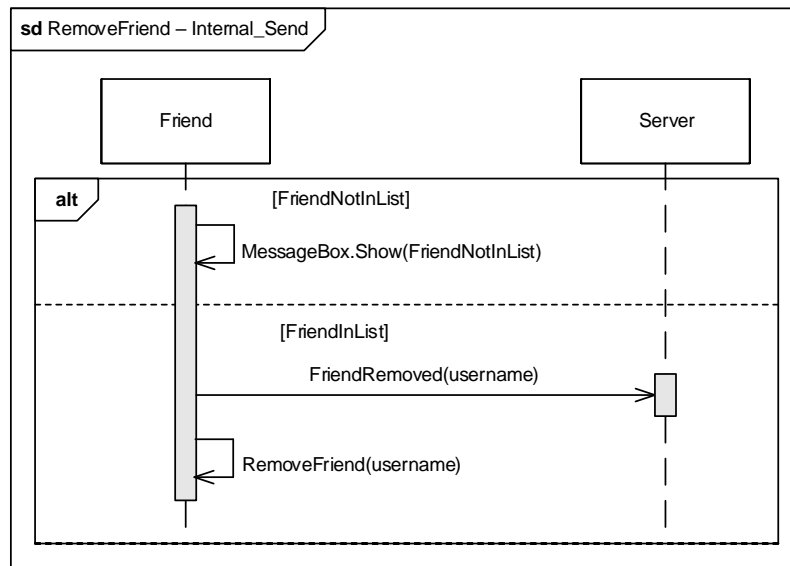
Figur 32: Sekvensdiagram over tilføj ven, intern modtag i fokus

Figur 33 fremviser et overordnet sekvensdiagram for fjernelse af en ven. I modsætning til at tilføje en ven, sendes der ved fjernelse ikke en anmodning, da dette ikke ville give mening. I stedet bliver den fjernede ven blot informeret om at de er blevet fjernet. Det vil sige at når en ven fjerner en fra deres venneliste, bliver vennen også fjernet fra ens egen venneliste, hvorpå en notifikation, om hvorfor vennen ikke længere er at finde på listen, kan findes under notifikationer.



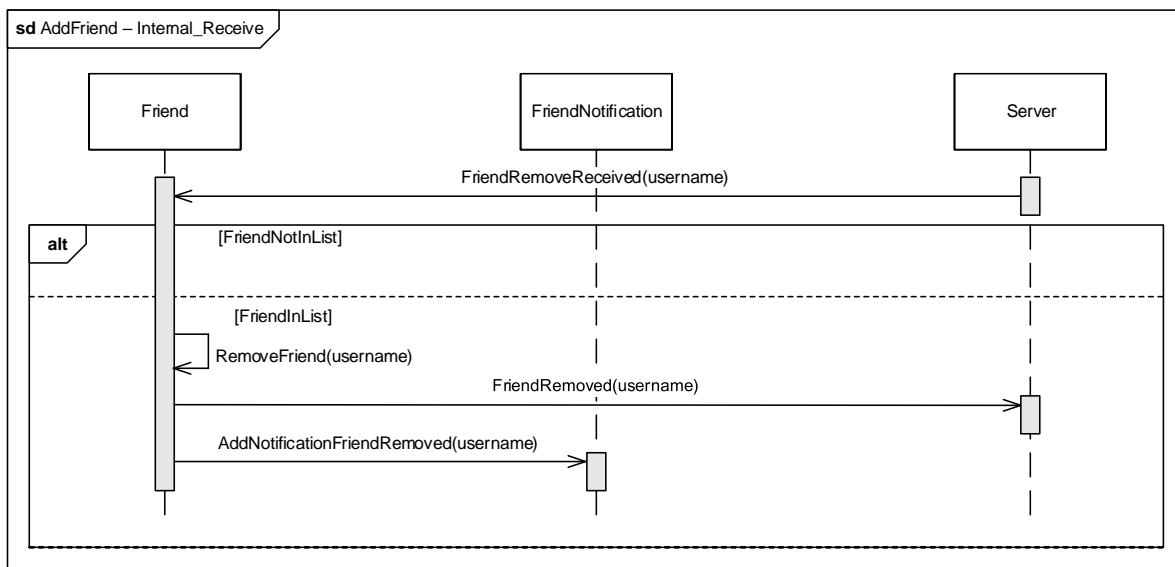
Figur 33: Sekvensdiagram over fjern ven, ekstern kommunikation i fokus

Der ses på nedenstående figur 34, at der ligesom med tilføjelse af ven også laves et lokalt tjek, for at sørge for at vennen rent faktisk er på vennelisten inden de fjernes.



Figur 34: Sekvensdiagram over fjern ven, intern send i fokus

Igen ses der på figur 35, at der laves et lokalt tjek for at sikre at vennen er på vennelisten. Er vennen ikke det, bliver beskeden blot ignoreret.



Figur 35: Sekvensdiagram over fjern ven, intern modtag i fokus

4.1.6.9.2 Implementering

For at forstå implementeringen af vennelisten i MartUI, er visse ting nødvendige. Først og vigtigst indeholder Friend, en venneliste bestående af FriendModels.

```
private ObservableCollection<FriendModel> _friendList;
```

En ObservableCollection minder meget om en List, den implementerer bare også:

```
INotifyCollectionChanged
```

hvilket vil sige, at listen og objekter i listen notificerer alt i Viewet der er afhængigt af den. Dette er essentielt for rent faktisk, visuelt, at opdatere vennelisten i MartUI. En FriendModel består af følgende attributer:

```
private string _username;
private ObservableCollection<ChatModel> _messageList;
private ChatViewModel _chat;
```

Det kan virke lidt bagvendt, i forhold til MVVM, at FriendModel indeholder en ObservableCollection af ChatModels, samt en ChatViewModel. Nogen gange giver det bare bedst mening at småbryde et design pattern, i dette tilfælde af rent logistiske årsager.

FriendModel indeholder en liste af beskeder(ChatModels) for at alle beskeder er knyttet til den rigtige ven, indgående såvel som udgående. Ligeledes har hver FriendModel en chat knyttet til sig.

Til sidst har hver FriendModel også et brugernavn, hvilket er det der knytter alle brugere sammen. Det er det eneste "officielle" data, man kan finde om en bruger ud over tags.

Når MartUI startes op bliver følgende funktion gennemført, efter login er godkendt:

```
private void HandleGetFriendList(string friendlist)
{
    FriendList.Clear();
    string[] temp = friendlist.Split(Constants.DataDelimiter);
    foreach (var f in temp)
    {
        if(f != "")
            FriendList.Add(new FriendModel(){Username = f});
    }

    var message = Constants.GetOldMessages + Constants.GroupDelimiter;
    _eventAggregator.GetEvent<SendMessageToServerEvent>().Publish(message);
}
```

Det der sker er at vennelisten ryddes, hvorefter en, potentielt, lang streng fra serveren splittes op i individuelle venner. Så længe vennens navn ikke er karakterløst, bliver de tilføjet til vennelisten. Det antages her at serveren ikke sender to identiske venner, da der ikke tages forbehold for det. Når vennelisten er fyldt op sendes der besked til serveren om, at brugeren vil modtage alle tidligere beskeder. Disse beskeder modtages så og inddeles i de korrekte venners beskedlister.

Når det klikkes på en ven bliver følgende metode kørt:

```
private void SelectFriend(FriendModel friend)
{
    _eventAggregator.GetEvent<SelectedFriendEvent>().Publish(friend);
    _eventAggregator.GetEvent<ChangeFocusPage>().Publish(friend.Chat);
}
```

Denne metode hæver et event som ChatViewModel abonnerer på. Det betyder at ChatViewModel holder informationer om hvilken ven, der er klikket på, og dermed kan fremvise den rigtige beskedliste. Ydermere skiftes der også view til vennens chat.

En anden vigtig funktionalitet der er implementeret i FriendViewModel er:

```
public class BindingProxy : Freezable
```

Formålet med denne klasse er egentlig at "snyde" det visuelle og logiske træ, da kontekstmenuen der fremkommer, når man højreklikker på en ven ikke er synlig i det visuelle eller logiske træ. Objekter

der er "Freezable" kan imidlertid godt tilgå DataContext, til elementer der ikke fremgår i det visuelle træ. [5]

Til sidst nævnes en funktionalitet som er anvendt i mange Views, netop skalérbare ObservableCollection. Dette skal forstås således at indholdet visuelt skalerer med det der rent faktisk er i den pågældende ObservableCollection. I dette tilfælde ens venner:

```
<ItemsControl ItemsSource="{Binding FriendList,
    UpdateSourceTrigger=PropertyChanged}" Width="185">
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Button
                Content="{Binding Username,
                    UpdateSourceTrigger=PropertyChanged}"
                Command="{Binding
                    DataContext.ChooseFriendCommand,
                    RelativeSource={RelativeSource
                        AncestorType={x:Type UserControl}}}"
                CommandParameter="{Binding}"
                Margin="0,3" Width="185" Style="{StaticResource
                    RoundCornerButton}">
                <Button.ContextMenu>
                    <ContextMenu>
                        <MenuItem Header="View Profile"
                            Command="{Binding
                                Source={StaticResource Proxy},
                                Path=Data.ViewProfileCommand}"
                            CommandParameter="{Binding}"/>
                        <MenuItem Header="Remove"
                            Command="{Binding
                                Source={StaticResource Proxy},
                                Path=Data.RemoveFriendCommand}"
                            CommandParameter="{Binding}"/>
                    </ContextMenu>
                </Button.ContextMenu>
            </Button>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

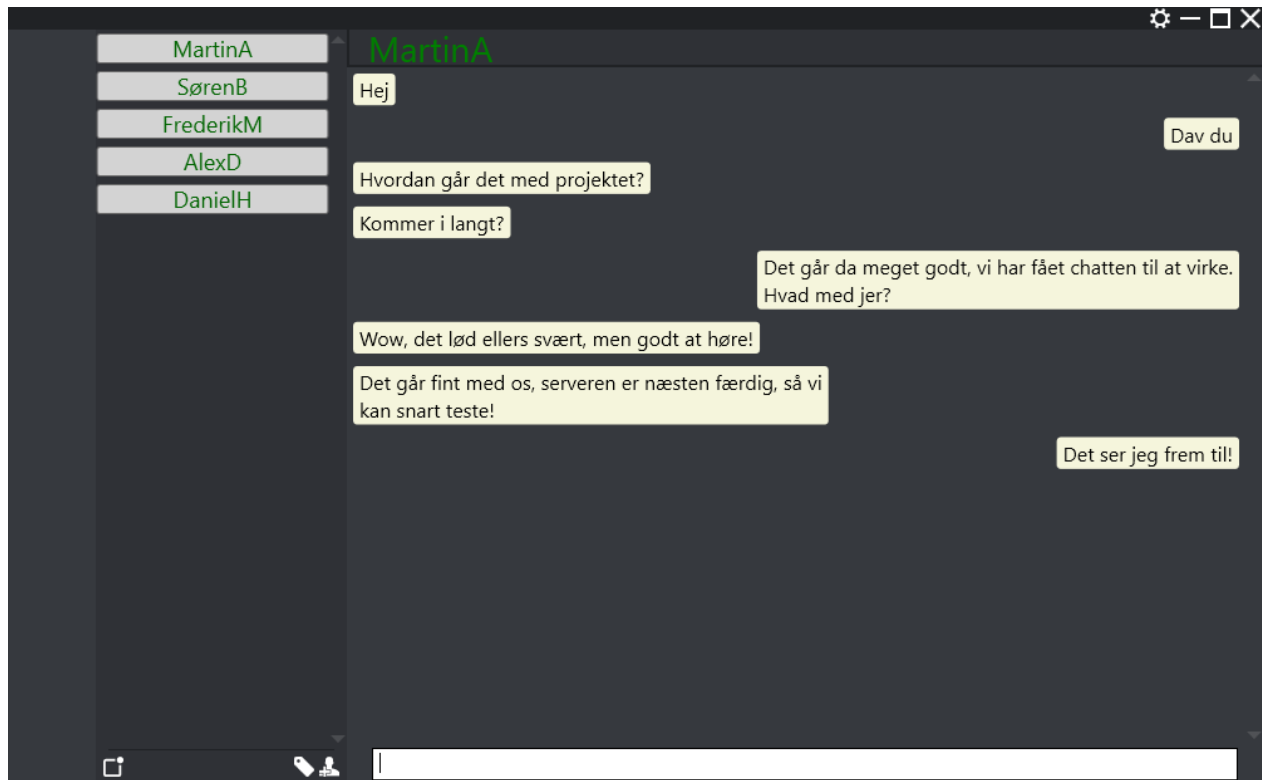
Det sker ved at alt dataen der bliver fremvist er bundet til FriendList, som opdateres med: "UpdateSourceTrigger=PropertyChanged". Og på grund af at der anvendes ObservableCollection og ikke lists, vil kollektionerne selv informere Viewet om, at de er blevet ændret, da et event bliver hævet. For hver FriendModel i listen bliver der så lavet en tilhørende knap, hvis content binder til brugernavnet og ChooseFriendCommand, hvilket er kommandoen der sætter SelectFriend metoden i gang og viser chatten frem. Ydermere kan her også ses den omtalte ProxyBinding, her under navnet Proxy, som fremgår i de to MenuItems.

4.1.6.10 Chat

Chatten i MartUI er stedet, hvor brugeren kan kommunikere med venner. Der er ikke rigtig andre krav, end at man skal have minimum en ven for at kunne chatte. For at åbne chat vinduet op skal man blot klikke på en vens navn i vennelisten, og chatten samt alle tidligere beskeder vil blive vist. Her er det så muligt at skrive beskeder til den valgte ven. Modtages en besked uden at chatten er åben vil denne besked stadig kunne ses, når chatten med vennen åbnes igen.

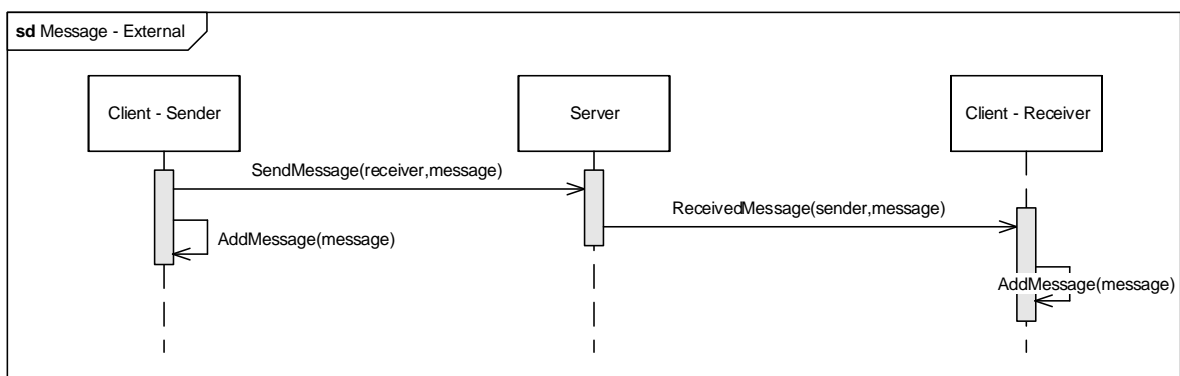
4.1.6.10.1 Design

På nedenstående figur 36 ses et billede af MartUI med chatten åben. På venstre hånd ses vennelisten, og i toppen kan der ses, at brugeren har "MartinA's" chat åben. Der er blevet skrevet beskeder frem og tilbage, og beskederne på venstre hånd er fra "MartinA", mens beskederne på højre er fra brugeren selv.



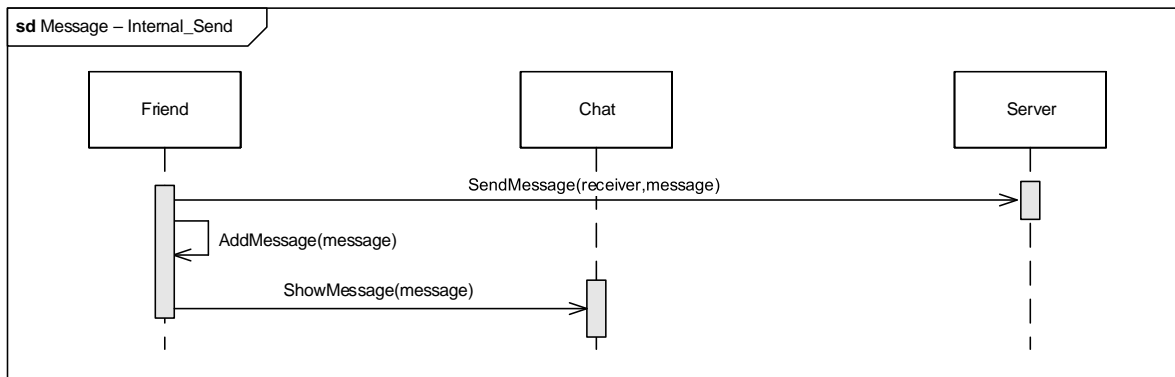
Figur 36: Billede af MartUI der viser chatten og vennelisten

Figur 37 viser den eksterne kommunikation med en anden klient, når der sendes en besked.

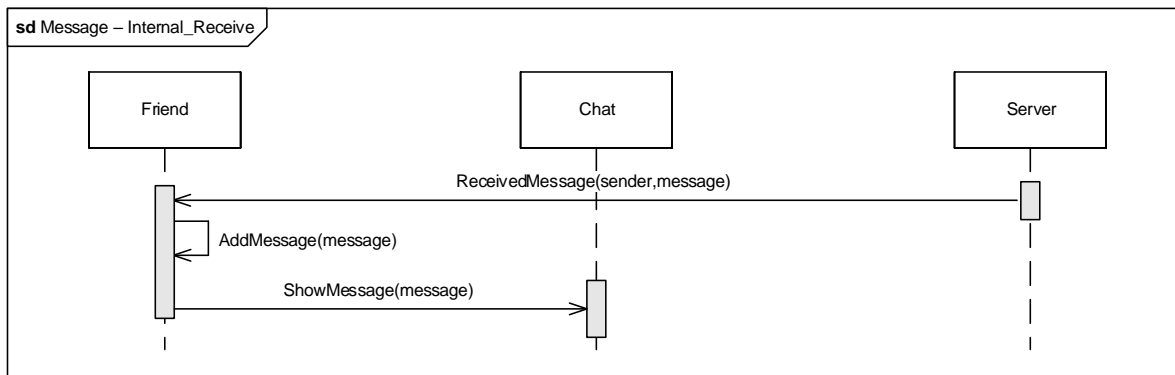


Figur 37: Sekvensdiagram over besked, ekstern kommunikation i fokus

I modsætning til Friend, laves der ikke et tjek for at tjekke om beskeden allerede er i listen, da man gerne må sende to ens beskeder. Dette er illustreret på figur 38 og 39. Der ses også her at selve tilføjelsen af beskeden foregår igennem Friend, og ikke Chat. Chat fremviser blot den nye besked.



Figur 38: Sekvensdiagram over send besked, intern send i fokus



Figur 39: Sekvensdiagram over modtag besked, intern modtag i fokus

4.1.6.10.2 Implementering

Som der ses på ovenstående diagrammer, foregår rigtig meget af chatten i Friend. Dette er fordi FriendModel indeholder en liste af ChatModel(beskeder). En ChatModel ser ud på følgende måde:

```

private string _message;
private string _sender;
private string _receiver;
private string _messagePosition;
  
```

Message, Sender og Receiver, gøremål siger lidt sig selv. MessagePosition definerer hvilken side af skærmen beskeden skal fremstå i. Altså højre hvis det er en besked brugeren selv har afsendt og venstre, hvis det er en besked, brugeren har modtaget. Message, Receiver og Sender er noget der deles op i klienten når beskeden modtages fra serveren. Det bliver sendt som en string til serveren

fra MartUI.

Når en ny besked modtages fra serveren køres den igennem følgende metode i FriendViewModel:

```
private void HandleNewMessage(ChatModel message)
{
    foreach (var friend in FriendList)
    {
        if (message.Sender == UserData.Username &&
            friend.Username == message.Receiver)
        {
            message.MessagePosition = "Right";
            friend.MessageList.Add(message);
            break;
        }
        else if (message.Sender == friend.Username &&
            UserData.Username == message.Receiver)
        {
            message.MessagePosition = "Left";
            friend.MessageList.Add(message);
            break;
        }
    }
}
```

Bekseden sendes altså ud til rigtige side, og da gamle beskeder modtages når brugeren bliver logget på, tjekkes der om beskeden skal forekomme i venstre eller højre side. UserData.Username er en Singleton instans af brugerens eget brugernavn.

Når en besked afsendes foregår det via metoden fra ChatViewModel:

```
private void SendMessage()
{
    if (TextToSend != "")
    {
        var message = new ChatModel();
        message.Sender = UserData.Username;
        message.Message = TextToSend;
        message.MessagePosition = "Right";
        message.Receiver = User.Username;
        var msg = Constants.Write + Constants.GroupDelimiter +
            message.Receiver +
                Constants.GroupDelimiter + message.Message;
        _eventAggregator.GetEvent<NewMessageEvent>().Publish(message);
        Application.Current.Dispatcher.Invoke(() =>
```



```
        {  
            _eventAggregator.GetEvent<SendMessageToServerEvent>().Publish(msg);  
        });  
        TextToSend = "";  
    }  
}
```

Her laves der aldrig et check om hvilken side beskeden skal placeres i, da det altid er højre side. Modtagen her er den "SelectedFriend" som FriendViewModel sender over til chatten. Teksten der bliver sendt, er den tekst brugeren har skrevet i tekstboksen, når der trykkes på enter. Eventet der sender beskeden til serveren gennem klienten bliver invoket fra dispatcheren, for at undgå at MartUI kommer til at køre på samme tråd som klienten.

I ChatView ses der at listen af beskeder binder til den nuværende bruger, altså den bruger der er klikket på i vennelisten:

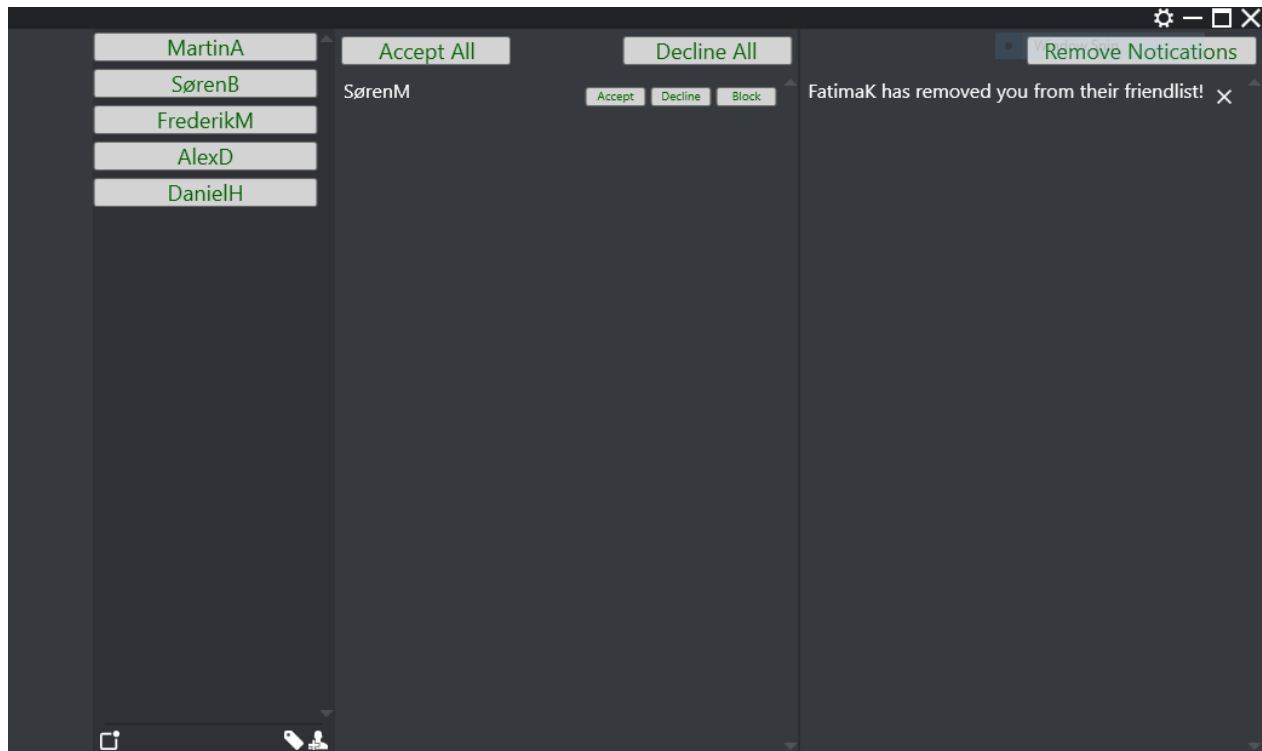
```
<ItemsControl ItemsSource="{Binding User.MessageList,  
    UpdateSourceTrigger=PropertyChanged}">
```

4.1.6.11 FriendNotification

I FriendNotification kan brugeren se informationer om venneanmodninger og generelle beskeder om venner, der har slettet brugeren fra deres venneliste eller accepteret/afvist en venneanmodning. Det er også herfra, muligheden for at acceptere og afvise venneanmodninger er.

4.1.6.11.1 Design

Figur 40 viser et eksempel på hvilke informationer FriendNotification kan indeholde. Helt til venstre ses vennelisten og i venstre side i bunden ses selve knappen der navigerer til FriendNotification. Efter der er trykket på den frembringes højre side af vinduet, som ses splittet op i to kolonner. Venstre kolonne indeholder venneanmodninger og højre side ses notifikationer. Som det ses er der mulighed for at acceptere eller afvise alle venneanmodninger og fjerne alle notifikationer. Det er også muligt at acceptere, afvise eller blockere individuelle venneanmodninger, samt at fjerne enkelte notifikationer.



Figur 40: Billede af venneanmodninger og notifikationer i MartUI

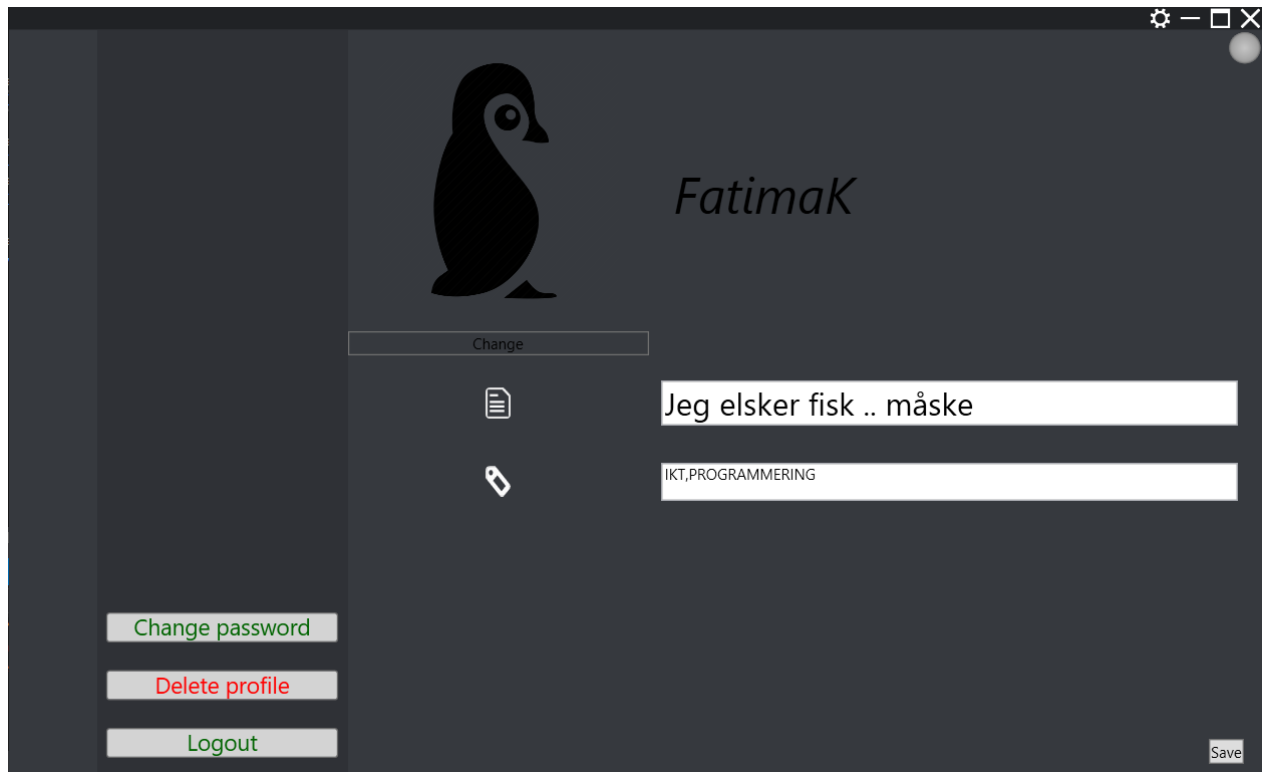
4.1.6.11.2 Implementering

Det meste implementering fra FriendNotification er gennemgået i afsnittet Friend. Dette er fordi de kaldende skal gennemgå vennelisten for at tjekke at vennen rent faktisk er i listen eller ikke allerede er i listen.

4.1.6.12 Profile

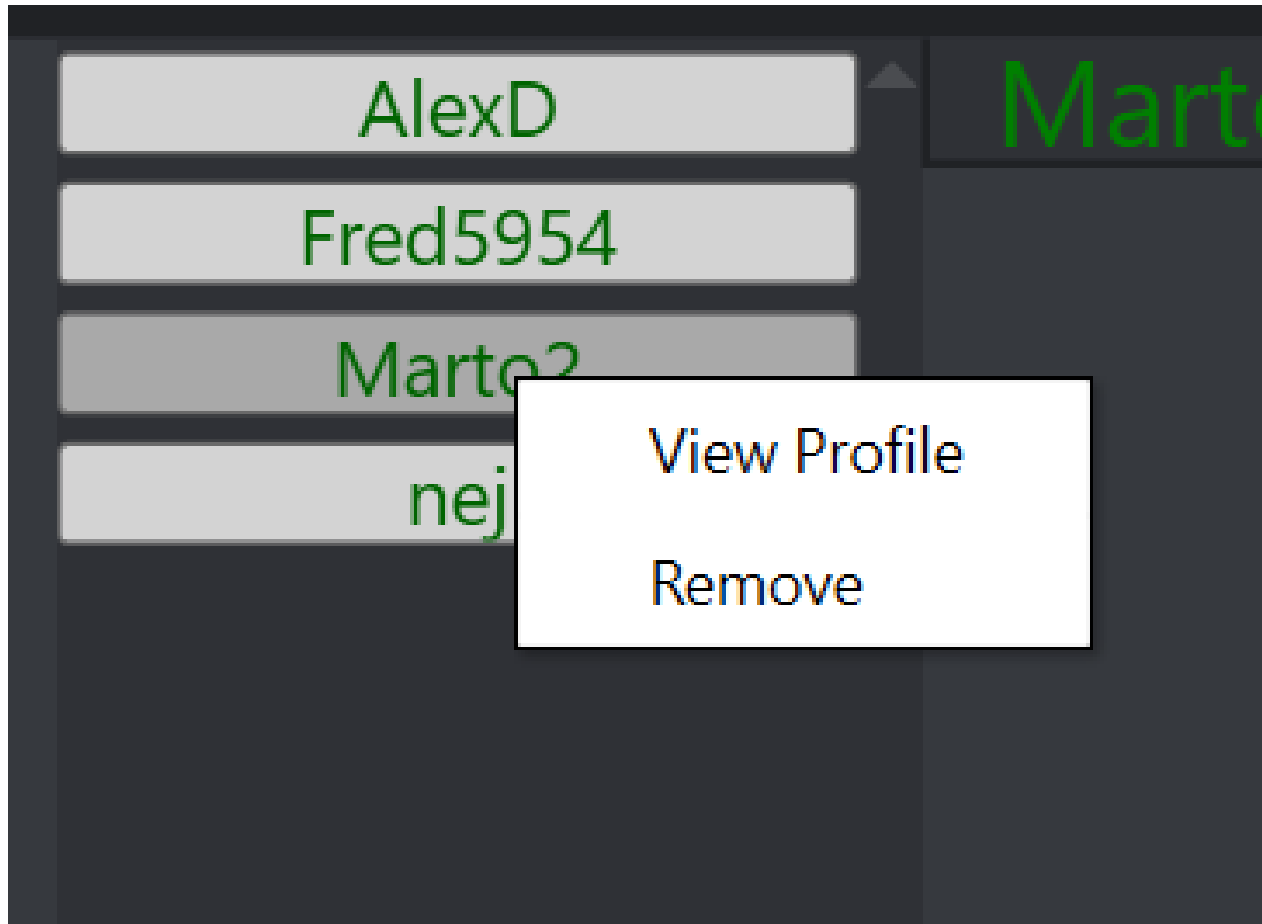
4.1.6.12.1 Design

Figur 41 viser et billede af profilen af brugeren, hvor brugeren har mulighed for at ændre sin profil. Profilen er opstillet, så det er nemt for brugeren at identificere hvor man kan skrive om sig selv og hvor man skriver om sine tags. Billedet er ikke sendt med fra serveren, men kan indsættes ved at trykke på "Change". Brugernavnet kan ses ved siden af billedet og beskrivelse og tags ses nedenfor. Der er indsat et lille ikon ved siden af beskrivelse og tags-boksene. Dette giver et brugervenligt touch, som der tit anvendes i mange applikationer. Tags og beskrivelse kan ændres af brugeren. For at gemme, kan der trykkes på 'Save' eller knappen i højre hjørne. Trykkes på denne mens der ikke er gemt ændringer, vil brugeren få en besked om, at ændringer ikke er gemt, hvor brugeren kan vælge at gemme eller lade være.



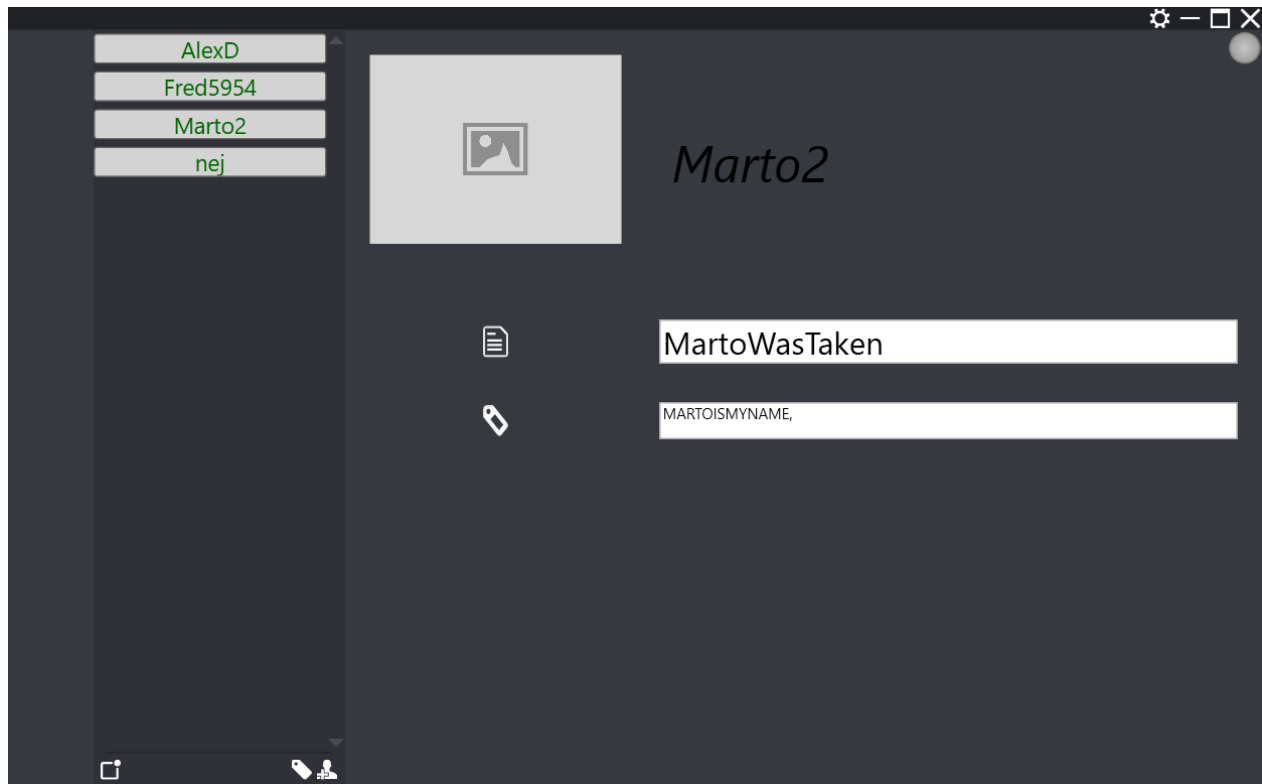
Figur 41: Billede af indstillinger og profil i MartUI

Det er muligt at se en anden brugers profil gennem vennelisten. Højreklikkes på en profil og trykkes der "View Profile" bliver man navigeret til en anden profil. Denne undermenu ses på figur 42.



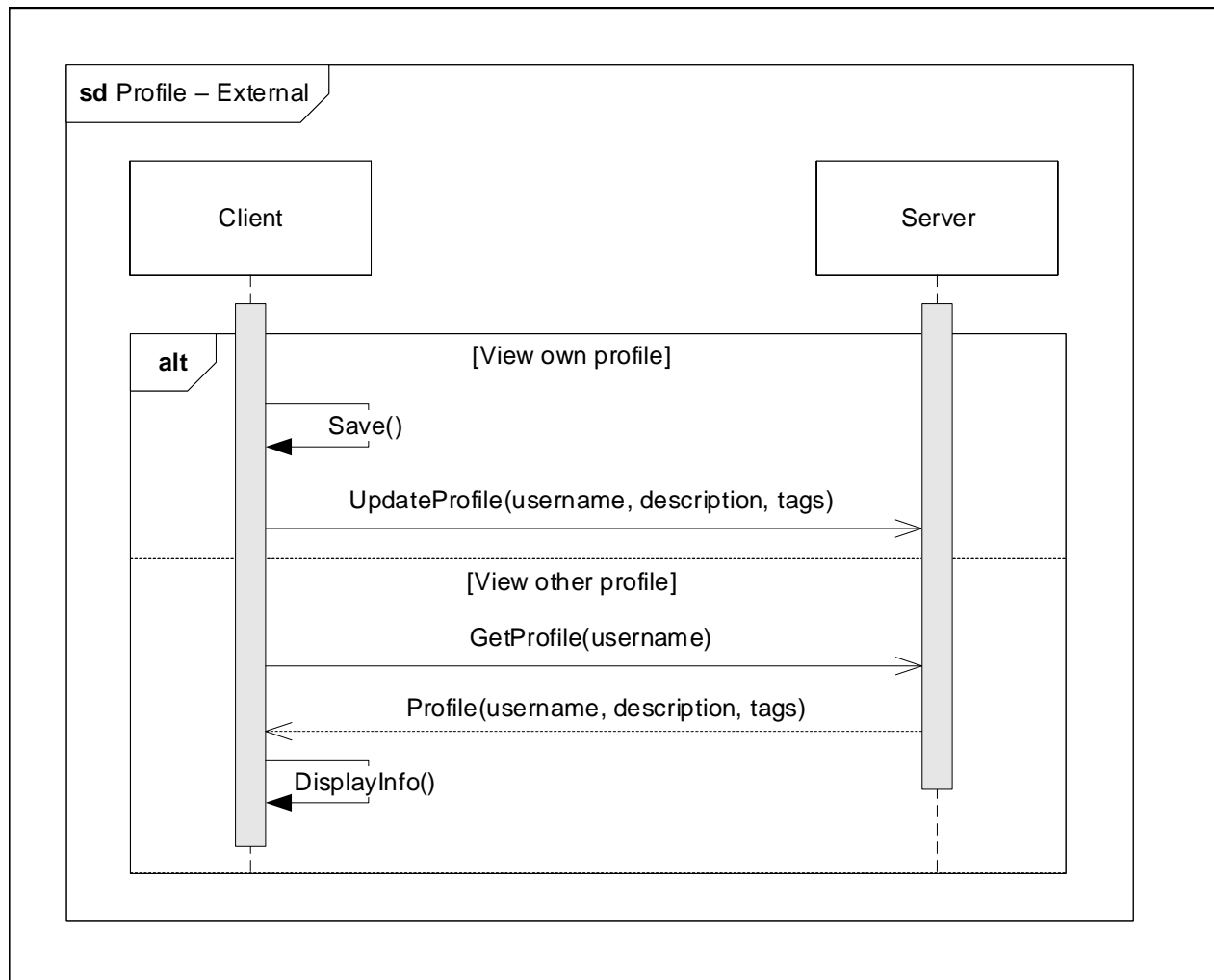
Figur 42: Højreklik på en ven som viser en undermenu

På 43 vises outputtet af at åbne en anden brugers profil. Det er lavet således, at det ikke er muligt for brugeren at ændre andres profiler eller at kunne skrive i felterne uden at de faktisk gemmes. Knapperne for at gemme ny information og ændring af billedet er også fjernet. Dette gør det nemt for brugeren at indse at det ikke er muligt at ændre andres profiler, blot at se på dem.



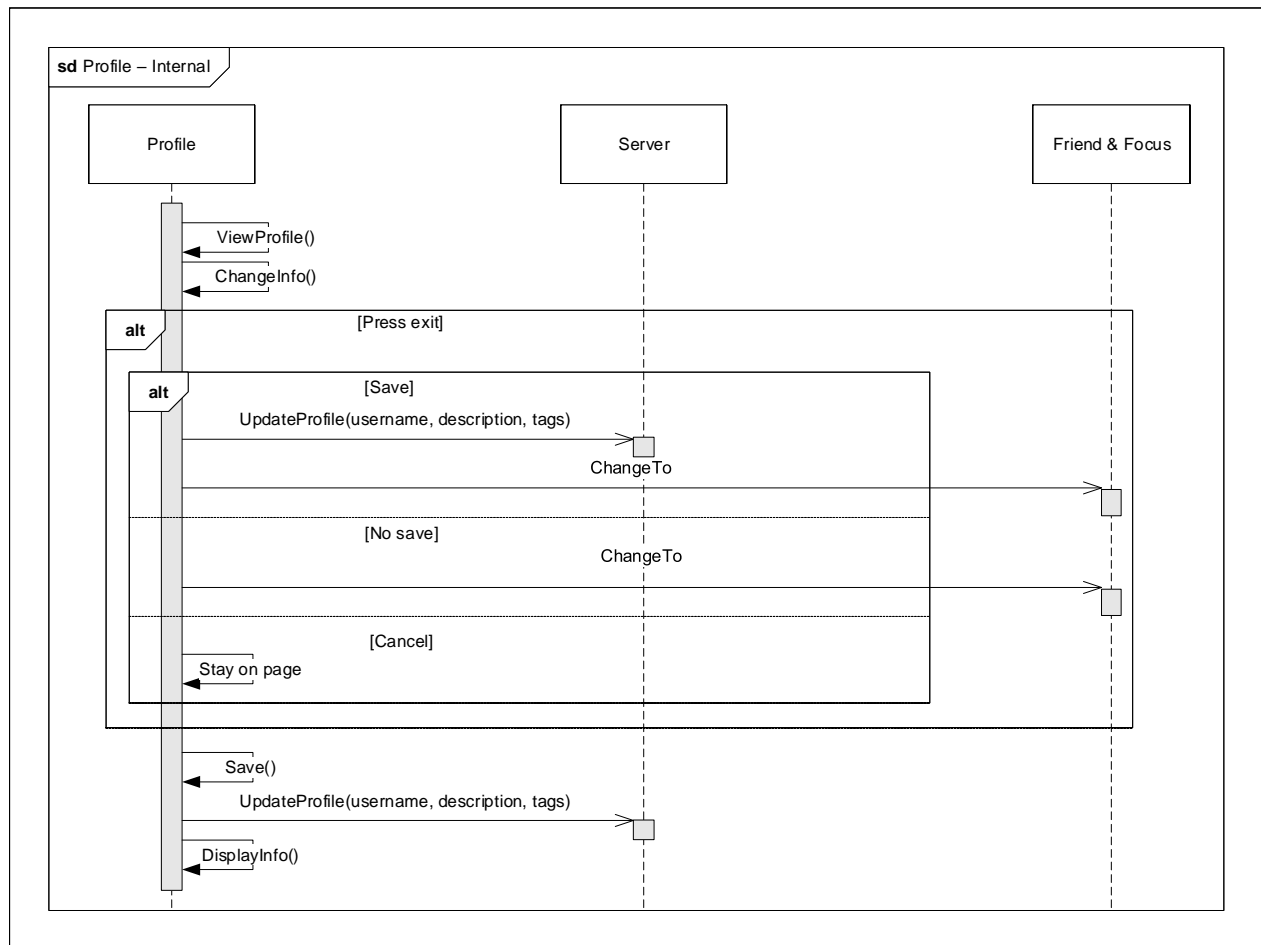
Figur 43: Billede af en anden profil

Figur 44 viser sekvensdiagrammet, hvor forbindelsen mellem klienten og serveren er i fokus. Klienten trykker "Save" og opdateringerne sendes til serveren. Vælger brugeren at se på en andens profil, bliver der anmodet om brugeren fra serveren, hvorefter den vises på klienten.



Figur 44: Sekvensdiagram af opdatering af en profil - client og server i fokus

Figur 45 viser den interne programkode, hvor der er taget udgangspunkt i, at brugeren ser og ændrer sin egen profil. Brugeren ændrer sin profil, og trykker på knappen i hjørnet som vil føre brugeren tilbage til sin venneliste. Dette medfører at en bekræftelsesdialogbox popper op. Brugeren kan vælge at gemme og skifte til sin venneliste, ikke gemme og skifte, eller blot annullere handlingen, hvormed brugeren forbliver på sin profil uden at gemme ændringerne. Denne ekstra funktion gør det muligt for brugeren at gemme informationer, selv hvis brugeren ikke har trykket på "Save"-knappen.



Figur 45: Sekvensdiagram af opdatering af en profil - intern kode i fokus

4.1.6.12.2 Implementering

Implementeringen består i at hente og parse information fra serveren men også håndtere dem lokalt. Det samme view anvendes til at vise alle profiler, derfor er det vigtigt at holde styr på om det er brugerens profil eller en vens profil, som der ønskes at inspicere.

En property, OtherUser, bruges til at vide, om det er brugerens egen profil der skal vises eller en anden bruger. Den bruges inde i XAML koden til aktivere og deaktivere de to knapper til at gemme og ændre profilbilledet. Dette er en style som aktiveres hvis OtherUser er lig med true, som gør at tekstboksen bliver ReadOnly, og knappen bliver usynlig. Dette er indsat i ressourcerne:

```

<UserControl.Resources>
    <Style x:Key="CannotChange" TargetType="{x:Type TextBox}">
        <Style.Triggers>
            <DataTrigger Binding="{Binding OtherUser}" Value="True">
                <Setter Property="IsReadOnly" Value="True" />
            </DataTrigger>
        </Style.Triggers>
    </Style>
</UserControl.Resources>
  
```

```
        </Style.Triggers>
    </Style>
    <Style x:Key="CannotSee" TargetType="{x:Type Button}">
        <Style.Triggers>
            <DataTrigger Binding="{Binding OtherUser}" Value="True">
                <Setter Property="Visibility" Value="Hidden" />
            </DataTrigger>
        </Style.Triggers>
    </Style>
</UserControl.Resources>
```

Meget i denne kode minder om Login og Create User, men i denne kode bliver tags hentet og tilføjet til tekstboksen som en enkelt string, siden tags er indsat i en liste.

```
private string ConvertTagsToString(char delimiter)
{
    if (UserData.Tags.Any())
    {
        StringBuilder str = new StringBuilder();

        for (int i = 0; i < UserData.Tags.Count - 1; i++)
            str.Append(UserData.Tags[i] + delimiter);

        str.Append(UserData.Tags[UserData.Tags.Count - 1]);

        return str.ToString();
    }

    return "";
}
```

Når brugeren trykker på "Save" bliver ændringer gemt og sendt til serveren:

```
private void Save()
{
    UpdateTags();
    UserTagsInOneString = ConvertTagsToString(',');
    UserData.Description = Description;

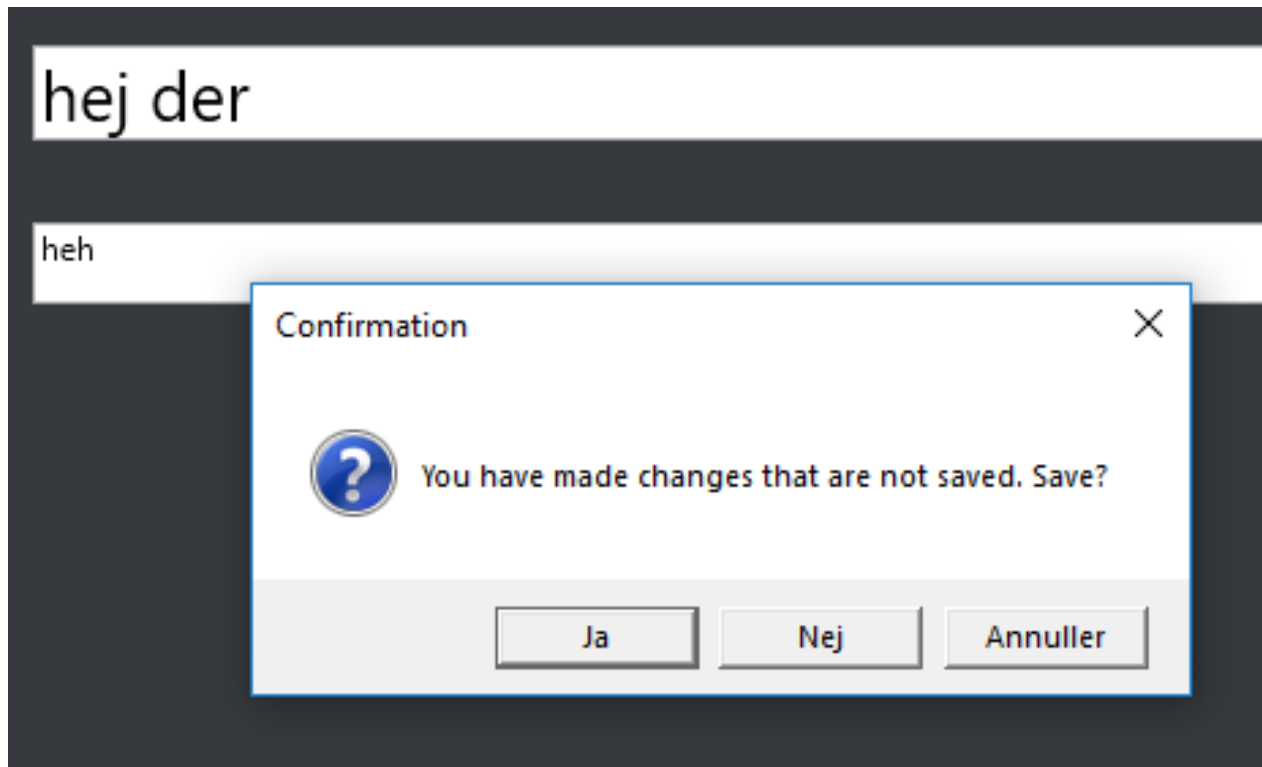
    var tagsToSend = ConvertTagsToString(Constants.DataDelimiter);

    // Send profile to server
    var msg = Constants.UpdateProfile + Constants.GroupDelimiter +
```



```
        UserData.Username + Constants.GroupDelimiter +  
            UserData.Description  
        + Constants.GroupDelimiter + tagsToSend;  
  
        _eventAggregator.GetEvent<SendMessageToServerEvent>().Publish(msg);  
    }
```

Vælger brugeren at trykke på tilbage-knappen, før ændringer er gemt, bliver man advaret med en popup-boks, som der ses på figur 46.



Figur 46: Billede af pop up

Koden hertil er en MessageBox som viser en Confirmation box med "Ja/Nej/Annuller". Vælger brugeren "Ja", gemmes ændringer samt skiftes der view. Vælges "Nej" gemmes ændringer ikke og der skiftes stadig view. Vælges "Annuller" bliver ændringer ikke gemt og der skiftes ikke view. Følgende kodeudsnit viser et kodeudsnit af denne kode.

```
var result = MessageBox.Show("You have made changes that are not  
    saved. Save?", "Confirmation",  
        MessageBoxButton.YesNoCancel,  
        MessageBoxImage.Question);  
  
if (result == MessageBoxResult.Cancel)
```

```
        return;

    if (result == DialogResult.Yes)
    {
        Save();
    }
    else if (result == DialogResult.No)
    {
        Tags = UserTagsInOneString;
        Description = UserData.Description;
    }
```

Til sidst vil der skiftes view og data vil 'resettes'. Dvs. næste gang man går ind på "Profile" via "Settings" vil brugerens egen profil vises igen samt kan den også ændres. Dette gøres ved at sætte brugernavnet, beskrivelsen og tags tilbage til brugerens informationer. OtherUser bliver sat lig false.

Forespørges der om en anden profil, vil sendes en forespørgsel til serveren om en profil, mens OtherUser sættes lig false. Når brugeren modtages, sker der parsing af profilen, hvormed den herefter bliver sat.

4.1.6.12.3 Test

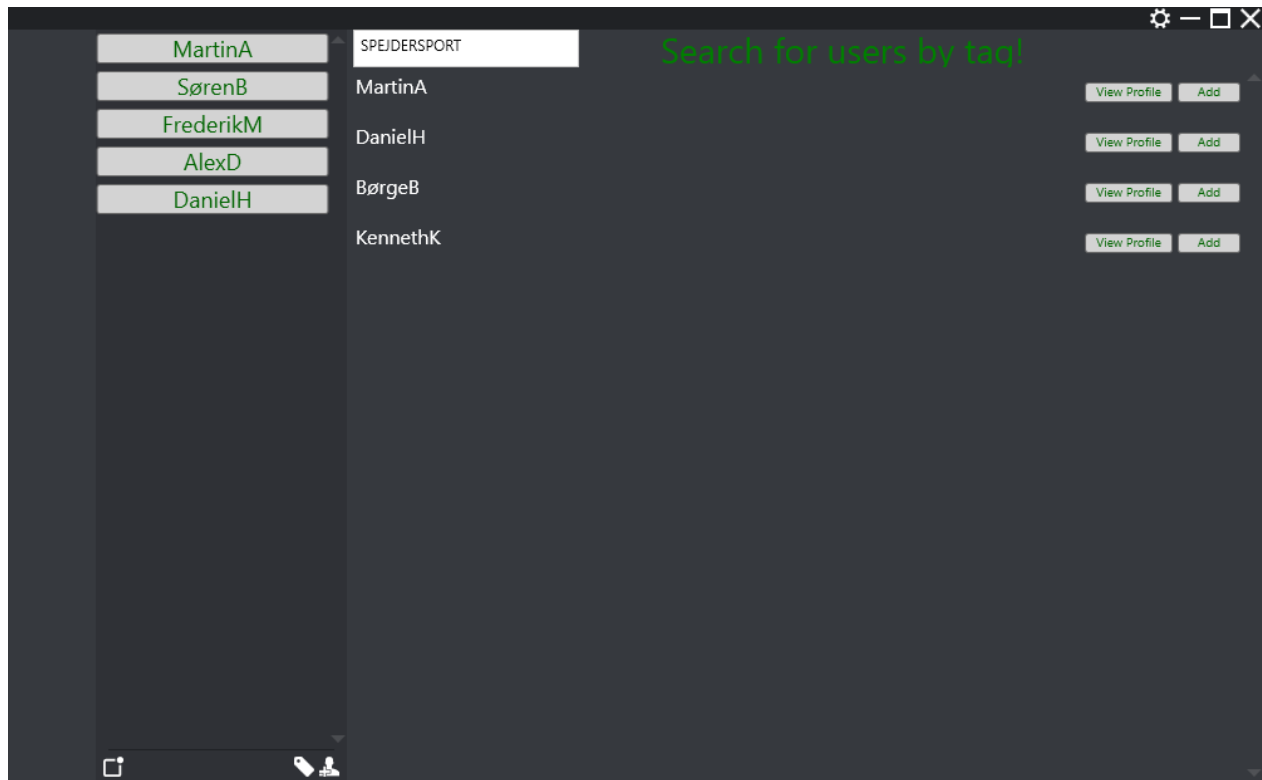
Der er lavet en test om brugeren kan skifte mellem sin egen profil og andres profil, samt om der kan ændres på andre profiler. Dette kan ses på figur 41 og 43. Det er også muligt at ændre tags og beskrivelse. Dette er nemt at verificere, da det sker lokalt, og ikke sendes over serveren.

4.1.6.13 Tag

Brugeren har mulighed for at finde, eller i hvert fald søge efter, venner ud fra tags. Hver person har muligheden for at knytte tags til deres profil. Et tag kan være hvad som helst, fx: GUITAR, PROGRAMMERING, og så videre. Ud fra disse tags kan andre profiler finde en. Søger man efter et tag, vil alle personer med det tag blive vist på brugergrænsefladen. Derefter har man mulighed for at: søge efter et nyt tag, tilføje end af dem der kommer frem eller se en dem der kommer frems profil.

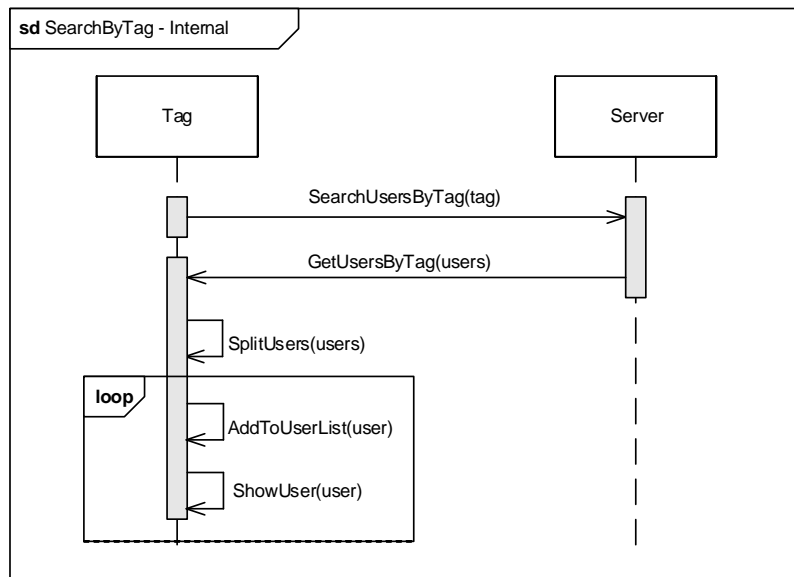
4.1.6.13.1 Design

Figur 47 viser et billede af TagView. Der ses her at brugeren har søgt efter "SPEJDESPORT" og at fire brugere er poppet frem, hvoraf to af dem allerede er på vennelisten, mens to andre ikke er.



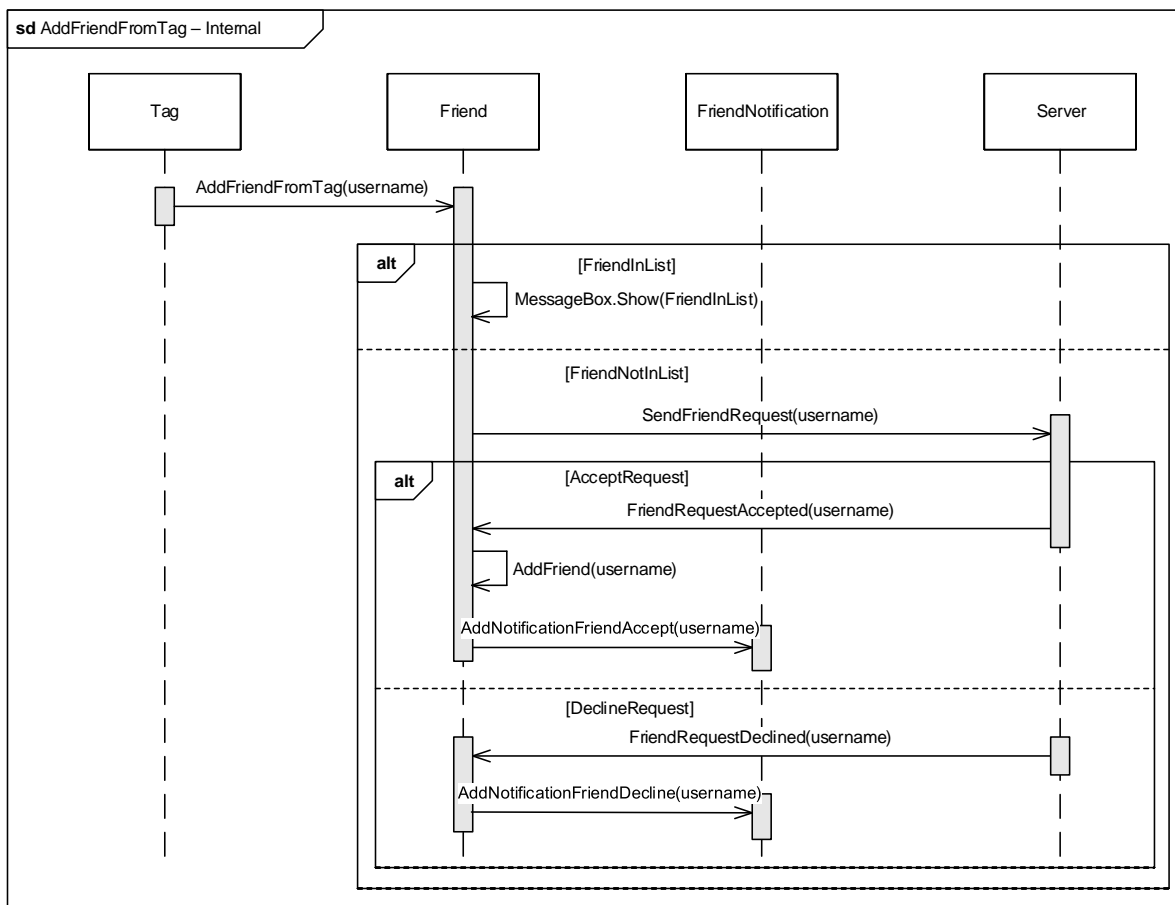
Figur 47: Billede af tag i MartUI

På nedenstående sekvensdiagram, figur 48, ses hvordan brugeren søger efter tags på serveren. Et svar vil blive returneret med en lang streng af brugere, som splittes op i individuelle brugere, og tilføjes til listen som fremvises.



Figur 48: Sekvensdiagram over søgning efter tag, intern søgfunktion i fokus

Figur 49 viser et sekvensdiagram og tilføjelse af en ven fra TagView. Den er næsten magen til metoden der tilføjer venner fra Friend, dog kommer kaldet her fra Tag.



Figur 49: Sekvensdiagram over send venneanmodning fra tags, intern send i fokus

4.1.6.13.2 Implementering

Når der søges efter et tag, vil serveren sende alle personer med tagget tilbage. Dette blive håndteret på følgende måde:

```

private void HandleGetTag(string username)
{
    UserList.Clear();

    var users = username.Split(Constants.DataDelimiter).ToList();
    foreach (var u in users)
    {
        if (!string.IsNullOrEmpty(u))
            UserList.Add(u);
    }
}
  
```

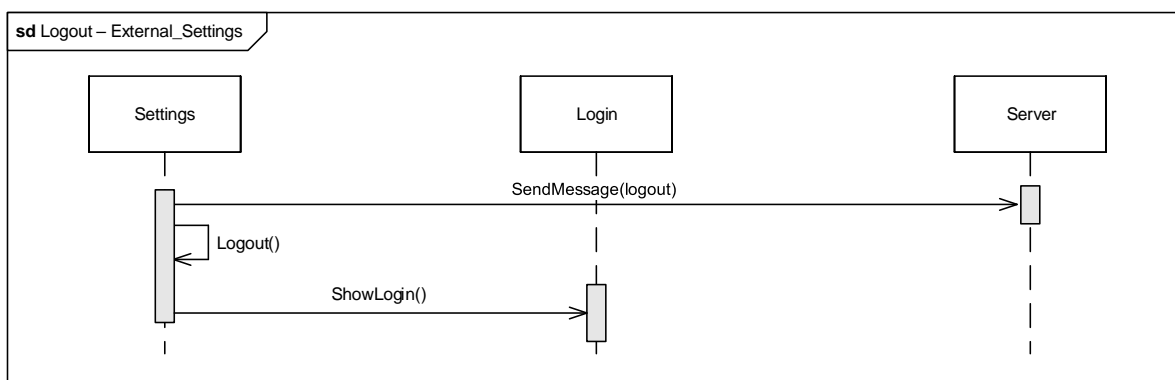
Ligesom med når vennelisten initielt bliver modtaget, bliver listen af brugere med det eftersøgte tag, også nulstillet inden der sættes brugere ind i den.

4.1.6.14 Settings

På nuværende tidspunkt er det kun log ud funktionen i settings, der har funktionalitet. På senere tidspunkt kunne dette udvides.

4.1.6.14.1 Design

På nedenstående figur 50 ses et logout kald. Der sendes først logout til serveren, hvorefter et lokalt logout kald kaldes, og login skærmen frembringes.



Figur 50: Sekvensdiagram over log ud-funktionalitet i indstillinger, intern log ud i fokus

4.1.6.15 Windowsfunktionalitet

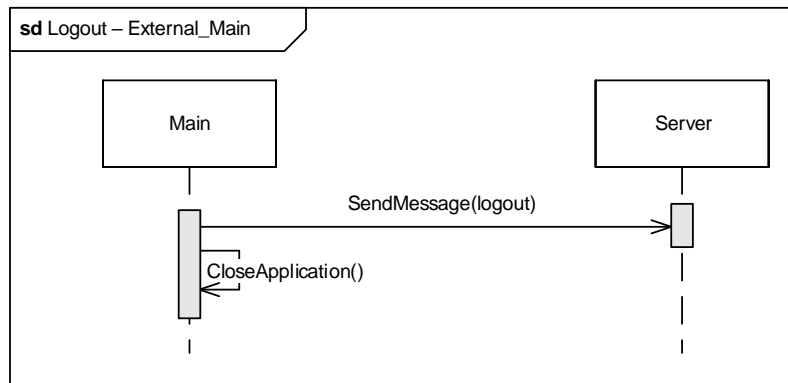
Det har været nødvendigt, at "gen-implementere" Windowsfunktionaliteten, da der fra start var ønsket at lave en brugerdefineret titellinje. Da titellinjen, samt outline på vinduer, er det der binder Windowsfunktionalitet, har det været nødvendigt at undersøge gen-implementering af Windowsfunktionaliteten.

Da der er lavet en modificeret titellinje frem for Windows' standard, har det været nødvendigt at, undersøge hvilke muligheder der er til dette. Det har imidlertid vist sig, at det ikke er et problem, mange har valgt at løse. Det er den generelle konsensus, blot at bruge Windows' standard funktionalitet, selvom det indebærer at anvende deres titellinje, og hvide outline på alle vinduer. Dette kan dog godt fremstå lettere uprofessionelt, på en consumer-gearet applikation, og der er derfor trods besværligheden, blevet lagt tid og energi i at opnå Windowsfunktionalitet, med en brugerdefineret titellinje.

Det er især selve, lukning af program, maksimering af program og minimering af program der er lagt fokus på.

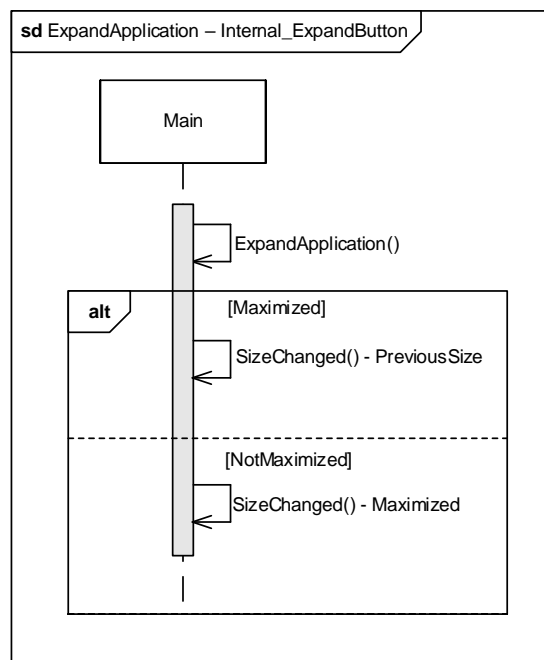
4.1.6.15.1 Design

Der vil ikke blive lagt overvejende vægt på design eller implementering, da dette blot er et sideafsnit. Figur 51 viser hvad der sker, når der trykkes krydset i øverste højre hjørne.



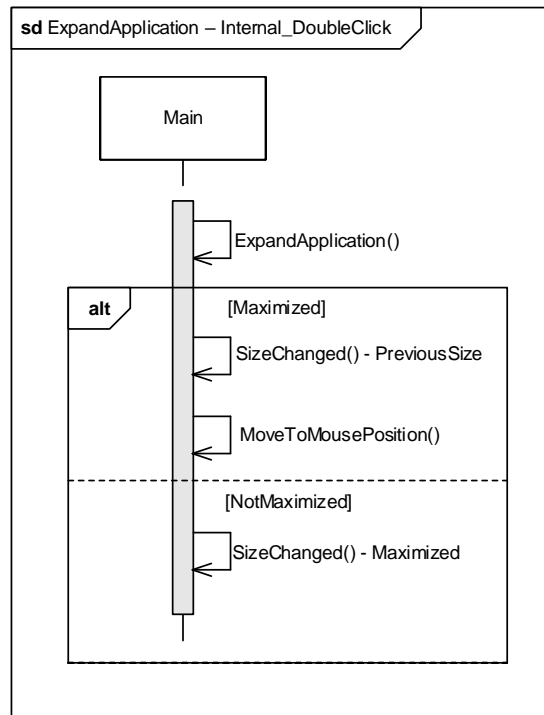
Figur 51: Sekvensdiagram over luk applikation, ekstern kommunikation i fokus

Figur 52 er et sekvensdiagram over metodekaldene, der maksimerer eller kalder vinduet tilbage til forrige størrelse, ved tryk på Expand Window knappen.



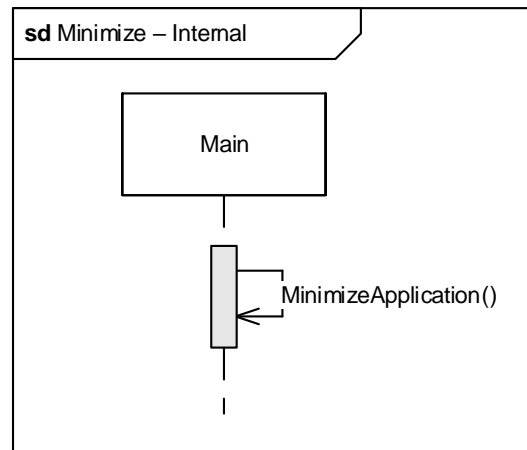
Figur 52: Sekvensdiagram over udvid applikation ved klik på udvid applikation-knappen

Figur 53 er næsten magen til ovenstående sekvensdiagram, dog med den forskel at vinduet automatisk placeres ved musen, når vinduet bringes tilbage til original størrelse.



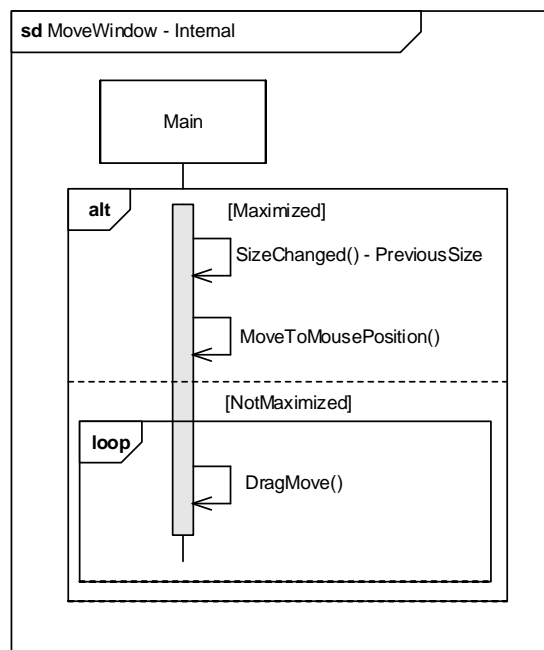
Figur 53: Sekvensdiagram over udvid applikation ved dobbeltklik på titellinjen

Nedenstående figur 54 viser Minimize funktionen.



Figur 54: Sekvensdiagram over minimer applikation

På figur 55 ses funktionaliteten når vinduet skal flyttes. Vinduet bliver automatisk lavet tilbage til forrige størrelse, hvis det er maksimeret.



Figur 55: Sekvensdiagram over translokation af applikation

4.1.6.15.2 Implementering

Implementeringen af Windowsfunktionalitet er lavet i code-behind i MainView.

Da WPF's Maximize funktion bringer skærmen i fullscreen mode, har det været nødvendigt at implementere noget logik, der forhindrer dette, således at taskbaren stadig er synlig:

```
private void MainView_SizeChanged(object sender, SizeChangedEventArgs e)
{
    if (WindowState == WindowState.Maximized) //Makes sure the window
        doesn't fullscreen when maximized
    {
        WindowState = WindowState.Normal;
        PrevLeft = Left;
        PrevTop = Top;
        PrevWidth = Width;
        PrevHeight = Height;
        Width = SystemParameters.WorkArea.Width;
        Height = SystemParameters.WorkArea.Height;
        Top = SystemParameters.WorkArea.Top;
        Left = SystemParameters.WorkArea.Left;
        Maximized = true;
    }
}
```

Som der ses laves der alternativ funktionalitet for at "snyde" Windows, når den ellers ville gå fullscreen. Den nuværende skærmstørrelse gemmes i "Prev" variablerne og den aktuelle skærmstørrelse, sættes til skrivebordets størrelse. Dette bør skalere til alle opløsninger.

MoveWindow er en anden funktionalitet, der viser endnu en finurlighed Windows fremmaner:

```
private void MoveWindow(object sender, MouseButtonEventArgs e)
{
    if (e.LeftButton == MouseButtonState.Pressed && e.ClickCount == 1)
    {
        if (Maximized) //Places the middle of the titlebar at the
            mouse and reverts to previous window size
        {
            Point Position = Mouse.GetPosition(TitleBar);

            ExpandApplication(sender, e);
            Left = Position.X - 7 - Width / 2; //Windows randomly
                moves the window 7 pixels to the right
            Top = Position.Y - 9;
        }
    }
}
```

```

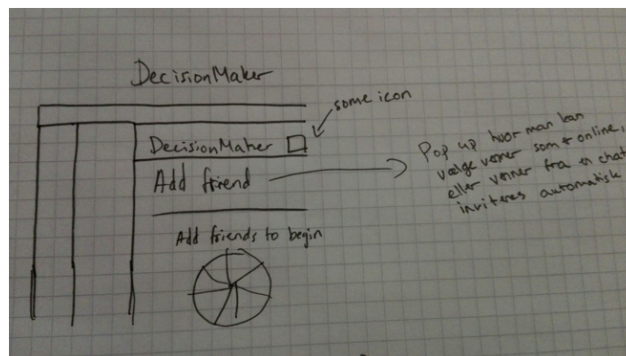
    }
}
if (e.ChangedButton == MouseButton.Left)
{
    DragMove();
}
}

```

Her bliver midten af vinduet automatisk sat ved musens placering, hvis vinduet trækkes fra maksimal størrelse. Windows flytter automatisk vinduet syv pixels til højre, hvilket gøres op for når vinduets "Left" position sættes. De ni der trækkes fra "Top" er blot halvdelen af titellinjens tykkelse.

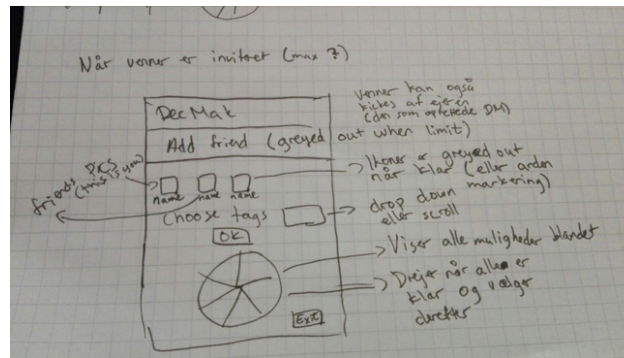
4.1.7 DecisionMaker

DecisionMaker (herefter DM) er en del af kravene, men denne del er blevet nedprioriteret og set som en tilføjelse. Idéen bag ved DM er at en række brugere kan vælge en række tags, hvormed et spil vil starte. Spillet vil bestå af et lykkehjul som tilfældigt vælger et af de tags, som brugerne har valgt. Et eksempel på hvordan dette kunne se ud ses på figur 56. Her kan brugeren vælge en række venner ved at klikke på en knap, eller de vil automatisk blive tilføjet hvis DM åbnes fra en gruppechat (som også er en mulig udvidelse). Et lykkehjul vil blive anvendt til at vise resultatet.



Figur 56: DecisionMaker - idé til layout

På figur 57 ses et eksempel på hvordan det kan se ud når venner tilføjes til DM. En knap "Add friend" kunne tilføjes, hvor venner kan tilføjes herfra. Denne knap kunne blive deaktiveret hvis et antal af venner er tilføjet. Vennernes billeder/ikoner vises. Tags kan vælges ved en drop down eller en scroll bar menu. Lykkehjulet viser alle mulighederne blandet sammen og drejer når alle brugere har trykket "OK" eller "Ready". En sidste knap, "Exit" tilføjes for at lade brugere gå ud af spillet.



Figur 57: DecisionMaker - idé til layout

4.1.8 Klient

4.1.8.1 Klient Klassen

Klassen af klienten er vigtig for at vise at den faktisk findes. Den har gennemgået nogle overvejelser som så fik denne her facon.

Klassenavn: Client

Metoder:

SslTcpClient()

ValidateServerCertificate(object sender, x509Certificate certificate, X509Chain chain

SslPolicyErrors sslPolicyErrors) : bool

RunClient()

SendMessage(string message)

ReceiveMessages()

Parametre:

static receiver : Receiver

static sender : Sender

static sslStream : SslStream

_eventAggregator : IEventAggregator

_userData : MyData

static certificateErrors : Hashtable

UserData(_userData = MyData.GetInstance) : MyData

Returværdi: Ingen

Beskrivelse: Klienten skal holde styr på de forskellige beskeder der kommer fra brugeren og sende dem direkte til serveren. Den skal også håndtere beskeder den modtager fra serveren og give besked videre til brugeren

4.1.8.2 Klient Implementering

Klienten skal sammensættes med GUI, og det gør det ved at bruge de klasser der er allerede sat fra GUI siden. Hovedsageligt håndtere klient klassen kommunikationen mellem server og klient/brugeren, dvs ikke hvad der sker, men forbindelsen bliver opretholdt og de korrekte oplysninger bliver sendt det korrekte sted.

```
public SslTcpClient()
{
    //Get certain event from GUI
    _eventAggregator = GetEventAggregator.Get();
    _eventAggregator.GetEvent<SendMessageToServerEvent>().Subscribe(SendMessage);

    string machineName = "192.168.101.1"; //IP Address of the server
    string serverCertificateName = "Martin-MSI"; //Name of the server to connect to

    SslTcpClient.RunClient(machineName, serverCertificateName);
}
```

Figur 58: Konstruktøren der håndtere Klientens login informationer

Ved figur 58 kan der ses at der bliver sat IP adressen og hvilket certifikat der skal være på for at den kan tilslutte til serveren. Dette bliver sat statisk i konstruktøreren pga. vi ejer ikke noget domæne eller DHCP server der skifter vores server IP.

```
public void ReceiveMessages()
{
    while (true)
    {
        string tempString = Receiver.ReceiveString(sslStream); //Blocking read
        string[] tempStringList = tempString.Split(Constants.GroupDelimiter); //Parsing

        switch (tempStringList[0])
        {
            case Constants.MessageReceived:
                var message = new ChatModel();
                message.Message = tempStringList[3];
                message.Sender = tempStringList[1];
                message.Receiver = tempStringList[2];

                Application.Current.Dispatcher.Invoke(() =>
                {
                    _eventAggregator.GetEvent<ReceiveMessageFromServerEvent>().Publish(message);
                });
                break;
            case Constants.FriendRequestReceived:
                ...
        }
    }
}
```

Figur 59: Handleren der håndtere Klientens modtagelses funktion

Ved figur 59 kan der ses at vi modtager noget. Selve serveren afgiver som kan ses i afsnit 5.2 hvor serveren bliver uddybt, nogle specifikke delimiters som håndteres forskelligt.

```
public static bool ValidateServerCertificate(  
    object sender,  
    X509Certificate certificate,  
    X509Chain chain,  
    SslPolicyErrors sslPolicyErrors)  
{  
  
    if (sslPolicyErrors == SslPolicyErrors.None)  
        return true;  
  
    Console.WriteLine("Certificate error: {0}", sslPolicyErrors);  
  
    //Since Marto does not have a certificate from a Certificate Authority, the server authentication will always fail.  
    //If a real certificate is used, the following line should return false  
    return true;    //Ignore false certificate. Used because of selfsigned certificate  
}
```

Figur 60: Validering til SSL angående korrekt certifikat

Ved figur 60 bliver selve TLS stream åbnet hvis den har den korrekte certifikat. Selve certifikatet bliver påført her, og hvis certifikatet er ok, så kan den sende. Siden vores system ikke har noget offentligt domæne så kan der kun komme et selfsigned certifikat på serveren, derfor bliver den faktisk ikke brugt, andet end at vise at den er klar til at udføre dette stykke arbejde.

4.2 Server

4.2.1 Indledning

Det er forudsat at vi som ingeniører skal indlede en analyse af de teknologier vi vil bruge for at danne et godt produkt.

Analysen foregår således at vi, med de respektive fag, danner en basis teori om de protokoller og løsninger der kan bruges helt bestemt til vores produkt.

4.2.1.1 Analyse

4.2.1.2 Data kryptering

Transmission Control Protocol (TCP) eller User Datagram Protocol (UDP)

Med vores krav om beskeder og tests, har vi kommet frem til at TCP vil blive brugt på vores server og klient når der skal sendes eller modtages beskeder. Dette skyldes at disse beskeder skal være fejlfri og komme i korrekt rækkefølge. Da der ikke er lavet noget krav om at der ikke må være en form for delay for at modtage beskederne, er TCP også acceptabelt selvom denne protokol kan være langsommere end UDP.

Vi har dog på basis af vores teori, undersøgt at streaming tjenester bør benytte UDP da pakkerne skal helst komme hurtigst muligt og der er ingen grund til at brugeren sender noget tilbage for at modtage den næste pakke. På samme tid er det også vigtigt at få de nye pakker her, fremfor at få outdatede pakker med lyd- eller billedinformation.

4.2.1.2.1 TLS Stream vs TCP stream

TLS protokollen (tidligere SSL) er en protokol med det formål at sikre krypteret kommunikation mellem en server og en klient. Udover denne krypterede kommunikation kan protokollen også bruges til identifikation så en klient eller server kan identificere om modtageren er korrekt.

Identificering: En klient der skal forbinde sig til en server vil gerne være sikker på at serveren er den korrekte server. Dette er for at undgå at en server udgiver sig for at være en anden, for at modtage hemmelig information, som for eksempel login oplysninger. Derfor kræver Marto klienter at en Marto server har det korrekte SSL certifikat. Har en server ikke det, vil Marto klienten nægte kommunikationen, og en forkert server vil derfor ikke kunne modtage en brugers oplysninger.

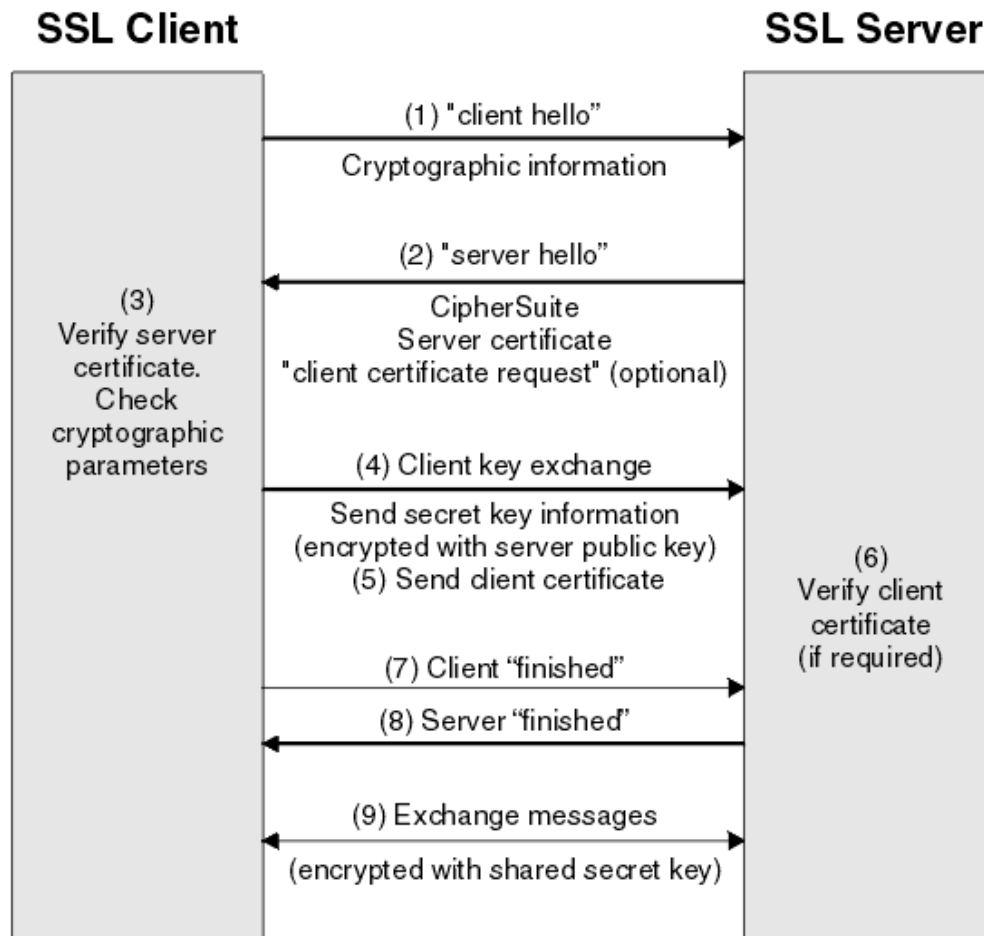
Krypteret kommunikation: Udover identificering har TLS protokollen en endnu større sikkerhedsforanstaltning, nemlig krypteret kommunikation. Dette sikrer at data der sendes mellem klient og server vil være scramblet, så selvom kommunikationen bliver sniffet (af et program såsom Wireshark), vil dataene ikke være forståelige.

TLS handshaking: For at muliggøre identificering og kryptering er der brug for en handshaking sekvens, blandt andet for at aftale krypteringsteknologi mellem server og klient. Som det kan ses på figur 61 herunder, består TLS/SSL handshaking af 8 trin (6 hvis klientens certifikat ikke skal checkes). I Marto checkes kun serverens certifikat, så her bruges kun 6 af trinene.

- 1. Klienten sender oplysninger om hvilke kryptografiske teknologier den supporterer.
- 2. Serveren sender oplysninger om hvilken teknologi den har valgt, samt serverens certifikat.
- 3. Klienten verificerer serverens certifikat. Er certifikatet forkert afbrydes forbindelsen.
- 4. Klienten sender data der bruges til fremtidig kryptering (private key). Dette data er krypteret med serverens public key.
- 7. Klienten sender en besked der er krypteret ved hjælp af private key. Denne viser at klienten er færdig med handshaking.
- 8. Serveren sender en besked der er krypteret ved hjælp af private key. Denne viser at serveren er færdig med handshaking.

Herefter kan klient og server sende beskeder der er krypteret med private key. Disse beskeder er ulæselige for alle der ikke har private key. Trinene der er beskrevet ovenfor er simplificerede, og kommer fra IBM[3].

Det samme gør figur 61.



Figur 61: SSL Handshake sekvens ¹

¹Billede fra IBM

TLS vs TCP: Kommunikationen mellem en Marto server og en Marto klient foregår over TCP forbindelser. Ovenpå TCP forbindelserne er TSL protokollen. For at demonstrere forskellen på data sendt over en almindelig TCP forbindelse, og en TCP+TLS forbindelse, er to testprogrammer udviklet. I testprogrammerne sender en klient et loginforsøg til serveren i formen: "l;UserName;PlaintextPassword<EOF". Kommunikationen er blevet sniffet ved hjælp af Wireshark programmet, og er derefter blevet sammenlignet. På Figur 62 ses den opsniffede packet som indeholder login forsøget fra klienten over en almindelig TCP forbindelse. Bemærk hvor letlæselig den sendte data er at læse (i den nederste del af figuren). Hvis en person får fat i denne packet vil en brugers login oplysninger ikke længere være hemmelige.

▷ Frame 4: 73 bytes on wire (584 bits), 73 bytes captured (584 bits)			
Raw packet data			
▷ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.1			
▷ Transmission Control Protocol, Src Port: 61040, Dst Port: 443, Seq: 1, Ack: 1, Len: 33			
▷ Data (33 bytes)			
0000	45 00 00 49 5e 2c 40 00	80 06 00 00 c0 a8 38 01	E..I^, @.8.
0010	c0 a8 38 01 ee 70 01 bb	a7 df f4 63 f7 5b d9 48	..8..p.. ...c.[.H
0020	50 18 01 00 08 2e 00 00	6c 3b 55 73 65 72 4e 61	P..... l;UserNa
0030	6d 65 3b 50 6c 61 69 6e	74 65 78 74 50 61 73 73	me;Plain textPass
0040	77 6f 72 64 3c 45 4f 46	3e	word<EOF >

Figur 62: Wireshark sniff af data pakke

På Figur 63 ses også en opsniffet packet med login oplysninger, men denne gang over en TCP + TLS forbindelse. De samme oplysninger er blevet sendt i dette testprogram som i det tidligere, men denne gang er oplysningerne krypterede. Bemærk at intet data data i bunden af figuren er læseligt.

▷ Frame 14: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits)			
Raw packet data			
▷ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.1			
▷ Transmission Control Protocol, Src Port: 60952, Dst Port: 443, Seq: 309, Ack: 1227, Len: 106			
▷ Secure Sockets Layer			
0000	45 00 00 92 5e 0f 40 00	80 06 00 00 c0 a8 38 01	E...^.@.8.
0010	c0 a8 38 01 ee 18 01 bb	65 bb 7d 17 e4 56 da c8	..8..... e.}.V..
0020	50 18 00 fb 7f 28 00 00	17 03 01 00 20 3c 52 16	P....(..<R.
0030	d4 16 76 eb 95 47 e8 d9	7f 19 04 d5 49 a5 26 19	..v..G..I.&
0040	0a 6b d6 87 26 4b 1d 35	9e dc a7 e1 67 17 03 01	.k..&K.5g...
0050	00 40 6d d6 d5 22 b2 a5	4b 82 31 04 72 39 63 0a	..@m..." K.1.r9c.
0060	c7 14 64 27 6d 42 a8 c8	6c 77 fc e5 6d e9 ee 5f	..d'mB.. lw..m.._
0070	a7 99 fc cd f7 24 7a 88	e7 e3 2d 42 94 64 8c d5\$z. ...-B.d..
0080	73 69 77 d2 69 75 ab a7	e7 5c 7b 61 95 18 b2 54	siw.iu.. .\{a...T
0090	4a 24		J\$

Figur 63: Wireshark sniff af data pakke

4.2.1.3 Hashing af adgangskoder

Da Marto brugernes login-oplysninger gemmes på en database på serveren, har gruppen bedømt det vigtigt at kryptere disse data. En brugers adgangskode gemmes derfor ikke i plaintext, da enhver med legal eller illegal adgang til databasen vil kunne logge ind som andre brugere, eller forsøge at logge ind med brugernavnet og adgangskoden på andre services. Adgangskoderne gemmes i stedet i hashet format. Til dette bruges algoritmen SHA256, også kaldet SHA-2, som er en forkortelse af Secure Hash Algorithm 2. Når en ny bruger laves, sættes password ind som input, og en 32 karakter lang string kommer ud som output. Dette output gemmes i databasen, og kan ikke laves om til en plaintext adgangskode igen.

For at hæve sikkerheden på de gemte login-oplysninger bruges der endnu en krypteringmetode kaldet Salting. Denne bruges til at besværliggøre en crackingmetode der bruger noget kaldet Rainbowtables ². Den type salt der bruges i Marto består af en string af 32 tilfældige karaktere der genereres hver gang en ny bruger tilføjes systemet. Plaintext adgangskoden sammen med saltet køres igennem SHA256 og det hashede output samt saltet gemmes i databasen sammen med brugernavn.

En ondsindet person kan derfor ikke sniffe brugeres data når de bliver sendt til og fra server, og kan heller ikke nemt finde en brugers plaintext adgangskode efter et database angreb på grund af de saltede og hashede adgangskoder.

4.2.1.4 Skalerbarhed

Da Marto er et chatprogram der ikke har et predefineret antal brugere, er systemets skalerbarhed relevant. I takt med at nye brugere tilslutter sig systemet er der brug for at gemme mere data i databasen. Og med flere brugere er der større chance for at mange klienter er online på samme tid. Dette kræver at serveren er skalerbar, så denne ikke crasher når mange brugere er på.

4.2.1.5 Async sockets Vs Multithreading

Der er mange måder at sikre skalerbarhed på, men den del som gruppen har fokuseret på har været indenfor internet sockets delen. Her har der været meget fokus på om der skulle startes nye tråde for hver klient, eller om der skulle startes en ny socket for hver klient, som så blev håndteret ved hjælp af async sockets. Med async sockets bruges callback metoder der kaldes ved hjælp af asynkrone metoder. For hver BeginX metode kald defineres en callback metode der skal kaldes senere. Hvis BeginReceive() kaldes med ReceiveCallback() som callback metode, vil denne kaldes når noget data er modtaget. Async sockets vil ikke blokere, og vil selv håndtere hvilke tråde der skal bruges for at sende og modtage data.

En multithreaded server kan derimod implementeres således at en ny tråd bliver lavet hver gang en klient forbinder. Disse tråde kan så stå i blokerende read på en socket eller stream. Da de forskellige klienter kører på hver deres tråde vil et blokerende read ikke skabe problemer, men der kan forekomme meget context switching. Dette tager tid og ressourcer. Med multithreading vil hver

²<http://keatas.kuliukas.com/RainbowTables/>

tråd også altid optage plads i RAM. Denne context switching sammen med muligt højt memory forbrug, kan hindre serverens scalability.

Selvom skalerbarheden kan være hindret af denne beslutning, har gruppen valgt at bruge den multithreaded server løsning da denne er dømt god nok og lettere at implementere.

4.2.2 Design

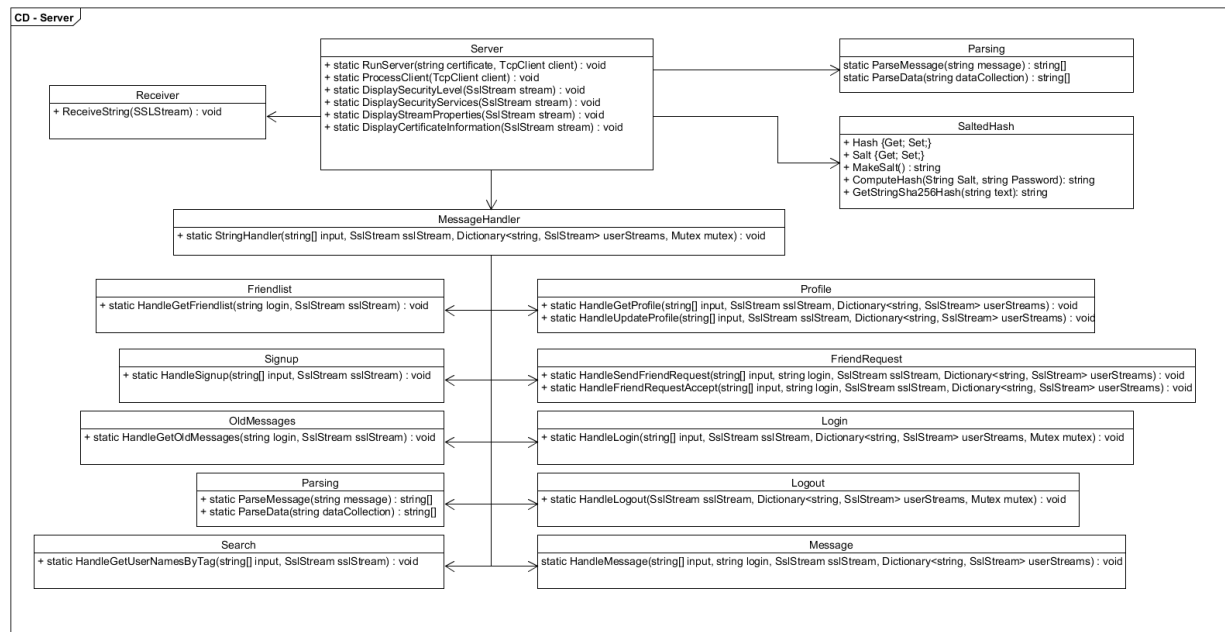
4.2.2.1 Server til klient protokol

Tabel 4 viser opbygningen af de strenge der sendes frem og tilbage mellem klient og server. Strengene er semi kolon separeret, hvor den første del af strengen er typen af request og det resterende er data.

Funktion	Protokol
Signup	S;Username;password
Login	L;Username;password
Logout	Q
Send message	W;DestinationUsername;Message
Receive message	R;Sender;Receiver;Message
Request profile	RP;Username
Profile response	RP;Username;Description;Tag1:Tag2(...)
Update profile	U;Description;Tag1:Tag2(...)
Search users by a tag	GUBT;Tag
Search users by a tag response	GUBT;User1:User2(...)
Send friend request	SFR;NewFriend
Receive friend request	FRR;Sender
Accept friend request	AFR;NewFriend
Get old messages	GOM
Request friendlist	RFL
Receive friendlist	RFL;User1:User2(..)

Tabel 4: Tabel over server-klient protokol

MessageHandleren kalder en af de område specifikke klasser, på baggrund af protokol strengen den modtager for klienten. Den første del af strengen, som det ses i tabel 4, bestemmer hvilken klasse metode der kaldes. De efterfølgende dele af strengen, indeholder information relevant til det request lavet af klienten, defineret af den første del af strengen.



Figur 64: Klassediagram over server

4.2.3 Klasse Beskrivelser

4.2.3.1 Server

Server klassens ansvar er at håndtere Server-Klient forbindelsen, samt håndtere den data der bliver modtaget ved at viderekalde besked handlers.

Metoder:

RunServer: void

Parametre: certificate: string, client: TcpClient

Retur værdi: Ingen

Beskrivelse: Metoden laver et nyt X509Certificate baseret på filstien der er defineret i certificate. ProcessClient metoden kaldes herefter med client.

ProcessClient: void

Parametre: client: TcpClient

Retur værdi: Ingen

Beskrivelse: Metoden laver en SslStream ud fra client og udfører TLS handshake sekvens med klienten. Herefter kaldes DisplayX metoderne. I en uendelig løkke kaldes Receiver.ReceiveString. Hvis SslStream mister forbindelse til klient lukkes denne forbindelse, brugeren logges af, og metoden returnerer.

DisplaySecurityLevel: void

Parametre: stream: SslStream

Retur værdi: Ingen

Beskrivelse: Metoden udskriver stream's sikkerhedsniveau. Disse er: CipherAlgorithm, CipherStrength, HashAlgorithm, HashStrength, KeyExchangeAlgorithm, KeyExchangeStrength, SslProtocol.

DisplaySecurityServices: void

Parametre: stream: SslStream

Retur værdi: Ingen

Beskrivelse: Metoden udskriver stream's sikkerhedsservices. Disse er: IsAuthenticated, IsServer, IsSigned, IsEncrypted.

DisplayStreamProperties: void

Parametre: stream: SslStream

Retur værdi: Ingen

Beskrivelse: Metoden udskriver stream's egenskaber. Disse er: CanRead, CanWrite, CanTimeout

DisplayCertificateInformation: void

Parametre: stream: SslStream

Retur værdi: Ingen

Beskrivelse: Metoden udskriver information om stream's certifikat. Disse er: Subject, GetEffectiveDate, GetExpirationDate.

4.2.3.2 Parsing

Parsing klassens ansvar er at parser beskeder baseret på nogle delimiters.

Metoder:

ParseMessage: string[]

Parametre: message: string

Retur værdi: String array med de forskellige datagrupper

Beskrivelse: Metoden parser message hvor GroupDelimiter forefindes.

ParseData: string[]

Parametre: dataCollection: string

Retur værdi: String array med de forskellige data

Beskrivelse: Metoden parser message hvor DataDelimiter forefindes.

4.2.3.3 SaltedHash

Klassens ansvar er at kryptere adgangskoder.

Metoder:

MakeSalt: string

Parametre: Ingen

Retur værdi: String med 32 karakter langt salt

Beskrivelse: Metoden genererer en tilfældig string med 32 karakterer.

ComputeHash: string

Parametre: salt: string, password: string

Retur værdi: String med 32 karakter langt hashet adgangskode

Beskrivelse: Metoden udregner et hash med SHA256 baseret på salt og password.

GetStringSha256Hash: string

Parametre: text: string

Retur værdi: string med 32 karakter hash

Beskrivelse: Metoden udregner et hash med SHA256 baseret på text.

4.2.3.4 MessageHandler

Klassens ansvar er at håndtere de beskeder der kommer fra klient.

Metoder:

StringHandler: void

Parametre: input: string[], sslStream SslStream, userStreams: Dictionary<string, SslStream>, mutex: Mutex

Retur værdi: Ingen

Beskrivelse: Metoden kalder de korrekte handler metoder baseret på Marto Server-Klient protokollen. Baseret på input[0] kaldes handler metoderne. Der er kun adgang til nogle af handle metoderne hvis en klient er logget ind og kan findes i userStreams.

4.2.3.5 Friendlist

Klassens ansvar er at håndtere de beskeder der omhandler Friendlists.

Metoder:

HandleGetFriendlist: void

Parametre: login: string, sslStream SslStream

Retur værdi: Ingen

Beskrivelse: Metoden kalder ProfileConsole's GetFriendList metode med login som parameter. Fra denne metode kommer alle brugerens venner, og en besked bliver lavet ud fra disse i formatet der er defineret under Receive friendlist i tabel 4 på side 82. Dette sendes derefter til klient ved brug af Sender's SendString metode.

4.2.3.6 FriendRequest

Klassens ansvar er at håndtere de beskeder der omhandler friend requests.

Metoder:

HandleSendFriendRequest: void

Parametre: input: string[], login: string, sslStream SslStream, userStreams: Dictionary<string, SslStream>

Retur værdi: Ingen

Beskrivelse: Metoden kalder ProfileConsole's AddFriendRequest metode med login og input[1]. Hvis den nye ven er online, sendes en besked med formatet der er defineret under Receive friend request i tabel 4 på side 82, ved brug af Sender's SendString metode.

HandleSendFriendRequestAccept: void

Parametre: input: string[], login: string, sslStream SslStream, userStreams: Dictionary<string, SslStream>

Retur værdi: Ingen

Beskrivelse: Metoden kalder ProfileConsole's AcceptRequest metode med login og input[1]. Hvis den nye ven er online, sendes en besked med formatet der er defineret under Accept friend request i tabel 4 på side 82, ved brug af Sender's SendString metode.

4.2.3.7 Login

Klassens ansvar er at håndtere de beskeder der omhandler login.

Metoder:

HandleLogin: void

Parametre: input: string[], sslStream SslStream, userStreams: Dictionary<string, SslStream>, mutex: Mutex

Retur værdi: Ingen

Beskrivelse: Metoden tjekker om en bruger med navnet i input[1] er online. Hvis brugeren er online sendes "NOK" til klienten med Sender's SendString metode. Med ProfileConsole's RequestUsername tjekkes om brugernavnet findes i databasen, hvis dette er tilfældet sendes "NOK" til klient. Ellers findes brugerens Salt med ProfileConsole's GetSalt metode, og med SaltedHash' ComputeHash metode laves et hashet password. Denne sammenlignes med brugerens hashede password i databasen med Login metoden, og hvis dette er korrekt tilføjes brugernavnet med SslStream til userStreams. Derefter sendes "OK" til klient.

4.2.3.8 Logout

Klassens ansvar er at håndtere de beskeder der omhandler logout.

Metoder:

HandleLogout: void

Parametre: input: string[], sslStream SslStream, userStreams: Dictionary<string, SslStream>, mutex: Mutex

Retur værdi: Ingen

Beskrivelse: Metoden tjekker om en bruger med navnet i input[1] er logget ind. Hvis dette er

tilfældet fjernes brugernavnet og SslStream fra userStreams.

4.2.3.9 Message

Klassens ansvar er at håndtere de beskeder der omhandler beskeder.

Metoder:

HandleMessage: void

Parametre: input: string[], sslStream SslStream, userStreams: Dictionary<string, SslStream>, mutex: Mutex

Retur værdi: Ingen

Beskrivelse: Metoden tjekker afsenderen (login) og modtageren (input[1]) er venner. Hvis dette er tilfældet sendes en besked med formatet der er defineret under Receive message i tabel 4 på side 82. Dette sendes derefter til klient ved brug af Sender's SendString metode. Derefter gemmes beskeden i database med ProfileConsole's SaveIncomingMessage metode.

4.2.3.10 OldMessages

Klassens ansvar er at håndtere de beskeder der omhandler gamle beskeder.

Metoder:

HandleGetOldMessages: void

Parametre: login: string, sslStream SslStream

Retur værdi: Ingen

Beskrivelse: Metoden henter en brugers gamle beskeder fra databasen med ProfileConsole's RequestAllMsgs metoden. Hver af beskederne sendes en af gangen i formatet der er defineret under Receive message i tabel 4 på side 82, ved brug af Sender's SendString metode.

4.2.3.11 Profile

Klassens ansvar er at håndtere de beskeder der omhandler profiler.

Metoder:

HandleGetProfile: void

Parametre: input: string[], sslStream SslStream, userStreams: Dictionary<string, SslStream>

Retur værdi: Ingen

Beskrivelse: Metoden finder en brugers profil fra databasen med ProfileConsole's RequestOwnInformation metode. En besked med formatet der er defineret under Profile response i tabel 4 på side 82, sendes ved brug af Sender's SendString metode. Findes brugeren ikke sendes i stedet "RPNOK".

HandleUpdateProfile: void

Parametre: input: string[], sslStream SslStream, userStreams: Dictionary<string, SslStream>

Retur værdi: Ingen

Beskrivelse: Metoden parser input[3] (tags) med Parsing's ParseData metode. Brugers bruger-

navn findes i userStreams. Brugerens nye profiloplysninger gemmes derefter i databasen med ProfileConsole's UpdateProfileInformation metode.

4.2.3.12 Search

Klassens ansvar er at håndtere de beskeder der omhandler søgninger.

Metoder:

HandleGetUsernamesByTag: void

Parametre: input: string[], sslStream SslStream

Retur værdi: Ingen

Beskrivelse: Metoden finder brugere med et specifikt tag ved brug af ProfileConsole's RequestTag metode. En besked med formatet der er defineret under Search users by a tag response i tabel 4 på side 82, sendes ved brug af Sender's SendString metode.

4.2.3.13 Signup

Klassens ansvar er at håndtere de beskeder der omhandler signup.

Metoder:

HandleSignup: void

Parametre: input: string[], sslStream SslStream

Retur værdi: Ingen

Beskrivelse: Metoden laver en ny bruger hvis denne ikke allerede findes. Et nyt Salt laves med SaltedHash's MakeSalt metode. Et hashed password laves derefter med ComputeHash. Brugeren tilføjes til database med ProfileConsole's CreateProfile metode. Hvis dette lykkedes sendes "SOK" til klient. Lykkedes det ikke sendes "SOK".

4.2.3.14 Sender

Klassens ansvar er at sende beskeder over en SslStream.

Metoder:

SendString: void

Parametre: sslStream SslStream, message: string

Retur værdi: Ingen

Beskrivelse: Metoden påsætter EndDelimiter i slutningen af message. Derefter sendes alt dette over sslStream's stream.

4.2.3.15 Receive

Klassens ansvar er at sende beskeder over en SslStream.

Metoder:

ReceiveString: void

Parametre: sslStream SslStream

Retur værdi: Ingen

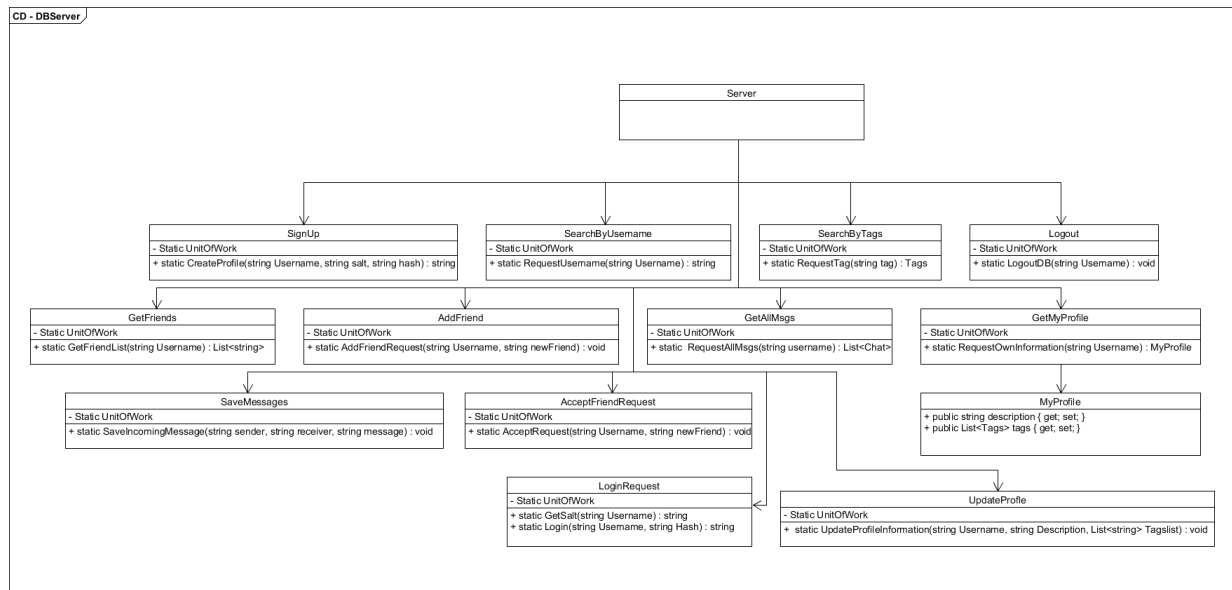
Beskrivelse: Metoden læser fra sslStream i en løkke indtil en EndDelimiter eller en tom tekst er modtaget. Herefter fjernes EndDelimiter fra beskeden, og beskeden returneres som string.

4.2.3.16 Constants

Klassens ansvar er at samle systemets konstanter i en fil. Disse konstanter er Delimiters og protokollernes start konstanter der kan ses i tabel 4 på side 82.

4.2.4 Kommunikation med DB

Til kommunikation med DB er der designet og implementeret en række klasser, hver med deres ansvarsområde. Klasserne er designet så de hver kun står for en ting, med få undtagelser, hvor en klasse kan have to funktionaliteter. Dette har gjort det nemt at teste og udvide, da der er få dependencies. Klasserne er navngivet så det er nemt at regne ud hvad deres funktionalitet er. Dette har overskueliggjort integrationen mellem internet kommunikationsdelen og databasen. Server klassen står tom for overskuelighedens skyld. Server defineres som klassediagrammet på figur 64.



Figur 65: Klassediagram over DB kommunikationen fra server

Alle klasserne benytter sig af databasens UnitOfWork og repositories, der gør det nemt at udføre CRUD operationer på databasen. På baggrund af hvad der sendes frem og tilbage mellem klient og server, kaldes der en metode fra en af de klasser, opstillet på figur 65.

4.2.5 Klassebeskrivelser

4.2.5.1 SignUp

Signup klassen står for at oprette en profil i systemet.

Metode

static CreateProfile()

Parametre: string Username, string salt, string hash

Returværdi: string

Beskrivelse: Createprofile står for at oprette en profil med de informationer der modtages fra klienten. Der tjekkes her om det brugernavn profilen ønskes at oprettes med, eksisterer i forvejen og returnerer her en fejlmeddelelse. Hvis brugernavnet er ledigt, vil en godkendt besked returneres.

4.2.5.2 SearchByTags

Klassens ansvar er at gøre det muligt at søge efter brugere med et specifikt tag.

Metode

static RequestTag()

Parametre: string tag

Returværdi: Tags

Beskrivelse: Funktionen modtager et tag navn i form af en string og kigger i databasen om det findes. Hvis det gør, kigges der på hvilke profiler, der har tagget. Disse profiler returneres i et Tags objekt. Tag objektet indeholder profilens userinformation og taggets navn.

4.2.5.3 Logout

Klassens ansvar er at gøre det muligt at gemme profilens online/offline tilstand i databasen.

Metode

static LogoutDB()

Parametre: string Username

Returværdi: void

Beskrivelse: Metoden modtager et brugernavn. Profilen med det brugernavn findes i databasen og profilens status sættes til offline.

4.2.5.4 GetFriends

Klassens ansvar er at returnere en profils venneliste

Metode

static GetFriendList()

Parametre: string Usernam

Returværdi: List<string>

Beskrivelse: Metoden modtager et brugernavn og profilen med tilsvarende brugernavn findes i databasen. Herefter tilgås profilens venneliste og profilens venne brugernavne smides i en liste der returneres.

4.2.5.5 AddFriend

Klassens ansvar er at gøre det muligt at sende en venneanmodning til en anden profil og gemme det i databasen.

Metode

static AddFiendRequest()

Parametre: string Username, string newFriend

Returværdi: Void

Beskrivelse: Metoden finder profilen med det brugernavn den modtager i parameteren. Der oprettes her et ny objekt af venneliste klassen og med de to funktionskald parametre. Venskabsstatussen sættes til Pending og personen der har foretaget handlingen sættes til profilens brugernavn.

4.2.5.6 GetAllMsgs

Klassen skal gøre det muligt at tilgå alle beskeder, sendt og modtaget, fra en profil.

Metode

static RequestAllMsgs()

Parametre: string Username

Returværdi: List<Chat>

Beskrivelse: Metoden finder profilen med det tilsvarende brugernavn. Herefter sorteres alle beskeder der er sendt eller modtaget med det tilsvarende brugernavn. Disse beskeder smides i en liste af Chat objekter, der returneres.

4.2.5.7 GetMyprofile

Klassen står for at returnere en profils informationer.

Metode

static RequestOwnInformation()

Parametre: string Username

Returværdi: MyProfile

Beskrivelse: Metoden finder profilen med det tilsvarende brugernavn. Profilens description og tagliste gemmes i et nyt objekt af MyProfile klassen, der hefter returneres.

4.2.5.8 MyProfile

Klassen bruges til at midlertidigt gemme profiloplysninger. Klassen består af to properties, en string og en taglist.

4.2.5.9 SaveMessages

Klassen står for at gemme beskeder der sendes mellem to klienter.

Metode

static SaveIncomingMessage

Parametre: string sender, string receiver, string message

Returværdi: Void

Beskrivelse: Opretter et nyt chat objekt og gemmer parameterne fra funktioneskaldet i objektet. Objektet gemmes i databasen med unitofwork.

4.2.5.10 AcceptFriendRequest

Klassen står for at acceptere de venneanmodninger der ligger i databasen.

Metode

static AcceptRequest()

Parametre: string Username, string newFriend

Returværdi: Void

Beskrivelse: Funktionen kigger efter en venneanmodning der indeholder de to parametre og sætter friendlist objektets status til Added.

4.2.5.11 LoginRequest

Klassen står for at logge en profile korrekt ind.

Metode

static GetSalt()

Parametre: string Username

Returværdi: string

Beskrivelse: Funktionen henter salt i databasen, fra den profil der stemmer overens med brugernavnet fra metodekaldet.

Metode

static Login()

Parametre: string Username, string Hash

Returværdi: string

Beskrivelse: Funktionen tjekker i databasen om de indtastede informationer stemmer overens med en profil i databasen. Hvis de gør, returneres en OK besked og profilens status sættes til Online. Hvis de ikke gør, returneres en fejlmeddelse.

4.2.5.12 Klasse: UpdateProfile

Klassens ansvar er at opdatere en profils informationer.

Metode

static UpdateProfileInformation()

Parametre: string Username, string Description, List<String> TagsList

Returværdi: void

Beskrivelse: Metoden finder den profil hvis brugernavn stemmer overens med parameteren i funktionskaldet. Her overskrives profilens description med den nye og profilens tags opdateres. Hvis de tags ikke eksisterer i databasen i forvejen, tilføjes de som nye Tags objekter først.

4.2.6 Implementering

Dette afsnit omhandler implementering af server delen. Der vil her fremhæves væsentlige dele af softwaren, med kodeudsnit og beskrivelser. Det resterende kode kan ses i det vedhæftede source kode.

Kodeudsnittet på figur 66 viser vores listener. Den har til ansvar at lytte efter nye klienter der tilkobles serveren. Når en ny klient tilkobles serveren, oprettes der en ny tråd, der startes. Denne tråd håndterer nu alt kommunikation fra den nyligt tilkoblede klient.

```
listener.Start();

//Keep listening and start new thread when client connects
while (true)
{
    TcpClient client = listener.AcceptTcpClient(); //Someone has connected

    Thread newThread =
        new Thread(
            unused => RunServer(certificate, client)    //The method where the thread is run
        );

    newThread.Start();
}
```

Figur 66: Kodeudsnit fra server.cs

Når en ny klient er tilsluttet og der er oprettet en tråd, skal der tilføjes sikkerhed til kommunikationen. Her oprettes der en ny SSLStream til tråden, så klient og server kan kommunikere med krypteret data.

```
static void ProcessClient(TcpClient client)
{
    // A client has connected. Create the
    // SslStream using the client's network stream.
    SslStream sslStream = new SslStream(
        client.GetStream(), false);
    // Authenticate the server but don't require the client to authenticate.
    try
    {
        sslStream.AuthenticateAsServer(serverCertificate,
            false, SslProtocols.Tls, true);
    }
}
```

Figur 67: Kodeudsnit fra server.cs

Figur 68 viser en klienttråd der venter på et end delimiter. Herefter kaldes messagehandleren, med

den besked der lige er modtaget.

```
while (true)
{
    string messageData;
    // Read a message from the client.
    Console.WriteLine("Waiting for client message...");
    try
    {
        messageData = Receiver.ReceiveString(sslStream);
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
        Server.Logout.HandleLogout(sslStream, userStreams, _mutex);
        return;
    }

    string[] parsedMessage = Parsing.ParseMessage(messageData);
    MessageHandler.StringHandler(parsedMessage, sslStream, userStreams, _mutex);
    Console.WriteLine();
}
```

Figur 68: Kodeudsnit fra Server.cs

Udsnittet på figur 69 viser et udsnit af den switch case messagehandleren er opbygget af. På baggrund af den konstant der modtages, kaldes en funktion til håndtering af den type request.

```
switch (input[0])
{
    case Constants.Write: //Write message
        Message.HandleMessage(input, userStreams.FirstOrDefault(x => x.Value == sslStream).Key,
            break;
    case Constants.RequestProfile: //Profile get
        Profile.HandleGetProfile(input, sslStream, userStreams);
        break;
    case Constants.UpdateProfile: //Update profile
        Profile.HandleUpdateProfile(input, sslStream, userStreams);
        break;
    case Constants.RequestLogin: //Login
        Console.WriteLine("User is already logged in");
        Sender.SendString(sslStream, "You are already logged in");
        break;
    case Constants.Logout: //Logout
        Logout.HandleLogout(sslStream, userStreams, mutex);
        break;
}
```

Figur 69: Kodeudsnit fra MessageHandler.cs

4.2.7 Unit Test

Til kommunikation med databasen er der lavet unit test på de forskellige klasser. Dette er foregået i en mappe i projektet, frem for et nyt projekt som man normalt ville gøre det. Dette skyldes at unit testene skal have adgang til databasen og det her var nemmest blot at oprette en mappe til tests. Testene er korte og simple og tester hoved funktionaliteten for klassefunktionerne. Da klasserne overholder single responsibility principle fra SOLID, er der ikke opstillet mange unit tests for hver klasse, da flere af klasserne kunne gennemtestes med et par unit tests. Testene har gjort det nemmere at implementere og senere integrere klasserne med resten af systemet. Vi kunne gennem testene vurdere om en klasse virker alene, hvis systemet hermed fejler under integrationen, kan vi vurdere at det er interfacet mellem de to moduler der fejler.

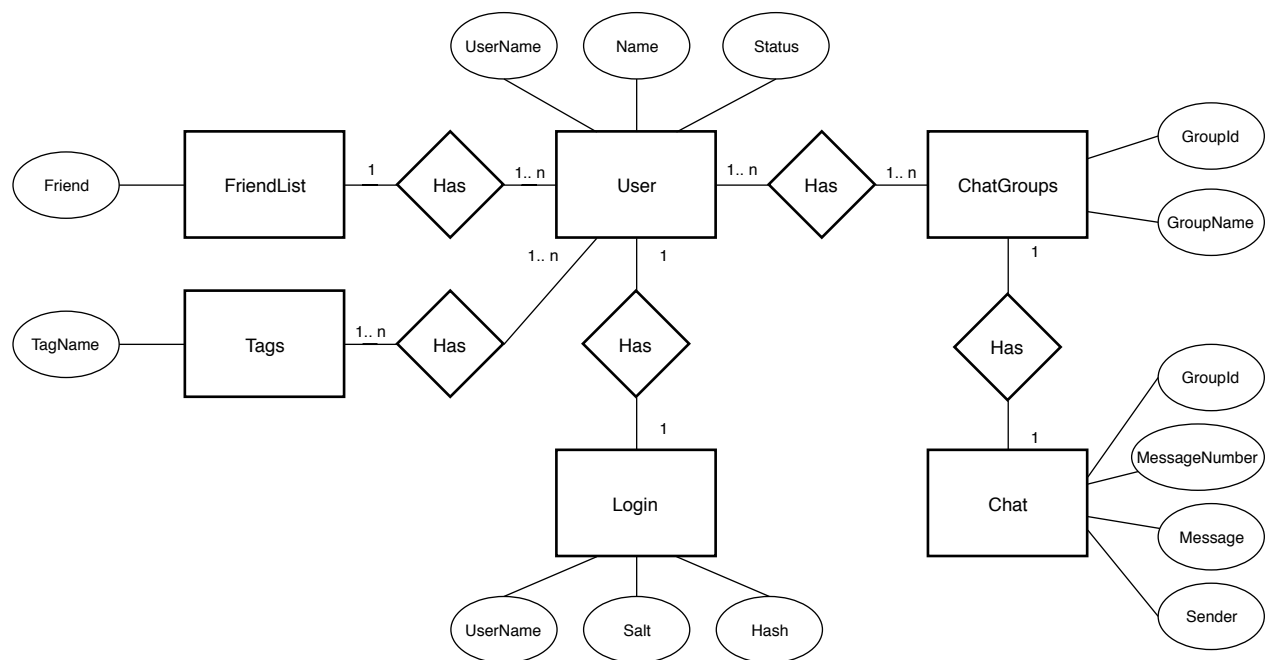
Testene er implementeret og udført med NUnit, der er et anerkendt test framework. Funktionerne i klasserne kaldes med realistiske parametre og der antages et resultat. Hvis resultatet af eksekveringen er som forudset, vurderes klassen funktionel. Testene er udført ved at kalde funktionerne under test, med parametre hvor vi er i stand til at forudse resultatet. Herefter bruger vi Assert() funktionen til at antage et resultat. De funktionerne der har en returværdi, testes selvfølgelig i forhold til dette og der antages at returværdien er korrekt. Der testes også med queries, om det der gemmes i databasen stemmer overens med det forventede resultat. Her har vi mulighed for at tilgå det data der ligger i databasen og Assert at et givent objekt indeholder det korrekte data.

4.3 Database

Databasen skal kunne holde styr på en masse forskellige informationer om Marto, samt alle brugere, der er oprettet på systemet. Hver bruger har flere entities, som skal også gemmes i databasen. Dette er informationer om deres brugernavn, tags, chatgrupper, beskeder osv. Der bruges her en relationel database, så hver bruger kan have relationer til de andre entities.

4.3.1 Analyse

For at få en bred forståelse for, hvordan databasen skal se ud, laves der et ERD for databasen. Dette diagram kan ses på figur 70. Diagrammet viser User som en central entitet, som er brugeren, der bruger systemet. Denne User har så relationer til andre entiteter, samt kan relationernes connectivity ses på diagrammet, som er den lille skrift ved siden af entiteterne. Denne connectivity beskriver om relationen mellem entiteterne er "en-til-en", "en-til-mange" eller "mange-til-mange".



Figur 70: ERD for databasen

Databasen laves som en relationel SQL database, da de forskellige tabeller i databasen er statiske, dvs. at de ændrer sig ikke, eller de ændrer sig sjældent. Grunden til at de er statiske er at alle brugere skal indeholde den samme information, samt skal alle chatgrupper indeholde samme information osv. Der er ikke nogen forskel på strukturen for disse oplysninger.

Hvis dette var et system der skulle udvikles og opdateres ofte, kunne det være smartere at lave det som en NoSQL database, da det f.eks. kunne være at brugerinformationen skal opdateres til at kunne indeholde flere oplysninger. Det kunne også være, at der i fremtiden skal kunne være en

større forskel på brugerne, med at brugerne skal indeholde forskellige oplysninger alt efter, hvilken type person det er.

Databasen er oprettet på en lokal server, dvs. at der kun er én bruger der har adgang til den adgang. Dette har medført at hvis en anden person skulle bruge databasen, har personen skulle oprette sin egen lokale database ud fra Code First Migrations. Derfor endte der med at ligge flere lokale databaser, som var helt ens rundt omkring forskellige computer, og hvis en database blev ændret skulle alle andre databaser også opdateres vha. migrations. Grunden til, at der blev valgt at have en lokal database var udelukkende pga. uvidenhed omkring ulemperne ved dette, og disse ulemper blev opdaget for sent.

I fremtiden vil der helt klart blive valgt at have en online server, som alle medlemmer skal kunne tilgå, hvor databasen skal være. På den måde skal der ikke deles ændringerne frem og tilbage mellem flere computere.

4.3.2 Design af databasen

Databasen bliver implementeret vha. DbContext, der har ansvaret for at oprette selve databasen. Denne DbContext har også ansvaret for at holde styr på alle tabeller inde i databasen. Alle tabellerne implementeres som hver sin klasse, som indeholder alle de kolonner, der skal være med i tabellen.

4.3.2.1 ChatGroups

ChatGroups er en tabel over alle chatgrupper, der er blevet oprettet på systemet. Et eksempel ses på figur 71

GroupId	GroupName
1	Marto-Entusiast
3	Ingeniørerne

Figur 71: Database over ChatGroups

4.3.2.2 Chat

Tabellen for chat er delt op i to dele. Der er både en chat mellem to brugere, og så er der en gruppechat.

Tabellen for chat mellem to brugere indeholder informationer om, hvem der er sender og modtager ved de forskellige beskeder, samt indeholder tabellen alle de beskeder der er skrevet mellem to brugere. Et eksempel på denne tabel kan ses på figur 72

Sender	Receiver	Message
Marto	Farto	Hej
Farto	Marto	Hva' så?

Figur 72: Database over Chat

Tabellen for gruppechatten indeholder informationer over alle beskeder, samt hvilken chatgruppe beskeden bliver skrevet i. Denne tabel indeholder også, hvem der er senderen på de enkelte beskeder. Der er ikke en kolonne til, hvem der er modtagere, da alle brugere, som er medlemmer af gruppechatten er modtagere. Et eksempel på denne tabel kan ses på figur 73.

GroupId	Message	Sender
1	Hej	Marto
1	Hva så?	Farto
1	Ikke så meget	Marto
3	Hey gutter	Marto

Figur 73: Database over Gruppechat

Pointen med at holde på alle beskeder i databasen, er at hver bruger har muligheden for at se alle sine tidligere beskeder, samt hvem der har skrevet hver enkel besked.

4.3.2.3 Emoji

Ethvert stort chatprogram nu til dags har en liste af emojis, som viser forskellige udtryk. Som bruger er det rart at kunne bruge disse emojis, da de normalt siger mere end ord. Derfor bliver der lavet en database som holder styr på alle de tilgængelige emojis, som brugerne kan vælge imellem. Hver emoji har en shortcut, for meget hurtigt at kunne bruge dem i en chatbesked. Et eksempel på emoji-databasen ses på figur 74

EmojiShortcut	Emoji
:)	😊
<-	👈
<3	💕

Figur 74: Database over Emojis

4.3.2.4 FriendList

Databasen til FriendList bruger en relation mellem to brugere. Derudover bruges der en kolonne, som viser hvilken status de to brugeres relation har. Der er fire forskellige statuser, som to brugere kan have.

- Pending - Dette vil sige at en bruger har ansøgt den anden bruger om at være venner, men den anden bruger har ikke svaret endnu.
- Accepted - Dette vil sige, at de to brugere nu er venner.
- Declined - Dette betyder, at den ene bruger har afslået at være venner med den anden bruger.
- Blocked - Dette betyder, at den ene bruger har blokeret den anden bruger, som medfører at den ene bruger aldrig kan skrive eller sende en anmodning til den anden bruger, indtil den anden bruger klikker på "unblock"
- Removed - Hvis to brugere er venner, er det muligt for den ene bruger at slette den anden bruger fra sin venneliste. Hvis dette sker, skiftes deres status til "Removed".

Den sidste kolonne hedder "aktion_bruger_id", og den viser, hvilken af de to brugere der sidst har lavet en aktion, f.eks. hvis bruger 1 har sendt en venneanmodning til bruger 2, og bruger 2 ikke har svaret endnu, vil aktion_bruger_id være 1, da det var bruger 1 der sidst lavede en aktion. Hvis bruger 2 så trykker på accept, vil det nu være bruger 2 der har lavet den sidste aktion, og derfor bliver aktion_bruger_id erstattet med 2. Et eksempel kan ses på tabel 75

User1	User2	Status	Action_User
Farto	Marto	Accepted	Marto
Darto	Marto	Declined	Darto
Darto	Farto	Blocked	Darto
Darto	Sarto	Pending	Sarto

Figur 75: Database over Friendlist

4.3.2.5 Login

Login er en tabel over alle UserNames, samt deres "saltede og hashede" adgangskode. Grunden til at brugernes rigtige adgangskode ikke gemmes på databasen er for at øge sikkerheden. På denne måde, hvis nogen får fat i databasen, kan de ikke logge ind på andre profiler. Når en bruger logger ind på Marto, vil deres kode blive tilføjet salt og hash, og så vil der på databasen blive tjekket om den indeholdte salt og hash passer sammen med den adgangskode brugeren skriver for at logge ind. Et eksempel kan ses på tabel 76

Username	Salt	Hash
Marto	DDFERE134A	c412b37f8c0484
Farto	IUHJKA331	362132fc05ba31

Figur 76: Database over Login

4.3.2.6 Tags

Tags er en tabel over alle de tags, der er tilføjet til systemet. Dette vil kunne komme til at indeholde alle mulige tekststrengene. Hver gang en bruger tilføjer et tag til sin profil, vil det blive gemt i denne tabel. Et eksempel kan ses på figur 77

TagName
Fodbold
Tennis
Databaser
CSGO

Figur 77: Database over Tags

4.3.2.7 UserInformation

UserInformation er en tabel over alle brugernavne, samt deres beskrivelse, som er en tekst, brugerne kan skrive om sig selv på sin profil. Her kan også ses om en bruger er online eller offline på systemet. Et eksempel ses på figur 78

UserName	Description	Status
Marto	...	Online
Farto	...	Offline
Darto	...	Online
Sarto	...	Online

Figur 78: Database over UserInformation

4.3.2.8 UserChatGroups

En bruger kan være medlem af flere chatgrupper, samt kan der en chatgruppe være flere brugere. Her er der tale om en many-to-many relation. Derfor skal der laves en ekstra tabel til at holde styr på dette. Denne tabel kaldes UserChatGroups, og har en kolonne til en bestemt chatGruppe,

samt en kolonne til et bestemt brugernavn. På denne måde kan alle relationer mellem brugere og chatGrupper ses. Et eksempel på dette kan ses på tabel 79.

GroupId	UserName
1	Marto
1	Farto
3	Marto
3	Darto

Figur 79: Database over UserChatGroups

4.3.2.9 UserTags

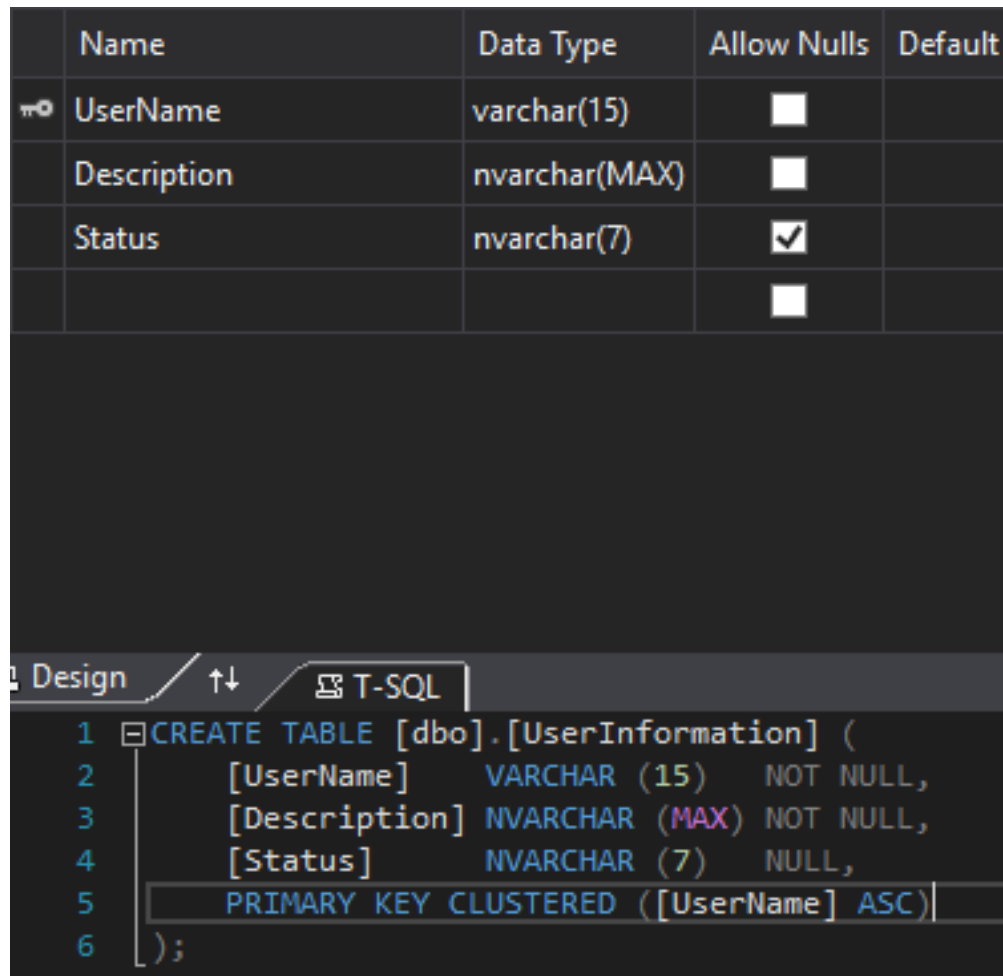
Der er også en many-to-many relation mellem en bruger og et tag. Derfor laves der også en tabel til at holde styr på relationerne mellem disse. Denne tabel kan ses på figur 80

TagName	UserName
CSGO	Marto
CSGO	Darto
Databaser	Marto
Tennis	Sarto

Figur 80: Database over UserTags

4.3.3 Implementering af databasen

Oprettelsen af databasen og tabellerne blev lavet manuelt vha. SQL Design, som ses på figur 81. Der blev herefter brugt ADO.NET til at få oprettet klasser til tabellerne, samt en DBContext.



Figur 81: Oprettelse af tabel for Userinformation

ADO.NET virker som en omvendt "Code First", hvor at det her er databasen der bliver lavet først. Det som ADO.NET gør, er at det går ind i databasen og kigger alle tabeller og andre informationer igennem, så den ved præcist, hvilke egenskaber databasen har. Derefter bliver der lavet en klasse for alle tabeller, der indeholder alle colonner og andet information i tabellen. Der bliver her også oprettet en DBContext, som indeholder alle disse klasser. Den oprettede DBContext kan ses på figur 82. Det kan her ses at der oprettes en DbSet til hver klasse. DbSet sørger for at der er en connection til databasens tabeller.


```
public ProfileContext()
{
    : base("name=ProfileContext")
}

2 references | Soeren Bech, 18 days ago | 1 author, 1 change
public virtual DbSet<ChatGroups> ChatGroups { get; set; }
0 references | Soeren Bech, 18 days ago | 1 author, 1 change
public virtual DbSet<Emoji> Emoji { get; set; }
0 references | Soeren Bech, 18 days ago | 1 author, 1 change
public virtual DbSet<FriendList> FriendList { get; set; }
0 references | Soeren Bech, 18 days ago | 1 author, 1 change
public virtual DbSet<Login> Login { get; set; }
1 reference | Soeren Bech, 18 days ago | 1 author, 1 change
public virtual DbSet<Tags> Tags { get; set; }
10 references | Soeren Bech, 18 days ago | 1 author, 1 change
public virtual DbSet<UserInformation> UserInformation { get; set; }
1 reference | Soeren Bech, 18 days ago | 1 author, 1 change
public virtual DbSet<Chat> Chat { get; set; }
```

Figur 82: Kodeudsnit af DBContext

Når der bliver kørt ADO.NET, bliver der også oprettet en connectionString. Dette sørger for at der er en connection til databasen. Koden til ConnectionsString kan ses på figur 83. På første linje i koden ses at der står name="ProfileContext". Dette navn er det samme, der står i DBContext ved constructoren, som kan ses på figur 82.

Det næste der sker er at der bliver oprettet en connection til den lokale server som hedder (localdb)\MSSQLLocalDB.

Derefter bliver der kigget i serveren efter en bestemt database. I dette tilfælde er det databasen, som hedder "ProfileDatabase", og der bliver connected til denne database.

```
<connectionStrings>
  <add name="ProfileContext" connectionString="data source=(localdb)\MSSQLLocalDB;
  initial catalog=ProfileDatabase;
  integrated security=True;MultipleActiveResultSets=True;
  App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Figur 83: Kodeudsnit af med connectionString

4.3.3.1 Repository

For at kunne opdatere tabellerne, altså tilføje, slette nogle informationer osv. bliver der lavet et repository til databasen.

Repository er et pattern, der opfører sig som en "in-memory" kollektion af domæne objekter. Nogle fordele med repository er:

- At minimere antallet af duplikerede queries, ved at inkapsle disse queries i nogle metoder.
- At afkoble applikationen fra persistence frameworks, hvilket betyder at hvis applikationen på et tidspunkt bliver skiftet til et andet persistence framework, vil der være en minimal ændring på applikationen. Dette sker fordi applikationen nu er afhængig af repositoret i stedet for at være direkte afhængig af framework.
- At gøre det nemmere at unit teste applikationen, da der bliver lavet interfaces, som kan bruges til tests.

Repository har metoder som: `add(obj)`, `remove(obj)`, `get(id)`, `getall()` og `find(predicate)`. Disse metoder er generiske og kan bruges af alle klasser. Udover disse, kan der også være brug for at oprette nogle individuelle metoder, som kun skal kaldes af en bestemt klasse. Derfor laves der også et repository til de klasser, som har deres egen metoder. Disse metoder er vigtige at bruge når man f.eks. har en many-to-many relationship mellem nogle klasser.

Til tabellen for `UserChatGroups` er der blevet oprettet en metode, der har det ansvar at returnere alle chatgrupper, en bruger er medlem af. Der er også oprettet en metode til at se hvilke brugere, der er medlem af en given chatgruppe.

Til tabellen for `UserTags` er der blevet oprettet en metode, der har det ansvar at returnere alle tags, en bruger har på sin profil. Der er også oprettet en metode til at se hvilke brugere, der har et givent tag på deres profil.

Der er til sidst også oprettet en metode til at returnere alle chatinformationer, en given chatgruppe indeholder.

4.3.3.2 Unit of Work

Når der bliver lavet en opdatering på databasen skal denne selvfølgelig gemmes. For at gøre dette bliver der brugt unit of work.

Unit of work har ansvaret for at gemme de ændringer der bliver lavet i repository, og opdatere databasen ud fra ændringerne. Unit of work er godt at bruge, hvis der er flere repositories i applikationen, da der er mange tilfælde, hvor flere repositories bliver kaldt lige efter hinanden, f.eks. hvis man vil tilføje en adresse, vil man først tilføje en person, som vil blive tilføjet en adresse. Her er der ingen grund til at gemme ændringer både når der bliver tilføjet en person, og når der bliver tilføjet en adresse. Her bruges i stedet Unit of work, så der kun bliver gemt, når alle de ønskede kommandoer er blevet fuldført

4.3.3.2.1 REST API

Da Unit of Work bruges til at tilføje, slette, opdatere og hente informationen i databasen, er der, efter overvejelse, valgt *ikke* at tilføje et REST API til databasen.

4.3.4 SQL Sanitation/Injection

Konceptet bag SQL Sanitation er at det ikke skal være muligt for brugeren at manipulere source koden ved at lave en injection. Dette kan bl.a. være et problem hvis SQL-databasen er koblet sammen med et program hvor der kan tilføjes til databasen via en tekstboks. Hvis brugeren skriver noget tekst i tekstboksen der udkommenterer den afsluttende "; vil det være muligt for brugeren at manipulere koden gennem den tekst der skrives i tekstboksen. Dette er blot ét eksempel på SQL injection. I de fleste tilfælde vil det være i en applikations login-vindue at muligheden for injection opstår.

SQL injection-angreb kan forekomme i alle SQL-koder hvor der er et brugerinput og databasekald.

Injection kan undgås på flere måder, én af dem er ved at afgrænse input. Dette kan f.eks. være ved at afgrænse hvad der kan skrives i et tekstfelt (ingen "/", "-", eller andre tegn der f.eks. kan anvendes til at udkommentere kode).

SQL Sanitation har også et princip inden for SQL-databaser der skal sørge for at der ikke tilføjes utilsigtede eller urealistiske data til databasen. Et eksempel på dette fra vores egen applikation kunne f.eks. være opret bruger-funktionen. Her skal det ikke være muligt for brugeren at oprette en bruger med et brugernavn der er længere end omkring 15 tegn, samt nogle tegn skal ikke være lovlige. På denne måde kan man undgå injection, plus det er nemmere når tilføje den oprettede bruger som ven, fordi deres brugernavn kun er 15 tegn eller mindre. Sanitation handler defor også om at indføre nogle begrænsninger for brugerinput der kan spare databasen fra at holde på urealistisk eller utilsigtet data.

Der er i dette projekt *ikke* blevet implementeret SQL Sanitation til databasen, da der har været tidsnød, og SQL Sanitation har været et nedprioriteret emne. Dette kan dog ses som en fremtidig tilføjelse til projektet.

4.3.5 Servercommunication klasser

Det skal være muligt for Marto at gemme data i databaseserveren, og det er derfor nødvendigt at opsætte nogle klasser der kan gennemløbe data i databasen, og returnere denne data så det kan sendes mellem serveren og klienter. Databasen indeholder derfor nogle klasser i mappen "ServerCommunication" som har til formål at gennemløbe database tables og returnere de påkrævede lagrede data.

Et eksempel på en af disse klasser kunne være "GetMyProfile". Denne klasse har til formål at returnere brugerens egen information når brugeren trykker på "My Profile"-knappen i Marto's GUI.

"GetMyProfile" returnerer brugerens brugernavn, status, beskrivelse, venneliste og brugerens tags. "GetMyProfile" skal derfor gennemløbe de tables i databasen hvor disse data er lagret. Brugernavn "Username", status "Status" og beskrivelse "Description" kan alle findes i databasetabellen "User-Information". Vennelisten kan findes i tabellen "FriendList", og tags kan findes i tabellen "Tags". Dataene gemmes i et "MyProfile"-objekt som har alle de nævnte data der skal vises for brugerens profil som parametre. Efter en gennemløbning af databasen returneres "MyProfile"-objektet.

"GetMyProfile" er kun ét eksempel af de klasser der skal skabe funktionalitet mellem database og server. "GetMyProfile" har til formål at returnere informationer om brugerens egen profil, hvorimod en klasse som f.eks. "AddFriend" skal tilføje en ven til brugerens venneliste.

4.3.6 Unit testing

For at kunne sikre sig, at modulerne virker som forventet, bliver der brugt unit tests. Unit tests har ansvaret for at teste, at når en metode eller funktion bliver kaldt, sker der det, som forventes. I databasens tilfælde, har dette været at, når der blev brugt unit of work til at f.eks. tilføje data i databasen, så ville der i databasen rent faktisk blive tilføjet data.

Udover disse individuelle unit tests for klasserne i databasen, bliver der også oprettet nogle klasser, der snakker sammen med serveren.

Unit testing er et yderst brugbart redskab hvad der angår arbejdet med databaser og servere. Når der sendes data mellem server og database vil der ofte være stor chance for at data bliver modtaget eller sendt i et andet format end det ventede. Det er derfor smart at kunne teste dette vha. en unit test. Unit tests er gode fordi det er isolerede tests, der kun fokuserer på en enkelt funktionalitet, og man kan derfor være sikker på, at det man tester fungerer ordentligt alt efter hvilket scenarie man tester.

Der anvendes unit testing til at teste om data hentes, tilføjes eller slettes korrekt fra databasen. Dette opnåes ved at skrive en Unit test klasse for hver ServerCommunication klasse. Hver klasse kommer til at have en Unit test-klasse af samme navn plus "Test" (eks. "GetMyProfileTest"). "GetMyProfile" anvendes som eksempel igen. Unit testområdet angives som et "TestFixture". Herefter defineres de parametre som den klasse der testes tager. I "GetMyProfile" ønskes der at se om klassen kan kigge i databasen og finde det brugernavn der angives. Inden testen kan laves skal der derfor oprettes et brugernavn i databasen. Dette kan gøres manuelt ved at "View Data" for den tabel som brugernavne ligger i. Der indsættes et brugernavn med samme værdi som det der bliver defineret i testen. Herefter kan der oprettes en "Test" ved at lave en ny klasse med et navn der giver mening i forhold til hvad der testes. Et eksempel på sådan en testklasse er "Profile_Username_IsInDB_Returns_Correctly()". Dette er en klasse der tester om det givne brugernavn findes i databasen, hvilket også fremgår tydeligt af klassenavnet. Selve kodelinjen der checker dette er funktionen `Assert.That(x,is.EqualTo(y))`, hvilket er en indbygget funktion fra `NUnit.Framework` namespace. Denne funktion anvendes hyppigt til de fleste unit tests, og er den funktion der kan sammenligne de to værdier. Hvis begge værdier er ens i testen vil testen vise sig som succesfuld.

5 Integrationstest

I dette afsnit bliver der beskrevet hvordan integrationen mellem GUI-klienten, serveren og databasen blev testet.

Systemet består af 2 moduler:

- Klient
- Server

Disse moduler består af 2 undersystemer:

- GUI
- Database

GUI hænger sammen med klienten og Serveren hænger sammen med Databasen

Integrationstest er udført i tre etaper. Første etape er Database og server som blev integreret sammen, derefter anden etape blev GUI og klient integreret sammen. Da disse moduler virkede sammen, blev der som tredje og sidste etape sat sammen til en helhed.

Integrationstesten mellem server og database, blev udført ved først at designe og implementere de nødvendige klasser til håndtering af requests, som serveren havde brug for. Herefter blev der opstillet nogen unit tests, for om disse klasser nu kunne gemme og hente data korrekt fra databasen. Når disse opstillede scenarier gav det forventede resultat, kunne integrationen mellem server og database konkludere vellykket.

GUI og klient skulle modificeres til en enkelt sammenspillende modul, blev ved hjælp fra Server siden lavet nogle fælles afgrænsinger, også kaldt en protokol, gav en kommunikationsmetode mellem Server og Klienten. Der blev testet i form af en systematisk gennemgang fra userstories om de features virkede, startet med modtagelse og sending af beskeder. Derefter blev systemet testet igennem og rettet til alt efter hvad hovedpunkterne var for produktet kunne blive kaldt fuldført.

Der blev undervejs i integrationen stødt på problem i forhold til design af de forskellige klasser systemet består af. Det viste sig at nogen funktioner skulle refaktoreres en smule, for at modulerne kunne virke sammen.

6 Accepttestspecifikation

6.1 Funktionelle krav

Denne sektion omhandler accepttestspecifikation af de funktionelle krav for Marto. De funktionelle krav er i dette tilfælde specificeret gennem user stories.

6.1.1 Profil Log Ind

Aktuel user story		User story 1: Profil Log Ind		
Scenarie		Brugeren indtaster brugernavn og password og trykker log ind på GUI mens brugerinfo findes i databasen		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugernavn og password indtastes, hvorefter der trykkes på "Sign In"	MartUI fortsætter til næste vindue	MartUI fortsætter til næste vindue	Godkendt
2	Indtastede brugernavn og password er ugyldigt, eller eksisterer ikke i databasen. Der trykkes på "Sign In"	MartUI viser en fejlbesked, og forbliver på i det nuværende vindue	MartUI viser fejlbesked og forbliver i samme vindue	Godkendt

Tabel 5: Accepttest af user story 1 : Profil Log Ind

6.1.2 Opret profil

Aktuel user story		User story 2: Opret profil		
Scenarie		Brugeren indtaster brugernavn og password og trykker "Sign Up"		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugernavn og password indtastes, hvorefter der trykkes på "Sign Up". Brugerinformation findes ikke i databasen	MartUI viser en besked om at oprettelsen af bruger er godkendt, brugerinformation oprettes i databasen	MartUI viser besked om brugeroprettelse, database indeholder brugerinformation	Godkendt
2	Brugernavn og password indtastes, hvorefter der trykkes på "Sign Up". Brugerinformation findes allerede i databasen	MartUI viser fejlbesked om at brugerinformation allerede findes, brugerinformation oprettes ikke i databasen	MartUI viser fejlbesked, brugerinformation oprettes ikke i databasen	Godkendt

Tabel 6: Accepttest af user story 2: Opret Profil

6.1.3 Fjern profil

Aktuel user story		User story 3: Fjern profil		
Scenarie		Brugeren ønsker at fjerne sin brugerinformation fra databasen		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren trykker på "Remove Profile"	Brugerinformation for brugerens egen profil fjernes fra databasen	"Remove Profile" findes ikke	Fejlet

Tabel 7: Accepttest af user story 3: Fjern profil

6.1.4 Søg efter profilnavn

Aktuel user story		User story 4: Søg efter profilnavn		
Scenarie		Brugeren vil søge efter et andet profilnavn		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren trykker på Profil-ikonet, indtaster et profilnavn og trykker enter	Profilen med det givne brugernavn fremvises	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 8: Accepttest af user story 4: Søg efter profilnavn

6.1.5 Søg efter tag

Aktuel user story		User story 5: Søg efter tag		
Scenarie		Brugeren vil finde andre profiler ved at angive tags		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren trykker på tag-ikonet, indtaster et gyldigt tag og trykker enter	Brugeren får vist en liste over de profiler der har tilknyttet det angivende tag	MartUI viser en liste af brugere der er tilknyttet det angivende tag	Godkendt

Tabel 9: Accepttest af user story 5: Søg efter tag

6.1.6 Tilføj ven

Aktuel user story		User story 6: Tilføj ven		
Scenarie		Brugeren ønsker at tilføje en ven		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren trykker på ikonet med en profil og et plus. Brugeren indtaster et gyldigt profilnavn som eksisterer i databasen og trykker enter	Den indtastede profil tilføjes til brugeren venneliste	Den indtastede profil bliver tilføjet til brugerens venneliste	Godkendt
2	Brugeren trykker på ikonet med en profil og et plus. Brugeren indtaster et profilnavn som ikke eksisterer i databasen og trykker enter	Der tilføjes intet til vennelisten, og systemet ryder tekstboksen	Der tilføjes intet, systemet ryder tekstboksen	Godkendt

Tabel 10: Accepttest af user story 6: Tilføj ven

6.1.7 Se venneliste

Aktuel user story		User story 7: Se Venneliste		
Scenarie		Brugeren ønsker at se listen over tilføjede venner		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren tilføjer en ven, hvorefter den tilføjede ven bør vise sig på vennelisten ude til venstre	Den tilføjede ven viser sig på vennelisten	Den tilføjede ven viser sig på vennelisten	Godkendt

Tabel 11: Accepttest af user story 7: Se venneliste

6.1.8 Slet ven

Aktuel user story		User story 8: Slet ven		
Scenarie		Brugeren ønsker at fjerne en ven fra sin venneliste		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren højreklikker på den ven der ønskes fjernet fra vennelisten og vælger "Remove"	Den valgte ven fjernes fra vennelisten	Vennen fjernes fra vennelisten	Godkendt

Tabel 12: Accepttest af user story 8: Slet ven

6.1.9 Rediger profil billede

Aktuel user story		User story 9: Rediger profil billede		
Scenarie		Brugeren ønsker at ændre sit profilbillede efter at have oprettet sin profil		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren navigerer til sin egen profil og trykker på "Change" under sit profilbillede og vælger det ønskede profilbillede	Profilbilledet inde på brugeren egen profil ændres til det valgte profilbillede	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 13: Accepttest af user story 9: Rediger profil billede

6.1.10 Rediger profil tag

Aktuel user story		User story 10: Rediger profil tag		
Scenarie		Brugeren ønsker at ændre de tags der er knyttet til sin profil		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren navigerer til sin egen profil og indtaster nye tags i feltet, hvorefter der trykkes på "Save"	Databasen opdateres med de nye specificerede tags knyttet til brugerens egen profil	De knyttede tags findes i databasen knyttet til brugeren	Godkendt

Tabel 14: Accepttest af user story 10: Rediger profil tag

6.1.11 Rediger profil beskrivelse

Aktuel user story		User story 11: Rediger profil beskrivelse		
Scenarie		Brugeren ønsker at sine egen profils beskrivelse		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren navigerer til sin egen profil, brugeren ændrer beskrivelsen for sin profil og trykker "Save"	Brugerens profilbeskrivelse opdateres i databasen	Brugerens beskrivelse ændres i databasen	Godkendt

Tabel 15: Accepttest af user story 11: Rediger profil beskrivelse

6.1.12 Rediger profilkoden

Aktuel user story		User story 12: Rediger profilkoden		
Scenarie		Brugeren ønsker at ændre koden til sin profil		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren navigerer til sin egen profil og trykker på "Change Password"-knappen	Brugeren indtaster et nyt password, og trykker "Save"	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 16: Accepttest af user story 12: Rediger profilkoden

6.1.13 Chat Sende tekstbesked

Aktuel user story		User story 13: Chat Sende tekstbesked		
Scenarie		Brugeren ønsker at sende en tekstbesked		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren vælger en ven at sende sin besked til, indtaster beskeden og trykker enter	Beskeden sendes til den valgte ven	Beskeden sendes til den valgte ven	Godkendt

Tabel 17: Accepttest af user story 13: Chat Sende tekstbesked

6.1.14 Modtage tekstbesked

Aktuel user story		User story 14: Modtage tekstbesked		
Scenarie		En anden klient har sendt besked til brugeren, og brugeren ønsker at se denne		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	En anden klient har sendt en tekstbesked til brugeren, og brugeren kigger i sin chatsamtale med den anden klient om denne besked ligger der	Beskeden viser sig i samtalen med den anden klient	Beskeden viser sig	Godkendt

Tabel 18: Accepttest af user story 14: Modtage tekstbesked

6.1.15 Se tidligere besked

Aktuel user story		User story 15: Se tidligere beskeder		
Scenarie		Brugeren ønsker at se tidligere beskeder sendt mellem brugeren og en ven		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Der er gemte beskeder mellem brugeren og den anden klient, det er muligt at se disse beskeder mellem brugerne, også hvis der skiftes vindue	Det er muligt at se alle beskeder der er gemt mellem to brugere i den givne samtale	Gamle beskeder mellem de to brugere vises	Godkendt

Tabel 19: Accepttest af user story 15: Se tidligere beskeder

6.1.16 Sende emojis

Aktuel user story		User story 16: Sende emojis		
Scenarie		Brugeren ønsker at tilføje en emoji til sin tekstbesked		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren vælger en emoji fra emoji-tabben, og denne indsættes i tekstbeskeden som brugeren er ved at skrive	Emoji'en indsættes på den plads i tekstbeskeden som brugeren på det nuværende tidspunkt sidder og redigerer på	Denne funktionalitet er ikke blevet implementeret	Fejlet

Tabel 20: Accepttest af user story 16: Sende emojis

6.1.17 Modtage emojis

Aktuel user story		User story 17: Modtage Emojis		
Scenarie		Brugeren modtager i tekstbesked der indeholder en emoji		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren åbner sine samtale med en anden klient og ser en tekstbesked med en emoji i	Emoji'en bliver fremvist korrekt	Dene funktionalitet er ikke implementeret	Fejlet

Tabel 21: Accepttest af user story 17: Modtage Emojis

6.1.18 Sende billeder

Aktuel user story		User story 18: Sende billeder		
Scenarie		Brugeren ønsker at sende et billede til en af sine venner		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren trykker på billedevedhæftnings-knappen, vælger billedet og trykker send	Billedet sendes til den anden klient og vises i samtalen	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 22: Accepttest af user story 18: Sende billeder

6.1.19 Modtage billeder

Aktuel user story		User story 19: Modtage billeder		
Scenarie		Brugeren ønsker at se de billeder der modtages fra andre brugere		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	En anden klient sender et billede i samtalen med brugeren	Billedet vises i korrekt format	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 23: Accepttest af user story 19: Modtage billeder

6.1.20 VOIP (Voice Over Internet Protocol)

Aktuel user story		User story 20: VOIP (Voice Over Internet Protocol)		
Scenarie		Brugeren ønsker at starte en samtale med en anden bruger hvor brugerne kan snakke med hinanden over en VOIP		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren højreklikker på den bruger som brugeren ønsker at ringe til på sine venneliste	Der oprettes en forbindelse hvor de to brugere kan tale via en VOIP	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 24: Accepttest af user story 20: VOIP (Voice Over Internet Protocol)

6.1.21 Gruppechat

Aktuel user story		User story 21: Gruppechat		
Scenarie		Brugeren ønsker at opstille en gruppesamtale hvor der kan være flere brugere der chatter på samme tid		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren kan tilføje flere brugere til en aktiv samtale ved at højreklikke på en bruger fra vennelisten der ikke findes i samtalen og trykke "Add to Conversation"	Der oprettes en ny samtale med både den bruger der var en aktiv samtale med før, og den bruger der netop er tilføjet	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 25: Accepttest af user story 21: Gruppechat

6.1.22 DecisionMaker Tilfældighed

Aktuel user story		User story 22: DecisionMaker Tilfældighed		
Scenarie		Brugere kan anvende DecisionMaker til at tage beslutninger ved at udvælge brugerdefinerede muligheder tilfældigt		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren starter DecisionMaker ved at skrive "!!DecisionMaker: x, y..." i tekstfeltet i en samtale med en eller flere brugere, og specificerer mere end én mulighed	En mulighed vælges helt tilfældigt efter DecisionMaker har kørt	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 26: Accepttest af user story 22: DecisionMaker Tilfældighed

6.1.23 DecisionMaker Format

Aktuel user story		User story 23: DecisionMaker Format		
Scenarie		Brugeren vil skrive de muligheder som DecisionMaker skal vælge tilfældigt ud fra		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Brugeren starter DecisionMaker og indtaster sine muligheder i det givne format: "!!DecisionMaker: x, y, z"	DecisionMaker åbner op med mulighederne "x", "y" og "z"	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 27: Accepttest af user story 23: DecisionMaker Format

6.1.24 DecisionMaker Layout

Aktuel user story		User story 24: DecisionMaker Layout		
Scenarie		Brugeren ønsker at have et visuelt lykkeshjul der drejer rundt når en af de specificerede muligheder skal udvælgles tilfældigt via DecisionMaker		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Efter der er specificeret muligheder for Decision-Maker skal den åbne for et interface der viser et lykkeshjul der udvælger en af mulighederne	Interfacet med lykkeshjulet åbner	Denne funktionalitet er ikke implementeret	Fejlet

Tabel 28: Accepttest af user story 24: DecisionMaker Layout

6.2 Ikke-funktionelle krav, samt resterende funktionelle krav

6.2.1 Understøttelse af flere klienter

Funktionelle krav		Ikke-funktionelle krav 1: Understøttelse af flere klienter		
Scenarie		Op til 5 klienter skal kunne anvende MartUI på samme tid		
Prækondition		GUI har forbindelse til systemet, server og database		
Nr	Test	Forventet resultat	Resultat	Godkendt
1	Der åbnes 5 klienter på MartUI, hvorefter der undersøges om programmet er funktionelt	Programmet fungerer som forventet	Programmet fungerer som forventet med op til 5 klienter	Godkendt

Tabel 29: Accepttest af ikke funktionelle krav 1: Understøttelse af flere klienter

Bibliography

- [1] Sometitle www.exampleurl.com
- [2] Server <https://stackoverflow.com/questions/6187456/tcp-vs-udp-on-video-stream>
- [3] TLS handshake fra IBM https://www.ibm.com/support/knowledgecenter/SSFKSJ_7.5.0/com.ibm.mq.sec.doc/q009930_.htm#q009930___q009930_4
- [4] GUI design patterns *Fra BlackBoard under Software Design -> MVC/MVP og MVVM. AARHUS UNIVERSITY SCHOOL OF ENGINEERING Electro and Information Communication Technology Department*
The ModelViewViewModel Design Pattern
2013 Poul Ejnar Rousing Version 0.51 – January 2014
- [5] Freezeable Guide <https://www.thomaslevesque.com/2011/03/21/wpf-how-to-bind-to-data-when-the-datacontext-is-not-inherited/>
- [6] MVVM - Commands <https://msdn.microsoft.com/en-us/magazine/dn237302.aspx>
- [7] MVVM - Events <https://lostechies.com/derickbailey/2013/03/18/event-aggregator-andorvs-mediator-a-tale-of-two-patterns/>
- [8] MVVM - PRISM <https://prismlibrary.github.io/docs/wpf/Implementing-MVVM.html>
- [9] MVVM - Simplifying properties <https://www.codeproject.com/Tips/412985/Simplifying-MVVM-Properties>
- [10] TokenizingControl <https://blog.pixelingene.com/2010/10/tokenizing-control-convert-text-to-tokens/>