

---

# Projektrapport for Marto

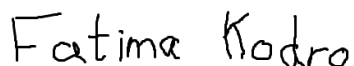
---

Forfattere:	Alexander Lichtenstein Davidsen, Daniel Pat Hansen, Fatima Kodro, Frederik Kastrup Mortensen, Martin Haugaard Andersen, Søren Bech og Søren Schou Mathiasen
Antal sider:	48
Lokation:	Aarhus Universitet
Vejleder:	Jesper Rosholm Tørresø



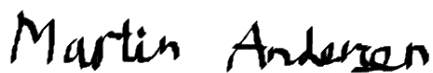
---

Alexander Lichtenstein Davidsen  
201608479, IKT



---

Fatima Kodro  
201609565, IKT



---

Martin Haugaard Andersen  
201605036, IKT



---

Daniel Pat Hansen  
201601915, IKT



---

Frederik Kastrup Mortensen  
201607221, IKT



---

Søren Bech  
201604784, IKT



---

Søren Schou Mathiasen  
201605264, IKT

May 29, 2018

# 1 Resume/Abstract

## 1.1 Resume

Denne rapport omhandler udviklingen af en chat app, Marto. Appen kan bruges af flere brugere til at skrive sammen, se informationer om hinanden, samt hvilke "tags" der kendetegner dem. På denne måde kan brugerne finde andre brugere baseret på fælles interesser.

Produktet er udviklet som 4. semesterprojekt på IKT uddannelsen på Aarhus Universitet. Der er i dette projekt blevet brugt værktøjer, som er lært i tidligere semestre som f.eks. SCRUM til at gøre arbejdsprocessen simple, og Git til versionsstyring.

Som krav var der, at alle semestrets fag skulle inddrages, hvilket vil sige der skulle være en GUI, Database og kommunikation over internettet. Her blev der sat en brainstorm igang med ideer til et projekt. Da projektet blev fundet, blev der arbejdet på en kravspecifikation, hvor den ønskede funktionalitet blev fundet. Dernæst blev der lavet arkitektur og design over systemet. Herefter blev implementeringsfasen igangsat, som endte i at der skulle laves unit- og integrationstests af hele koden.

Udviklingsforløbet er dokumenteret i denne rapport og dokumentationsdokumentet. Disse vil give indblik i hvordan arbejdet med projektet har været, samt hvordan chat appen er blevet til.

## 1.2 Abstract

This report is about the development of a chat app. This app can be used by multiple users to write together, see informations about each other, and see what "tags" characterizes them. In this way, the users can find other users with similar interests.

This product is developed as the 4th semester project on the IKT education in Aarhus University. There had in this project been used tools, which has been learned in previous semesters as fx. SCRUM to make the work process more simple, and Git for version control.

As requirements, all the subjects in the semester had to be used, this means there had to be a GUI, Database and communication through the internet. At this point, a brainstorm was started with ideas for a project. When the project was found, a requirement specification was worked out, where the functionality of the project was decided. Next there had been worked on architecture and design for the system. Then the implementation phase started, which ended in unit- and integration tests of the code.

The process is documented in this report and the documentation document. They will give insight into the work leading to the chat app completion.

# Indholdsfortegnelse

<b>1</b>	<b>Resume/Abstract</b>	<b>1</b>
1.1	Resume . . . . .	1
1.2	Abstract . . . . .	1
<b>2</b>	<b>Forord</b>	<b>3</b>
<b>3</b>	<b>Indledning</b>	<b>4</b>
3.1	Arbejdsfordeling . . . . .	4
3.2	Termliste . . . . .	5
<b>4</b>	<b>Projektformulering</b>	<b>6</b>
<b>5</b>	<b>Kravspecifikation</b>	<b>8</b>
5.1	User stories . . . . .	8
<b>6</b>	<b>Projektafgrænsning</b>	<b>11</b>
<b>7</b>	<b>Metode og proces</b>	<b>12</b>
7.1	Metode . . . . .	12
7.2	Proces . . . . .	13
<b>8</b>	<b>Arkitektur</b>	<b>15</b>
<b>9</b>	<b>Design, implementering &amp; test</b>	<b>19</b>
9.1	GUI . . . . .	19
9.2	Server . . . . .	27
9.3	Database . . . . .	34
<b>10</b>	<b>Accepttestspecifikation</b>	<b>41</b>
<b>11</b>	<b>Resultater &amp; Diskussion</b>	<b>42</b>
11.1	Personlige Konklusioner . . . . .	43
<b>12</b>	<b>Konklusion</b>	<b>46</b>
12.1	Projektarbejdet . . . . .	46
12.2	Procesarbejdet . . . . .	46
<b>13</b>	<b>Fremtidigt arbejde</b>	<b>47</b>
	<b>Bibliography</b>	<b>48</b>

## 2 Forord

Dette projekt er udviklet af syv 4. semester ingeniørstuderende inden for Informations- og kommunikationsteknologi.

Projektets vejleder er Jesper R. Tørresø, som er en tildelt vejleder med knytning til universitetet. Afleveringsfristen er den 30. Maj 2018.

Bedømmelsesdatoen er den 28. Juni 2018 kl. 12:00, hvor der skal gives en mundtlig præsentation efterfulgt af en individuel eksamination.

Projektet dokumenteres i denne projektrapport, samt et dokumentationsdokument. Dokumentationsdokumentet indeholder yderligere beskrivelser af sourcekoder, figurer osv. samt indeholder det andre projektrelateret materialer.

## 3 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 brugeren kan logge ind med et brugernavn og adgangskode vha. en grafisk brugergrænseflade. Herefter bliver brugeren navigeret ind til selve programmet, hvor det er muligt at redigere sin profil, samt møde andre brugere og skrive med dem.

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.

### 3.1 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

## 3.2 Termliste

**Marto** er selve chat-programmet.

**MartUI** er den grafiske brugergrænseflade.

**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.

**Klient** referres til klassen Client i selve MartUI.

**DecisionMaker** er en slags roulette til at udvælge en tilfældig kategori ud af flere mulige brugerdefinerede kategorier.

## 4 Projektformulering

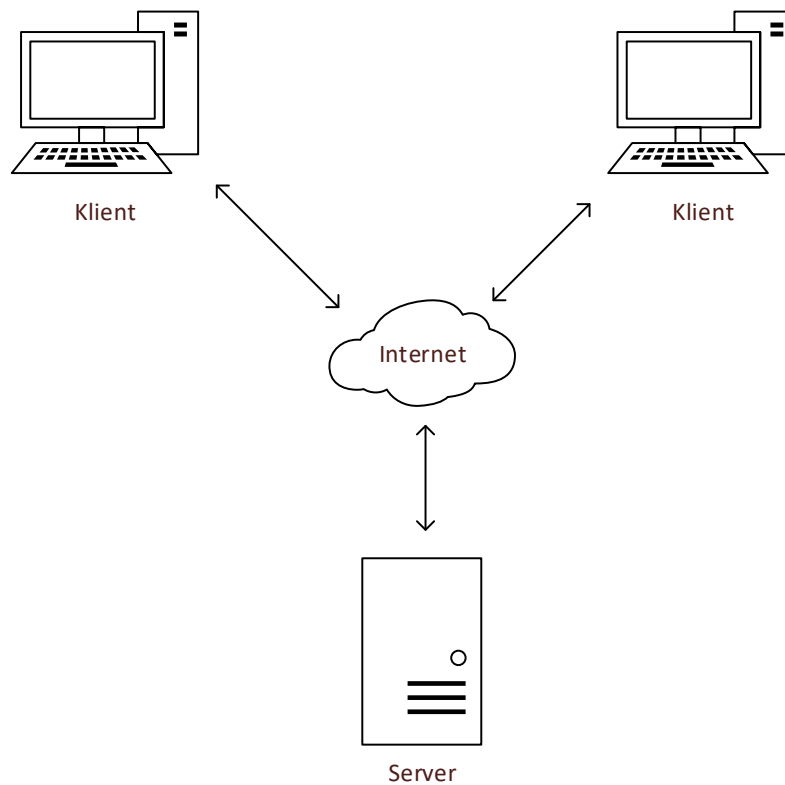
Som emne til 4. semesterprojekt er der valgt at der skal designes og implementeres et chatsystem som en Windows applikation. Brugeren kan oprette sig med et brugernavn og et kodeord gennem brugergrænsefladen. Herefter kan brugeren logge på med sit brugernavn og kodeord.

Brugeren kan søge efter andre brugere og tilføje dem som venner. Herefter kan man indgå i en privat chat med sine venner. Her skal der være en chat historik, så man kan se beskeder fra tidligere sessioner, samt beskeder der er sendt, mens brugeren har været offline.

Brugeren kan redigere sin profil, med en beskrivelse af sig selv og et billede. Man kan tilføje brugerdefinerede tags til sin profil, så andre brugere kan finde folk med tags efter interesse. Som addition til chat-programmet laves der en DecisionMaker, der er en slags roulette, som tilfældigt kan udvælge en kategori, ud fra flere brugerdefinerede kategorier. Dette kan anvendes i forbindelse med brugere, som gerne vil lave noget sammen, men har forskellige ønsker. Dette hænder ofte for personer, som gerne vil spille videospil sammen, men er uenige om hvad de skal spille.

På figur 1 ses et rigt billede af, hvordan de forskellige moduler kommunikerer sammen. På figuren ses nogle klienter, som kommunikerer til en server via internettet. Klienterne kan her sende forespørgsler til serveren i form af nogle tekststrengene.

Hvis en bruger f.eks. skal logge ind på Marto, vil brugere skrive deres brugernavn og adgangskode, hvorefter klienten sender disse informationer til serveren, som herefter sender informationerne videre til DB serveren. Her bliver der tjekket om disse oplysninger findes på databasen. Hvis oplysningerne stemmer overens, vil DB serveren sende oplysninger tilbage til serveren om at brugernavn og adgangskode er godkendt. Herefter vil serveren sende oplysninger til klienten, om at brugeren er blevet logget ind.

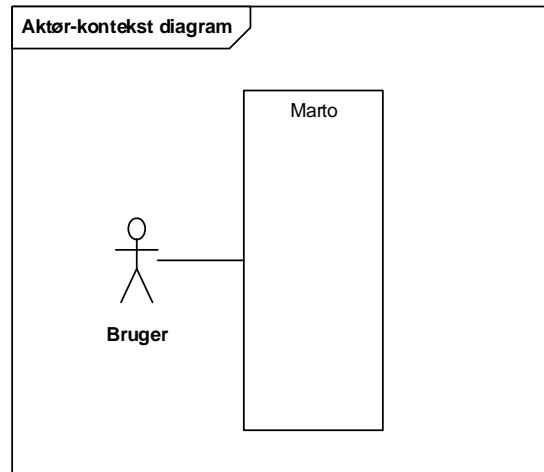


Figur 1: Rigt billede af Marto



# 5 Kravspecifikation

For at kunne udvikle et fuldendt produkt, er der opstillet en række krav. Disse krav er visualiseret som en række user stories, der fortæller hvordan en typisk bruger vil benytte produktet. Marto har en primær aktør som er brugeren, der skal bruge produktet. På figur 2 ses et aktør-kontekstdiagram, der viser interaktionen med Marto.



Figur 2: Aktør Kontekst Diagram

## 5.1 User stories

De forskellige user stories bliver vægtet efter prioritet. Til dette bliver der brugt MoSCoW modellen.

**MoSCoW** er et akronym for Must, Should, Could og Won't. MoSCoW anvendes til at beskrive produktkravene ud fra deres prioritet. Must beskriver de højest prioriterede krav for produktet. Disse krav er ofte hvad der gør produktet til det produkt det er, og er normalt hovedfunktionaliteten af produktet. Hvis et Must-krav ikke kan overholdes må produktet i de fleste tilfælde betragtes som mislykket.

Should-krav kan være lige så vigtige som nogle krav i Must-kravene, men er ikke nødvendige for den endegyldige levering af produktet. Should-krav er normalt ikke så tidskrævende som et Must-krav, og der kan være andre måder at tilfredsstille Should-krav.

Could-krav er ønskelige, men ikke nødvendige, og bør derfor kun være i fokus, hvis der er ekstra udviklingstid.

Won't-krav er krav for produktet, der er blevet skrottet af forskellige grunde. Dette kan enten være fordi, kravet ikke er vigtigt nok til at dedikere tiden til det, eller fordi det viste sig ikke at være nødvendigt.

### 5.1.1 Chat User stories

- **Sende tekstbesked:** Som bruger **skal** jeg kunne sende en teksbesked for at kommunikere 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.
- **Sende 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 **vil** jeg kunne sende billeder for hurtigt at vise andre brugere mine billeder.
- **Modtage billeder:** Som bruger **vil** 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 mundtligt.
- **Gruppechat:** Som bruger **vil** jeg kunne oprette en gruppechat for nemt at kunne kommunikere med flere brugere på samme tid.

### 5.1.2 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 mine informationer ikke længere er i systemet.
- **Tilføj ven gennem brugernavn:** Som bruger **skal** jeg kunne tilføje en profil som ven, baseret på brugernavn, så jeg kan kommunikere med denne.

- **Søg efter tag:** Som bruger **skal** jeg gerne kunne finde profiler, ved at søge efter et tag, så jeg kan finde andre brugere med dette.
- **Tilføj ven:** Som bruger **vil** jeg gerne kunne tilføje en ven, gennem deres profilside, så jeg kan kommunikere med denne profil.
- **Se ven:** Som bruger **skal** jeg gerne kunne se mine tilføjet 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 ændre min adgangskode for at have opdateret sikkerhed.

### 5.1.3 DecisionMaker

- **Tilfældighed:** Som bruger **vil** jeg kunne bruge DecisionMaker til tilfældigt at træffe en beslutning ud fra de brugerangivne muligheder.
- **Format:** Som bruger **vil** jeg kunne indtaste brugerdefinerede kategorier, for at gøre DecisionMaker mere relevant for mine interesser.
- **Layout:** Som bruger **vil** jeg kunne se rouletten med alle mulighederne, hvoraf den valgte tydeligt fremgår.

# 6 Projektafgrænsning

Der er på dette semester opstillet nogle krav fra skolen til udvikling af det færdige produkt. Fra oplægget til Semesterprojekt 4 er følgende afgrænsninger stillet:

- Projekt skal udføres iterativt
- Projektet skal udføres ud fra OOAD principper

Udover det er tilføjet understående punkter til afgrænsningen:

- Systemet skal virke på en Windows PC
- Systemet skal have en brugergrænseflade
- Systemet skal kommunikere over internettet
- Systemet skal gemme relevante informationer i en database

Til at opnå disse krav bliver der anvendt viden fra de forskellige fag på 4. semester. I dette projekt er alle semesterets fag inddraget. Herunder er fagene SWD(Software Design), SWT(Software Test), DAB(Database), IKN(Indledende kommunikations netværk) og GUI(Grafisk brugergrænseflade).

# 7 Metode og proces

## 7.1 Metode

Efter kravene til projektet var færdiggjort, og gruppen vidste hvad den ønskede funktionalitet skulle være, skulle der nu dannes et overblik over systemet. Her blev der arbejdet med UML.

### 7.1.1 UML

UML bruges i udarbejdelse af designet til software. UML indeholder en liste af værktøjer og en række diagrammer til softwaredesign. Funktionaliteten af softwaren bliver her dokumenteret med disse diagrammer. Hovedformålet er at en anden softwareingeniør, skal kunne læse disse diagrammer og få et godt indblik i softwaren, uden yderligere besvær.

Til dette projekt bruges nogle forskellige UML diagrammer såsom: Klassediagrammer og sekvensdiagrammer til det overordnede system.

Klassediagrammet bruges til at få et overblik over hvilke klasser systemet indeholder, samt hvilke metoder hver enkelt klasse indeholder.

Sekvensdiagrammet bruges til at se, hvad programmet skal gøre under nogle forskellige scenarier, og i hvilken rækkefølge de forskellige metoder bliver kaldt.

Udover disse overordnede diagrammer, bliver der også lavet nogle diagrammer til de enkelte områder, såsom GUI, database og kommunikation over internettet.

Til database er der lavet ERD-diagrammer, som viser hvilke entities, der har relationer mellem hinanden og, hvilke forbindelser de har.

Til GUI er der lavet state-machine diagrammer, der holder styr på, hvordan et system reagere på udefrakommende påvirkninger. Det viser hvilke states systemet er i, efter brugeren har trykket diverse knapper på GUI.

Til server er der lavet mere detaljerede sekvensdiagrammer, der viser sammenspillet mellem server, database og GUI.

## 7.2 Proces

Følgende sektion omhandler processen i projektforslaget

### 7.2.1 Gruppedannelse

Gruppen består af 7 IKT'er, som alle har tidligere kendskab til hinanden. Efter gruppen blev dannet, blev der hurtigst muligt fundet en dag, hvor hele gruppen kunne mødes sammen og finde ud af, hvilke styrker og svagheder der var hos de enkelte deltagere. Da hele gruppen består af IKT'er, gav det mest mening at projektforslaget kun skulle bestå af software udvikling.

### 7.2.2 Samarbejdskontrakt

For at få et godt udgangspunkt i gruppearbejdet, er der blevet oprettet en samarbejdskontrakt. I denne aftale er der beskrevet nogle betingelser og krav til alle medlemmer, som skal overholdes. Disse krav beskriver hvad der skal gøres, f.eks. hvis et medlem ikke kan møde op til et møde. Hvis disse krav ikke overholdes er der konsekvenser som også beskrives i samarbejdskontrakten.

### 7.2.3 Udviklingsforslaget

I gruppen er der anvendt SCRUM til at holde styr på de forskellige opgaver der skal udføres, ydermere bliver der arbejdet med en iterativ arbejdsmetode.

### 7.2.4 Projektledelse

Til SCRUM-møderne i gruppen, har der været en fast SCRUM-leder, som i starten af hvert møde havde planlagt en beskrivelse af alle de ting der skulle snakkes om til mødet. Herefter gik hele gruppen i gang med at diskutere disse emner. Møderne blev ofte brugt til at holde gruppen opdateret på de kommende deadlines. Der blev taget referater af møderne, opstillet på punktform, som beskrev de emner, der blev snakket om, samt de ting der skulle være klar til næste møde.

### 7.2.5 Arbejdsfordeling

Arbejdet er delt op således, at der er 2 personer der har arbejdet med Database, 2 personer der har arbejdet med GUI og 3 personer der har arbejdet med server. Herefter er hver gruppe blevet inddelt i flere mindre opgaver, som er fordelt mellem medlemmerne.

### 7.2.6 Planlægning

Der blev i starten af projektet uddelt en deadline til, hvornår der skulle afleveres et projektforslag og en problemformulering. Da dette var den eneste givne deadline, skulle der selv laves yderligere deadlines i gruppen. Dette blev lavet vha. sprints til SCRUM møderne. Dette forklares længere nede.

### 7.2.7 Projektadministration

Den måde gruppen har haft intern kommunikation mellem hinanden, er gennem facebook, hvor der bliver aftalt møder med hinanden. Der er brugt Github til at dele filer med hinanden, så alle filer bliver tilgængelige for allesammen i gruppen, samt til versionsstyring af projektet. Der er skrevet logbog til de dage, hvor der er kommet fremskridt i projektet, for at få en overordnet beskrivelse af, hvad der er blevet opnået i de forskellige perioder.

### 7.2.8 Møder

Der har været en del gruppemøder og vejledermøder. Vejledermøderne blev holdt, hvis gruppen sad fast i nogle arbejdsopgaver, der ikke kunne løses af sig selv. Der blev dog holdt gruppemøder en gang om ugen, hvor gruppen snakkede om, hvordan forskellige ting skulle laves.

### 7.2.9 SCRUM

I gruppen er der blevet anvendt SCRUM. SCRUM er en iterativ ramme i et projektforsløb, der gør det mere overskueligt at få styr på, hvilke opgaver der skal laves inden for de forskellige deadlines, samt giver det et indblik i hvilke opgaver den individuelle person skal arbejde på, ved at inddele et projekt i sprints.

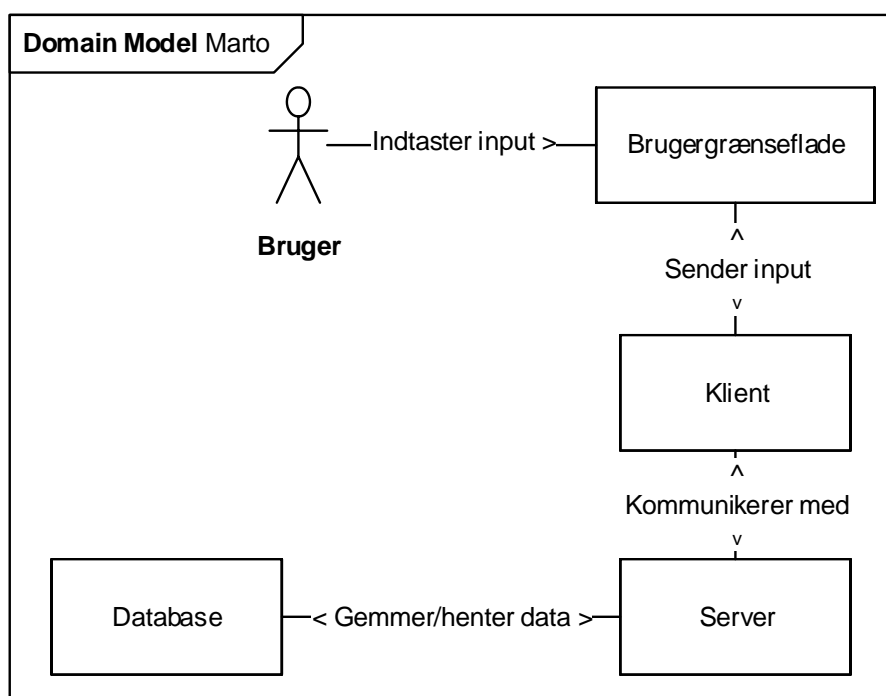
Til SCRUM møderne havde gruppen en SCRUM-master, der havde ansvar for at de forskellige sprints blev udført af alle medlemmer, samtidigt skulle SCRUM-master være ordfører til daily SCRUM.

Udover SCRUM-master, var det projektlederen der havde ansvar for at uddele nogle overordnede opgaver til hvert sprint, samt oprette en deadline til hvert sprint. Herefter skulle alle deltagere i fællesskab dele disse opgaver ind i mindre underopgaver. Når der var blevet lavet en god mængde undergaver, blev disse uddelt mellem alle medlemmer. Der blev sigtet efter at have omkring 4-5 underopgaver til hver deltager ved hvert sprint. Jo flere underopgaver hver deltager havde, jo nemmere var det at følge med i, hvor langt de var kommet med deres opgaver, da der hele tiden kunne følges med i hvilke underopgaver der var færdiglavede.

## 8 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.

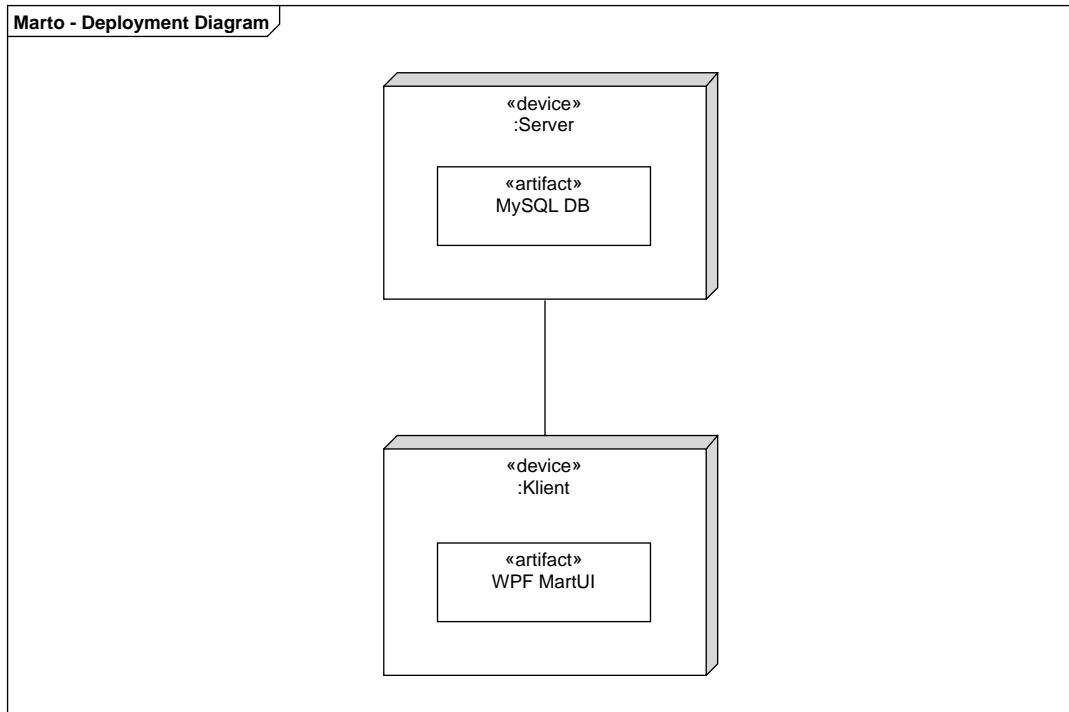
For at få et godt overblik over hele systemet, er der lavet en domænemodel. På figur 3 ses en domænemodel for det samlede system. Aktøren her er brugeren, og brugeren har her adgang til brugergrænsefladen, som sender/modtager input til/fra klienten. Klienten kommunikerer her sammen med serveren, som herefter går ind i databasen, for at tilgå relevant data for den type request, der er forekommet.



Figur 3: Domænemodel

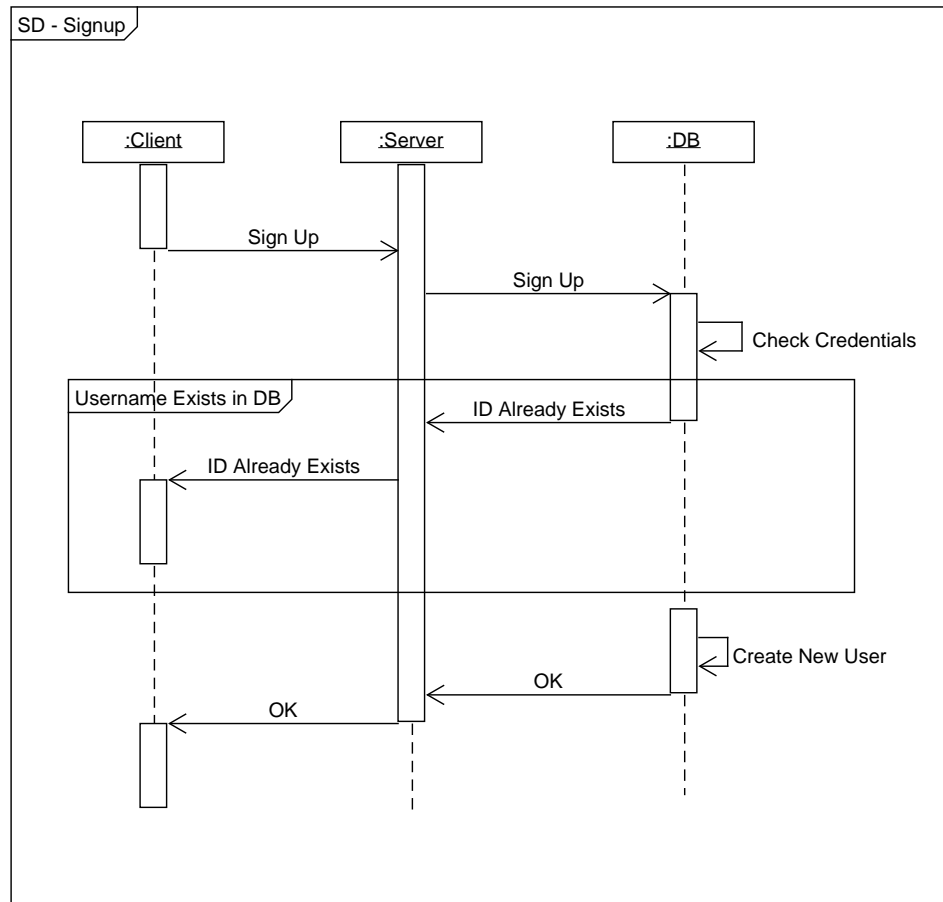


På figur 4 ses et deployment diagram over Marto systemet. Marto består af en server og en klient. Klienten har den grafiske brugergrænseflade, MartUI og serveren har MySQL databasen. Hvert device på deployment diagrammet repræsenterer en PC. Server device er her en PC, med både database og server.



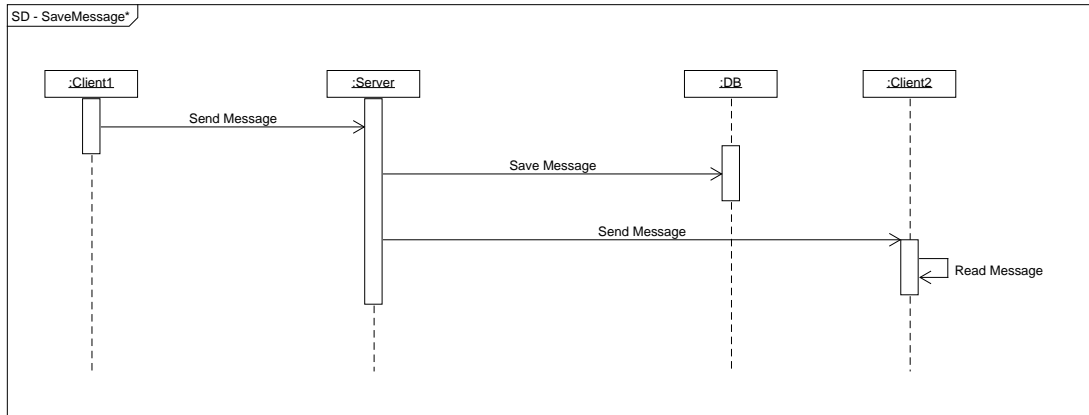
Figur 4: Deployment Diagram

På figur 5 ses, hvordan en bruger opretter en profil i Marto systemet. Klienten indtaster de ønskede login informationer: brugernavn og kodeord. Serveren kigger i databasen om det modtagne brugernavn allerede eksisterer i databasen. Hvis det gør, modtager brugeren en fejlmeddelelse, 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 5: Sekvensdiagram over oprettelse af bruger

Det ses på figur 6, hvordan en sendt besked opfører sig i systemet. Der sendes her en besked fra en klient til en anden klient. Når klient 1 skriver til klient 2, vil beskeden først blive gemt i databasen, med informationer om hvem der har sendt til hvem. Herefter bliver beskeden sendt videre til klient 2.



Figur 6: Sekvensdiagram over save message

For mere uddybende arkitektur henvendes der til dokumentationen. Der er i rapporten blot udvalgt få væsentlige diagrammer.

# 9 Design, implementering & test

## 9.1 GUI

Brugergrænsefladen i projektet, som brugeren kan tilgå, kaldes MartUI.

### 9.1.1 Overblik

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). Der er fokuseret på et design som er simpelt og brugervenligt. Der er anvendt dele af et framework fra Microsoft kaldet PRISM til at håndtere kommunikationen i MartUI. Her anvendes der to koncepter kaldet henholdsvis Events og Commands. Der er foretaget modultest af MartUI, men der er ikke anvendt et framework til at unit teste MartUI. Dette er muligt, men test af GUI i softwaretest faget kom sent i forløbet, og er derfor ikke blevet prioriteret fra starten af. Det tager desuden noget tid at opstille tests, og da tiden har været knap igennem projektet, er det blevet fravalgt. Det ville have været smart at lave dette fra starten af, men simple tests uden et framework har også fungeret fint.

#### 9.1.1.1 Overblik

Et af kravene til projektet er, at der skal udarbejdes en brugergrænseflade. Brugergrænsefladen udarbejdes i WPF. En WPF-applikation kan implementeres ved at definere arrangeringen 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.

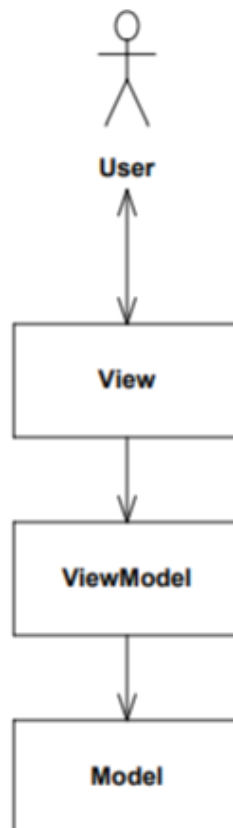
DecisionMaker, som er en del af kravene, er ikke implementeret, men en idé til layout kan ses i dokumentationen.

### 9.1.2 Analyse

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.

### 9.1.2.1 MVVM (Model-View-ViewModel)

MVVM er et design pattern som specifikt anvendes under design og udvikling af en brugergrænseflade 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 metode-kald og kan modtage events fra Model. Modellen kender ikke til View-Model. Figur 7 illustrerer MVVM. 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 7: Billede af et simpelt diagram over MVVM, som viser sammenhængen mellem View, View-Model og Model. Kilden til billedet findes i dokumentationen.

### 9.1.2.2 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. For en dybdegående analyse henvises til dokumentation under afsnit om GUI -> Analyse.

### 9.1.2.3 Klient valg

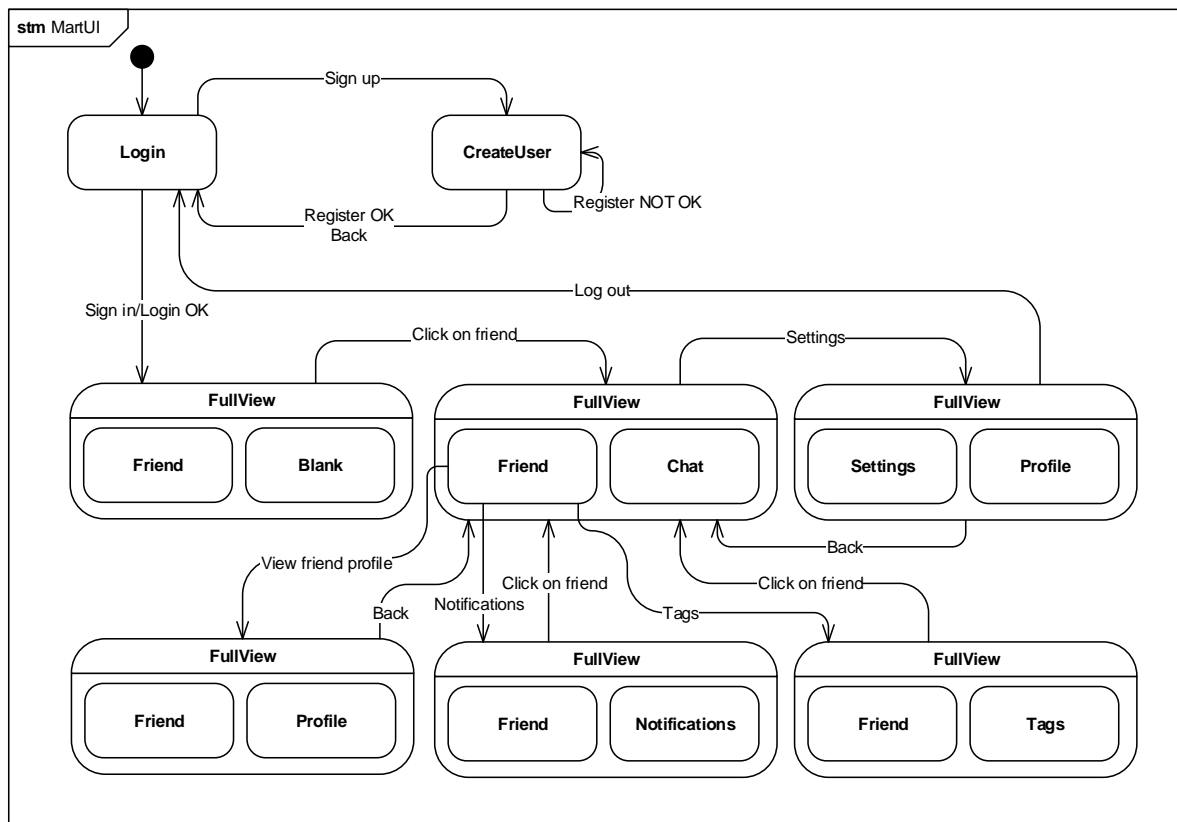
Der er flere måder at lave en klient på, den kan være baseret på en terminal eller GUI. Det vigtigste for en klient er, at den laver den såkaldte korrekte Client forbindelse som kunne enten være UDP eller TCP. Der blev Valgt at bruge en TCP client, og senere bliver denne forbindelse sendt igennem en sikker forbindelse.

Uddybbelse af selve klientvalget se dokumentation afsnit 5.1.1.7 For uddybbelse af selve den krypterede forbindelse henvises til dokumentationen afsnit 5.2.1.2.1

### 9.1.3 Design

MartUI er opbygget således, at brugeren nemt kan navigere rundt intuitivt. Det er noget, der generelt er vigtigt for brugergrænseflader, især som konkurrencen stiger, og kun de bedste bliver brugt. Når 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.

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 8.

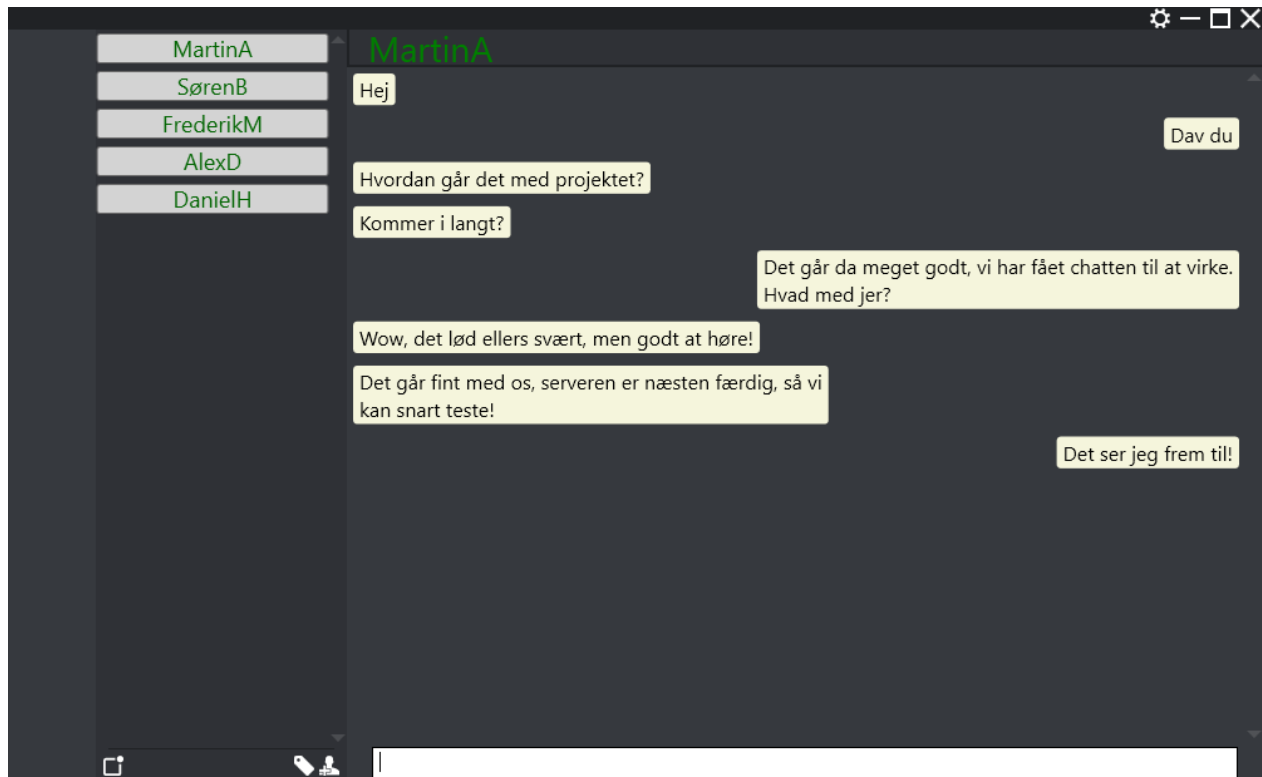


Figur 8: Tilstandsmaskine over MartUI

Når brugeren tilgår MartUI, bliver brugeren ført til login. Herefter kan brugeren vælge at trykke på Sign up eller Sign in. Trykker brugeren på 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 brugerens 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, på samme måde som "x" knappen altid kan tilgås. 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 henholdsvis på notifikation eller tags knappen.

Et billede af en skærm med vennelisten og en chat kan ses på figur 9. For billeder af alle "views"

(skærme) refereres der til dokumentation, afsnit om GUI, design og implementering.



Figur 9: Billede af det 'vigtigste' vindue, som er vennelisten og chatten

Da der forekommer en del forskellige sekvensdiagrammer og billeder til funktionaliteten af de forskellige views, refereres der til dokumentationen, afsnit om GUI, views samt design og implementering.

#### 9.1.3.1 Commands

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 handler 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.

#### 9.1.3.2 Events

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 viewmodels har ikke kendskab 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 pattern og events kan anvendes. Mediator pattern 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.

#### 9.1.3.3 Klient kommunikation

Klient klassen bruges til at styre selve beskederne der modtages fra serveren og beskeder fra selve brugeren. Klient er opbygget af en konstruktør som forbinder sig med den specifikke server og en eventhandler der håndterer de forskellige beskeder fra serveren. Selve modtagelsesdelen er eventhandleren og sender delen er en specifik brugt delimiter.

#### 9.1.4 Implementering

Implementeringen af MartUI tager udgangspunkt i at oprette et view med en tilhørende viewmodel (som oprettes som en klasse). En mulig model er også oprettet. Der er forsøgt at følge MVVM pattern så godt som muligt, dvs. der ikke er nogen logik i view (eller code-behind filen), og langt det meste af logikken er implementeret i viewmodel.

Der er taget delvist udgangspunkt i Microsofts PRISM, som anvendes til at håndtere MVVM. PRISM har en metode, SetProperty(), som kan anvendes i stedet for INotifyPropertyChanged. Denne anvendes for at kunne notificere view om at en værdi har ændret sig i viewmodel. Fordelen ved dette er blot færre linjer af koder, som der kan ses nedenfor.

SetProperty:

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

INotifyPropertyChanged:

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

```
    NotifyPropertyChanged();  
}
```

Disse to kodeudsnit er ens, men SetProperty() tjekker selv om værdien er opdateret og opdaterer den deraf.

#### 9.1.4.1 Commands og events

DelegateCommand fra PRISM er blevet brugt, som er måden hvorpå man kan definere en command. Dette er forbindelsen mellem view og viewmodel. Et eksempel på en ofte anvendt linje kode ses nedenfor.

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

ChangeToFriends er her en metode som er defineret i klassen, men metoden kan også defineres som et lambda expression.

For at kunne kommunikere på tværs af viewmodels er PRISM's EventAggregator klasse blevet anvendt. Dette er blot måden hvorpå man kan subscribe og publish events. Hvis en viewmodel vil dele noget information til en anden viewmodel uden at kende til denne viewmodel, kan den gøre dette ved at publish et event. Følgende kodeudsnit viser et eksempel på dette.

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

En command, changePageToFriends bliver defineret. Når den kaldes, vil den delegerer funktionen til at publish et event, GetFriendList. En mulig subscribe kan herefter reagere på dette.

Events er også anvendt når der skal skiftes view. En meget ofte anvendt linje kode til at skifte view ses nedenfor.

```
_eventAggregator.GetEvent<ChangeFriendPage>().Publish(new SettingsViewModel());
```

Denne linje kode vil kalde .GetEvent med ChangeFriendPage eventet, som tager en IPageView. Lader man alle viewmodels implementere dette view, er det muligt at sende viewmodels med som parameter. I dette tilfælde bliver SettingsViewModel sendt med som parameter, hvormed en subscriber kan reagere på dette. For yderligere forklaring om hvordan ChangePage og andre events fungerer, refereres der til dokumentationen om GUI, design og implementering.

#### 9.1.4.2 Klient klassen

Klienten er implementeret med GUIs events som håndtere alle de beskeder der bliver sendt fra serveren. Klienten er delvist en del af server, imens den også er en del af GUI. Derfor henvises implementeringen til selve dokumentationen på afsnit 5.1.8.1.

### 9.1.5 Test

Der er foretaget modul test af alle views/viewmodels. Der er blot testet om interfacet opdateres ved at observere skærmens output, eller/og udskrive værdier på skærmen. Det er muligt at indføre automatiserede unit tests med et framework, men denne måde fungerer fint. Optimalt ville unit tests tilføjes. For at læse nærmere om testene der er foretaget refereres der til dokumentationen, GUI, tests.

#### 9.1.5.1 Kommunikationstest

Det mest vitale ved klienten er at klienten kunne sende og modtage en string af beskeder. Næste del var indføre test med selve GUI events. Selve tests bestod af at om klienten kunne forbinde til serveren og derefter sende og modtage strings til og fra serveren.

## 9.2 Server

### 9.2.1 Indledning

Server analysen vil gennemgå, de forskellige teknologier der har været under overvejelse:

- Netværksprotokol (TCP, UDP)
- Besked Sikkerhed
- Data Sikkerhed
- Kryptering
- Skalerbarhed (Async, Thread)

Herefter beskrives det design af serveren gruppen er kommet frem til at være bedst for lige præcis denne løsning. Hvordan dette er implementeret og testet kommer efterfølgende.

Ved hvert emne har der været overvejelser som bliver uddybet i dokumentationen på afsnit 5.2 Server.

#### 9.2.1.1 Analyse

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

Baseret på vores viden omkring TCP og UDP samt baseret på de krav der er stillet, er 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 opsat krav angående 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.

### 9.2.2 Data kryptering

Der er blevet brugt tre metoder til at kryptere oplysningerne fra klienten.

- SHA-2 (Hashing)
- Salt
- TLS (SSL)

For uddybbelse af analysen henvises til dokumentationen afsnit 5.

#### 9.2.2.1 SHA-2

For at undgå at Marto brugernes adgangskoder gemmes i plaintext, køres disse igennem en hashing algoritme så de ikke længere er genkendelige. Algoritmen som gruppen bruger hedder SHA-2. Denne omdanner et input til en 32 karakter lang tekststreng som ikke ligner inputtet. Da SHA-2 er en meget brugt og veldokumenteret algoritme er det denne som gruppen har valgt at bruge.

#### 9.2.2.2 Salt

SHA-2 kan omdanne en plaintext adgangskode til en 32 karakter lang hashet tekststreng, men SHA-2 er ikke helt nok til beskyttelse mod ondsindede elementer, her er rainbow tables et problem.

Derfor kan en ekstra foranstaltning tilsættes systemet, kaldet Salt. Et Salt indenfor kryptering er en tilfældig genereret tekststreng, som sættes på den tekststreng, der skal hashes. På denne måde gælder et generelt SHA-2 rainbow table ikke for dette system, hvilket skaber større sikkerhed for brugerne.

#### 9.2.2.3 TLS

En TLS forbindelse er en krypteret form for forbindelse der kan gardere mod networksniffing. Da beskeder der sendes over et trådløst netværk kan bliver opfanget og læst af andre, ønsker gruppen at sende al data i krypteret form. TLS håndterer dette problem med at det kryptere beskeder imellem bruger og server. Dette gør at en "sniffer" stadig kan opfange beskeder, men disse vil være uforståelige og sensitivt data er derfor sikkert.

### 9.2.3 Skalerbarhed

Da Marto er et chatprogram der ikke har et prædefineret 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 der ikke forekommer crashes, når mange brugere benytter sig af systemet.

Grunden til at async sockets ikke blev valgt, selvom det er en fin løsning til et skalerbart system, så ville systemet blive svært at implementere. Derfor med de punkter i Userstories, og prioriteten af fri implementeringsvalg, er en multithreaded løsning valgt. Der er fuld opmærksomhed på at der muligvis er et højere forbrug af hukommelse, samt mere spildtid med context switching, som kan hindre skalerbarheden, men implementeringsmæssigt vil det være lettere at få systemet til at køre acceptabelt.

I afsnit 5 i dokumentationen, bliver der uddybet både for async og multithreading.

## 9.2.4 Design

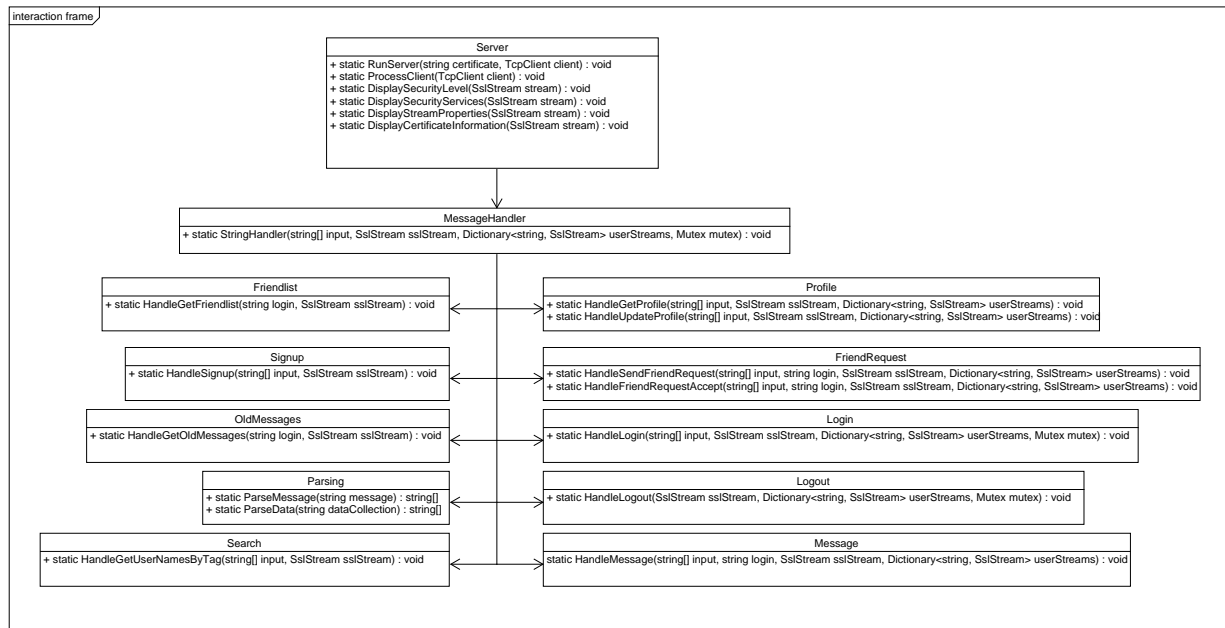
### 9.2.4.1 Server til klient protokol

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

<b>Funktion</b>	<b>Protokol</b>
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 i den første del af strengen.

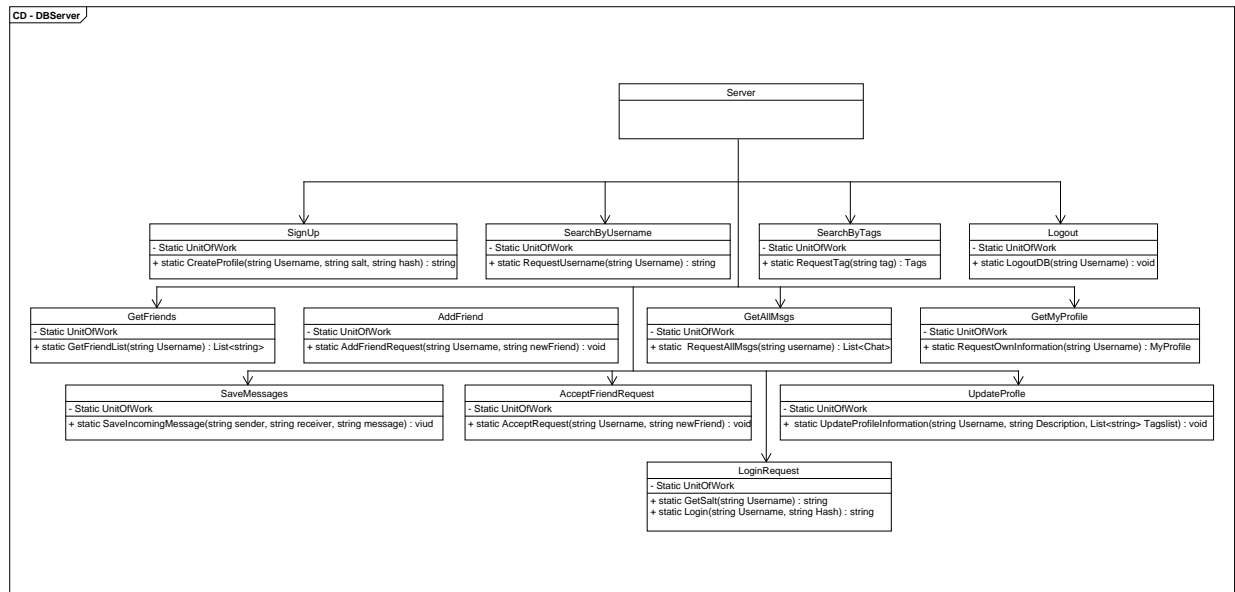


Figur 10: Klassediagram over server

På figur 10 ses et klassediagram over de klasser der udgør intern kommunikationsdelen af serveren. Klassen Sender er bevidst ikke medtaget i klassediagram for overskuelighedens skyld. Sender klassen bliver brugt af Friendlist, Friendrequest, Login, Signup, Search, Profile, Oldmessages, Messages og Messagehandler. En dybdegående beskrivelse af klasserne og deres metoder, kan findes i dokumentationen, under implementering og design for server.

### 9.2.4.2 Kommunikation med DB

Til kommunikation med DB er der designet og implementeret en række klasser, hver med deres ansvarsområde. Klasserne er designet ud fra SOLID, 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 10.



Figur 11: 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 11. En dybdegående beskrivelse af klasserne og deres metoder, kan findes i dokumentationen, under implementering og design for server.



### 9.2.5 Implementering

For at lave en server der altid er klar på at modtage nye klient forbindelser, samt modtage og sende beskeder fra og til klienterne, er en multithreaded løsning blevet implementeret. Server programmet har en tråd kørende i en listener, der opretter en ny tråd for hver nye klient der bliver oprettet. Disse tråde kører hele tiden i blokerende read, og håndterer så de modtagne beskeder. På denne måde giver de blokerende read metoder ikke problemer, da hver klient kører i sin egen tråd.

Den krypterede forbindelse mellem server og klienter, er implementeret ved hjælp af Microsoft's SslStream. Med et '.cer' certifikat og en Tcp stream, er en SslStream oprettet, som krypterer den sendte data. Med denne implementering er vores data blevet sikret mod packet sniffing.

For at sikre brugernes login oplysninger, bliver adgangskoder hashet før de gemmes i systemets database. Dette er implementeret således at når en ny bruger oprettes, generes en tilfældig tekststreng, kaldet Salt. Brugerens adgangskode sammen med saltet køres igennem en hashing algoritme, hvorefter brugernavn, salt og det hashede kodeord gemmes i databasen. Når en bruger ønsker at logge ind, sendes brugernavn og adgangskode til server. Server finder brugerens salt og hashede password fra databasen, og tilsætter saltet til adgangskoden hvorefter dette igen køres igennem hashing algoritme. Denne hashede adgangskode sammenlignes med den fra databasen, og hvis disse stemmer overens, bliver brugeren logget ind. Denne implementering gør at en brugers plaintext adgangskode ikke bliver gemt i databasen. For uddybende beskrivelse af dette refereres der til server implementerings afsnittet i dokumentationen.

### 9.2.6 Test

Til kommunikation med databasen er der lavet unit test på de forskellige klasser. 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 blot 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.

Til test af kommunikationen med databasen, er der opstillet en række unit tests 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.

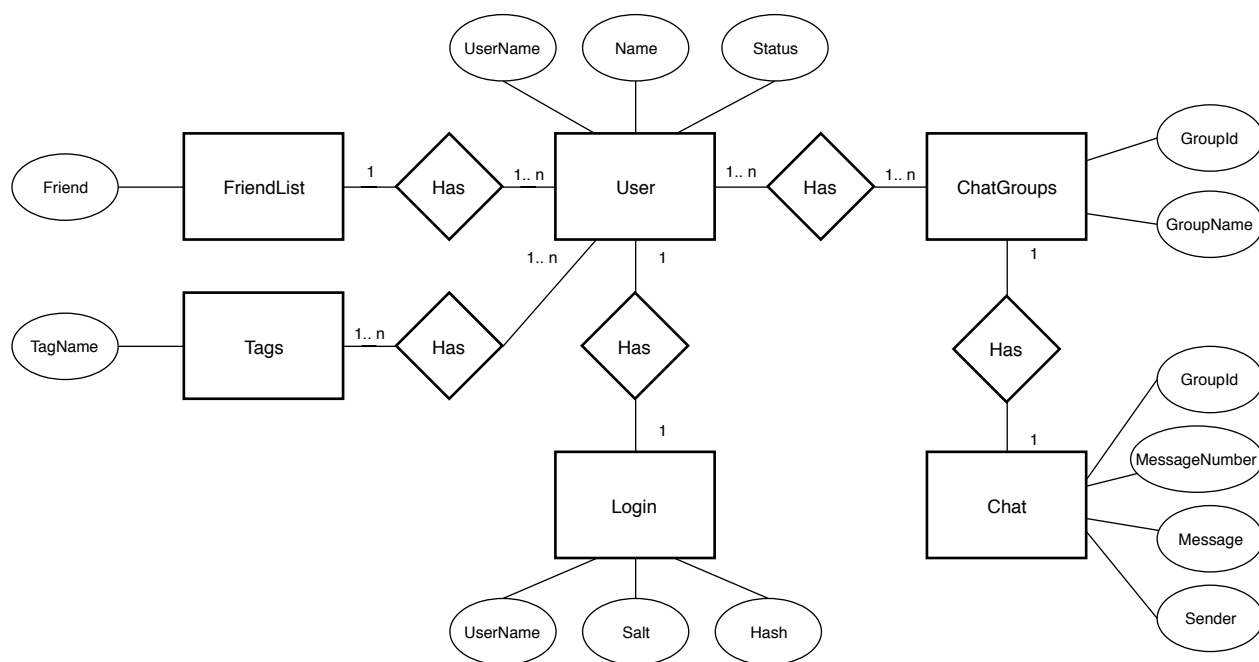
Til test af internetkommunikations delen af serveren, er der ikke opstillet unit tests med NUnit. Hver gang der er implementeret en ny feature, er denne funktion testet med tomme beskeder, forkerte beskeder og korrekte beskeder. Hvis denne feature opfører sig som tiltænkt, baseret på de beskeder den får, er den erklæret funktionel og der arbejdes på den næste feature.

## 9.3 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 også skal 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.

### 9.3.1 Analyse

For at få en bred forståelse for, hvordan databasen skal se ud, er der lavet et ERD for databasen. Dette diagram kan ses på figur 12. 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 12: 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 database. På den måde skal der ikke deles ændringerne frem og tilbage mellem flere computere.

### 9.3.1.1 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.

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).

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.

## 9.3.2 Design af databasen

I databasen er der lavet nogle tabeller, som hver har ansvaret for at opbevare en bestemt type data. Nedenunder ses det, hvordan de forskellige tabeller kommer til at se ud.

### 9.3.2.1 ChatGroups

Tabellen for ChatGroups holder styr på alle de chatgrupper, der er blevet oprettet på systemet. Et eksempel ses på tabel 5

GroupId	GroupName
1	Marto-Entutiaster
2	Ingeniørerne

Tabel 5: Database over ChatGroups

### 9.3.2.2 Chat

Tabellen for Chat holder styr på alle beskeder der er blevet skrevet mellem brugerne. Dette er inddelt i to dele. Der er både en chat mellem to brugere, og så er der en gruppechat.

Tabellen for chat mellem to brugere indeholder, hvem der er sender og modtager, samt selve beskeden. Et eksempel kan ses på tabel 6

Sender	Receiver	Message
Marto	Farto	Hej
Farto	Marto	Hva så?

Tabel 6: Database over Chat

Tabellen for gruppechatten minder meget om tabel 6, bortset fra at den indeholder gruppeId i stedet for en modtager. Et eksempel kan ses på tabel 7

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

Tabel 7: Database over Chat

Pointen med at holde på alle beskeder i databasen, er at hver bruger har mulighed for at se alle sine tidligere beskeder, samt hvem der har skrevet hver enkelt besked. Dette betyder også at du kan sende beskeder til en bruger der er offline. Denne bruger vil blot modtage beskederne næste gang brugeren går online.

### 9.3.2.3 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 tabel 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 8

emoji	emoji_shortcut
😊	:)
😞	:(

Tabel 8: Database over emojis

#### 9.3.2.4 Tabel for FriendList

Tabellen for FriendList indeholder informationer om statussen mellem de forskellige brugere. Det viser om to brugere er venner, blevet afvist, fjernet eller blokeret. Et eksempel kan ses på tabel 9. Den kolonne der hedder Action\_User, viser hvilken bruger der har lavet den sidste aktion af de to brugere.

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

Tabel 9: Database over FriendList

#### 9.3.2.5 Tabel for Login

Tabellen for Login indeholder alle Brugernavne, samt deres "saltede og hashede" adgangskode. Et eksempel kan ses på tabel 10

UserName	Salt	Hash
Marto	DDFERE134A	c412b37f8c0484
Farto	IUHJKVA331	362132fc05ba31

Tabel 10: Database over Login

#### 9.3.2.6 Tabel for Tags

Tabellen for Tags indeholder alle tags, der er tilføjet til systemet. Et eksempel kan ses på figur 11

TagName
Fodbold
Tennis
Tandlæge
CSGO

Tabel 11: Database over Tags

#### 9.3.2.7 UserInformation

Tabellen for UserInformation indeholder alle brugernavne, samt deres informationer. Her kan også ses om en bruger er online eller offline på systemet. Et eksempel ses på figur 12

<b>UserName</b>	<b>Description</b>	<b>Status</b>
Marto	...	Online
Farto	...	Offline
Darto	...	Online
Sarto	...	Online

Tabel 12: Database over UserInformation

### 9.3.2.8 UserChatGroups

En bruger kan være medlem af flere chatgrupper, samt kan en chatgruppe bestå af flere brugere. Her er der tale om en many-to-many relation. Derfor skal der laves en ekstra tabel til at holde styr på, hvem der indgår i en vilkårlig chatgruppe. Denne tabel kaldes UserChatGroups. Et eksempel på dette kan ses på tabel 14.

<b>GroupId</b>	<b>UserName</b>
1	Marto
1	Farto
2	Marto
2	Darto

Tabel 13: Database over UserChatGroups

### 9.3.2.9 Tabel for UserTags

Der er også et many-to-many relation mellem en bruger og et tag. Derfor laves der også en tabel til at holde styr på dette. Et eksempel på kan ses på tabel 14

<b>TagName</b>	<b>UserName</b>
CSGO	Marto
CSGO	Darto
Databaser	Marto
Tennis	Sarto

Tabel 14: Database over UserChatGroups

## 9.3.3 Implementering af databasen

For at kunne arbejde med en database, skal der først laves en connection til den. Dette gøres vha. en connectionString der tilføjes config filen i solutionen. Der bliver herefter oprettet klasser til at repræsentere hver tabel, som er med i databasen. Derefter laves der en DBContext, som sørger for at inkludere de forskellige klasser.

En yderligere forklaring af, hvordan dette er blevet oprettet kan ses i dokumentationsafsnittet for databasen

### 9.3.3.1 Repository

For at kunne opdatere tabellerne, herunder tilføje, slette og læse information, 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 indkapsle disse queries i en række 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 skal have sine egne metoder. Disse metoder er vigtige at bruge når man f.eks. har en many-to-many relationship mellem klasser.

For yderligere information omkring, hvilke klasser der er oprettet i repositoret, henvises der til dokumentationsdokumentet for database.

### 9.3.3.2 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 tilføjes en person og når der tilføjes en adresse. Her bruges i stedet Unit of work, så der kun bliver gemt, når alle de ønskede kommandoer er blevet fuldført.

#### 9.3.3.2.1 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.



### 9.3.3.3 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.

For yderligere information omkring Servercommunication, henvises der til dokumentationsdokumentet for database.

### 9.3.4 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 denne data.

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

For yderligere information om, hvordan der er blevet brugt unit testing, henvises der til dokumentationsdokumentet for database.

# 10 Accepttestspecifikation

Overordnet set er det lykkedes os at implementere og teste de mest essentielle funktionaliteter i Marto. De mest relevante user stories som "Sende tekstbesked", "Log ind", "Opret profil" og "Søg efter tag", er blevet prioriteret højere end andre user stories, da gruppen følte disse user stories definerede kernefunktionaliteten af programmet. Hovedmålet har derfor været at teste de funktionaliteter i programmet, der sørger for at der er kommunikation mellem brugerne.

På baggrund af accepttestene, kan det ses at en del features ikke er implementeret. Dette skyldes fejlvurdering af den mængde tid det ville tage at designe og implementere grund funktionaliteten af programmet. Gruppen har vurderet, at der ville være mere at vise frem af produktet, hvis kommunikationen mellem GUI, database og server var oppe at køre, og derfor er der blevet lagt en del user stories til side. En del af accepttestene har derfor fået opstillet en test der afspejler den måde som gruppen vil have stillet funktionaliteten og tests op på. Resultatet vil derfor blot være "Ikke implementeret".

For en mere detaljeret gennemgang af accepttest henvises der til dokumentationens sektion "Accepttest".

# 11 Resultater & Diskussion

Som det kan ses ud fra den accepttest der er opstillet på baggrund af projektets krav, virker størstedelen af den tiltænkte funktionalitet. Der er her features der er blevet udeladt, udelukkende grundet den tidsmæssige begrænsning der har været. Her blev kernefunktionaliteten i stedet prioriteret højere. Features som Voice Over IP og decisionmaker spillet blev en eftertanke undervejs i udviklingen, da det viste sig at implementeringen af grundfunktionaliteten, var et større arbejde end forventet. Her er der kommet en del features til fremtidigt arbejde. Grundet systemets opbygning og design, vil den sidste kernefunktionalitet ikke tage lang tid at implementere. Under kernefunktionalitet ligger muligheden for at fjerne en ven fra sin venneliste og ændre sit profilbillede. Det at slette sin egen profil ligger også herunder. At søge efter en anden profil er udgået, da featuren blev vurderet unødvendig på baggrund af, hvordan tilføj ven fungerer. Det er ikke muligt at ændre adgangskode eller indgå i en gruppechat og emojis er ikke implementeret endnu.

Produktet som helhed fungerer. GUI, server og database er i stand til at spille sammen, hvilket et funktionelt produkt med få mangler.

## 11.1 Personlige Konklusioner

### 11.1.1 Alexander

I dette projektforsløb har jeg taget rollen som Scrum-leder. Dette har bragt mig nogle nye udfordringer og opgaver end hvad jeg har været vant til i løbet af andre projektforsløb. Det har givet mig mere indblik i hvordan Scrum-projektledelses metoden skal anvendes og hvordan man kan opfordere de andre medlemmer i gruppen til også at anvende Scrum-ledelses metoden. Det er første gang i et projektforsløb at jeg har ansvaret for at indkalde til møder, hvilket jeg også synes har fungeret efter planen.

Et chatprogram er et produkt jeg selv hyppigt anvender, og selve det at produktet er noget man synes er interessant synes jeg hjælper utroligt meget med den overordnede interesse i projektet. Det styrker viljen til at arbejde effektivt på projektet og hjælper samtidig også med at give en et overblik da man allerede ved hvordan produktet skal fungere.

Projektet har hjulpet mig meget med at se sammenhængene mellem de fag jeg har haft i løbet af 4. semester. For det meste arbejder vi med fagene enkeltvis gennem afleveringer i løbet af semesteret, og det kan derfor være svært at overskue hvordan man kobler det man arbejder med i fagene sammen. Semesterprojektet virker derfor som en milesten hvor man endelig får lov til at anvende det man har lært i de enkelte fag og sætte det hele sammen. Det kan ofte være dette der hjælper en med at forstå det brugbare ved fagene.

### 11.1.2 Daniel

Det har været et fedt projekt med nogle fede ideer, og det har givet en god forståelse for at et team med god synergi er et vigtigt element for at et så stort projekt skal gå godt.

Min rolle som projektleder skulle jeg stå for nogle af de mere administrative opgaver og underliggende var jeg udvikler med et par dygtige kollegaer på server siden. Jeg stod for klient delen og når der var noget jeg ikke kunne forstå, så blev der taget hånd om det, effektivt og hurtigt.

Fagligt har jeg altid brug for mere læring i alt, men jeg synes jeg har fået god forståelse for de fag vi har haft på studiet og praktisk brug af dem igennem semesterprojektet.

Det var fedt som projektleder at prøve at håndtere de forskellige problematikker og uddeligere opgaverne omkring de forskellige kollegaer såvel at holde styr, med hjælp fra stand up møder, på udviklingen i projektets gang, men man skal altid huske på at det er ens kammerater og vi er ikke helt udviklet software udvikler, men det hæmmer ikke for ambitionerne.

Jeg er stolt af min gruppe og glad for det produkt der blev kreeret ud fra den process vi har haft igennem projektet.

### 11.1.3 Fatima

Projektet har været spændende idét det er et projekt hvor man virkelig lærer at bruge sin viden fra alle kurserne og anvende det som var man i et team hvor man skal udvikle et produkt. Det er allerede givet at der skal arbejdes med de tre hovedfag, men naturligt kommer de to andre fag ind (SWD og SWT). Jeg har især opdaget at designet af et system er enormt vigtigt for at sikre at kvaliteten af koden er optimal.

Jeg har arbejdet med GUI-delen af projektet, hvor der er fokuseret på at opdele koden og arbejde ud fra et design pattern. På denne måde er det nemt at opdele arbejdet. Det specifikke design pattern er blevet introduceret i kurset, men vi har yderligere fundet ud af, hvordan det anvendes og hvad de egentlige fordele og ulemper er ved det. Dette er altid en spændende og lærerig del af et projekt.

Der er anvendt SCRUM i forløbet, men jeg har ikke skabt mig nogle nye erfaringer deraf, da jeg har anvendt SCRUM i tidligere forløb.

Projektet har desuden været yderligere interessant, da projektet er bestående af IKT'ere hvor vi har kunnet få lov til at fokusere på et produkt som er rent software, hvilket har gjort det nemt for alle medlemmerne at sætte sig ind i de forskellige dele af projektet. Jeg synes der har været et godt samarbejde og en god kemi på tværs af de individuelle grupper hvilket også er en stor del af grunden til at projektet blev en success.

### 11.1.4 Frederik

Min rolle i projektet har været udvikler ligesom de andre gruppemedlemmer, men fik også titlen process leder. Process lederens job er at sørge for gruppens medlemmer trives med hinanden. Dette har ikke været noget problem da alle i gruppen kender hinanden fra tidligere og derfor trives godt med hinanden.

Fagligt har jeg skabt mig erfaring, med de overvejelser der har været, når man skal designe og implementere en server. Har arbejdet meget med at få serveren til at spille sammen med databasen, så er blevet bedre til at skabe sammenspil mellem større moduler. Her har jeg skabt erfaring med, at kommunikere hvad der er krævet af databasen, for at serveren kan fungere optimalt med databasen.

Processdelen af projektet er udført ved brug af SCRUM og en iterativ udviklingsprocess. Dette har jeg skabt erfaring med på tidligere semestre også, så dette var ikke nyt.

### 11.1.5 Martin

Dette projektforløb har været ret interessant. For første gang har vi i gruppen kunne lave et produkt præcist som vi ønskede det, og med et chatprogram har vi arbejdet med at få en bred forståelse for mange interessant områder.

Ved at arbejde på serverdelen af systemet, har jeg arbejdet med det emne jeg har haft mest interesse

for. Her er der brugt viden fra tidligere projekter, sammen med viden fra IKN og helt ny selvfundet viden, bl. a. indenfor sikkerhedsområdet. Dette har været fedt at endelig have en grund til at lære noget om, og det har været endnu federe at rent faktisk implementere det.

Dette projekt er gået rigtigt godt, med seriøst samarbejde og meget lidt stress. Det har været en fryd at se de forskellige dele af systemet blive udarbejdet, og at se det hele virke sammen har efterladt en god følelse efter et veludført gruppearbejde.

#### **11.1.6 Søren Bech**

Der har fra start af været en del krav til hvad projektet skulle inkludere, da alle fag på dette semester skulle inddrages. Det har derfor været udfordrende at skulle finde på et projekt der ville indholde alle disse krav, men det har også givet et realistisk billede af, hvordan det er at skulle arbejde på et projekt i arbejdsmarkedet.

Det har især været fedt med at man hver har haft et fokuspunkt til at fokusere på, og derfor få det til at spille sammen med alle andre moduler. Jeg har selv arbejdet på databasen og derefter fået det til at spille sammen med serveren.

Vi har også i dette semester som noget nyt arbejdet med unit tests, som jeg har haft godt nytte af, da man kan finde ud af om de forskellige moduler i databasen virker, uden at skulle tilslutte det sammen med serveren. Derfor vidste man, at hvis serveren og databasen ikke fungerede sammen som forventet, var det integrationen mellem disse der var noget galt med, da man allerede havde testet om modulerne virkede for sig selv.

#### **11.1.7 Søren Schou Mathiasen**

I dette projekt har jeg, arbejdet med grafisk brugergrænseflade, hvilket jeg har indfundet mig godt med. Jeg har arbejdet tæt sammen med Fatima selvom, det har været tydeligt, at vi har haft forskellige fokuspunkter. Jeg har også været meget ind over under integration med server, og har derfor også en rimelig forståelse for især klienten, og selve kommunikationen mellem klient over server.

På trods af at jeg lovede mig selv sidste projekt, at jeg ville påbegynde dokumentation tidligere, føler jeg ikke, at jeg har opfyldt dette til mit fuldeste. Selvom jeg er påbegyndt noget forholdsvis tidligt, er det stoppet lidt halvvejs igennem. Det vil jeg forsøge at rette op på, hvis jeg ender i samme situation igen.

Scrum har vi også anvendt, og jeg har ikke været den stærkeste i det, da jeg mener, at der bliver lagt mere krudt i at organisere ting, end der gør på rent faktisk at lave noget. Til et større projekt ville jeg have noget nemmere ved at se idéen i det. Daily Scrum og ugentlige scrum møder har nu været meget praktiske.

Alt i alt er jeg godt tilfreds med min arbejdsindsats, men der kan altid forbedres.

# 12 Konklusion

## 12.1 Projektarbejdet

Det kan konkluderes at størstedelen af de planlagte features virker som tiltænkt på Marto. De vigtigste hovedfunktionaliteter, som oprettelse af bruger, venner og chat virker som tiltænkt, og det var disse, der har været prioriteret højest. Der mangler noget funktionalitet, som at kunne fjerne en ven, ændre profil og mere, men disse har været nedprioriteret da tiden har været kort.

Den store mængde features, der fra start havde været planlagt, viste sig at være for ambitiøse i forhold til den tid der har været til rådighed. Gode prioriteringer har derfor været en nødvendighed og har gjort det muligt at præsentere et færdigt produkt med bæredygtig funktionalitet.

Server og GUI delen af projektet har vist sig at have været et større projekt end først antaget. Gruppen mistede en mand i starten af projektførelsen som gruppen kunne have gjort brug af. Her kunne der med fordel have været en ekstra mand på GUI-delen af projektet.

## 12.2 Procesarbejdet

Projektstyringen er drevet af SCRUM og en iterativ udviklingstilgang kaldt AGILE.

Ved hjælp af SCRUM blev der lavet en tidsplan og nogle faser der gav et lille overblik over hvad gruppen skulle lave til næste møde der blev afholdt.

Det er dog ikke altid gruppen har fået maksimal udbytte af SCRUM. Dette skyldes at projektet er sideløbende til 5 andre fag, der også fylder rigtig meget. Derfor var det ikke fremmed for gruppen, at gruppens medlemmer ikke havde udført noget arbejde på projektet, fra det ene daily SCRUM til det andet. Her føles det lidt som en pligt at udføre SCRUM, i stedet for at være et værktøj der skulle fremme udviklingen og overskueligheden.

# 13 Fremtidigt arbejde

Da dette projekt har haft en kort periode, har der været nogle ting, som er blevet sorteret fra, men som kunne være godt at have implementeret, hvis der havde været mere tid.

Nedenunder kan ses de features der kunne blive arbejdet på i fremtiden.

- Lave base funktionaliteten færdig
- Bedre GUI skalering
- Tooltips til vejledning i GUI
- DecisionMaker
- Seperere DB fra server
- Server scalability
- VOIP
- Send billede
- Emojis
- Filer



# Bibliography

[1] Server <https://stackoverflow.com/questions/6187456/tcp-vs-udp-on-video-stream>