Ontwikkeling van Oghma Chat, Android

Jan van de Molengraft, Thijs Werrij, Niek Janssen, Wouter van der Linde

27-06-2014

Inhoudsopgave

1 Voorwoord

Wat volgt is de documentatie over het ontwikkelen van de app Oghma Chat voor Android. Wij, team ProjectArteDerp, hebben deze app gemaakt in het kader van het vak Research and Development, dat deel uitmaakt van de bachelor Informatica aan de Radboud Universiteit Nijmegen. Dit document is bedoeld om inzicht te geven in hoe wij de ontwikkeling van Oghma Chat Android aangepakt hebben, wat er goed ging en wat niet, en wat we als team van dit proces geleerd hebben.

De opbouw van dit document is als volgt: we beginnen met een beschrijving van het product (de app Oghma Chat voor Android), waarin we uitleggen waarom we deze app ontwikkeld hebben, wat de app precies kan en in de toekomst nog moet gaan kunnen, en waarom Oghma Chat iets toevoegt aan de vele al bestaande chat apps, zoals Whatsapp, Facebook messenger en Skype.

Na deze beschrijving van de app gaan we in op het ontwerp. We beginnen dit onderdeel met een globale beschrijving van de opbouw van de app, waarin we het ontwerp zullen opsplitsen in drie delen. Deze drie delen werken we vervolgens in detail uit, om een goed beeld te geven van hoe Oghma Chat voor Android functioneert. We sluiten de beschrijving van het ontwerp af door te verantwoorden waarom we voor dit ontwerp gekozen hebben.

In het laatste hoofdstuk reflecteren we op hoe het ontwikkelen van de app in onze ogen gelopen is. We zullen de tegenslagen en successen bespreken, en als team aangeven wat we ervan vonden om aan dit project te werken en wat we ervan geleerd hebben.

2 Beschrijving

2.1 Inleiding

Oghma Chat voor Android is een android applicatie die aansluit op het Oghma Chat systeem van Project ArteMisc. Dit geeft het meteen een duidelijke specificatie: het moet met het volledige Oghma Chat systeem overweg kunnen. Voor de huidige staat van het systeem betekent dit dat de app moet kunnen omgaan met Contacten (denk aan contacten toevoegen, zoeken, verwijderen etc.) en het versturen van berichten volgens de specificaties van het systeem (exclusief berichten verzenden van gebruiker A naar B als A en B volledige contacten zijn).

Ter illustratie zijn hier screenshots van de mobiele en web applicatie:

2.2 Specificaties

2.2.1 Functional Requirements

We behandelen hier alleen de functionaliteit die de app volgens ons moet hebben aan het eind de acht weken die beschikbaar waren voor het vak Research and Development. Hierdoor zullen sommige requirements waarvan we willen dat de app ze uiteindelijk wel heeft, voor nu niet behandeld worden. Daarnaast gaan

we ervan uit dat de gebruiker al ingelogd is in de app (behalve bij de use case 'Inloggen' natuurlijk).

2.2.1.1 Use Case Inloggen Om gebruik te kunnen maken van de app, moet deze inloggen. Dat gebeurt aan de hand van een inlogscherm dat verschijnt als de app opstart. Hier heeft de gebruiker een valide Project ArteMisc gebruikersnaam/wachtwoord combinatie voor nodig.

2.2.1.1.1 Actors

- De gebruiker

2.2.1.1.2 Preconditions

- Wifi of 3G (of een andere vorm van mobiel internet) is actief op het mobiele apparaat dat de gebruiker gebruikt.
- De app is opgestart

2.2.1.1.3 Happy Path

- De gebruiker start de app door op het icoontje te drukken.
- Nadat de app is opgestart, voert de gebruiker zijn gebruikersnaam en wachtwoord in in de daarvoor bestemde tekstvakjes.
- De gebruiker drukt op de knop inloggen onder de tekstvakjes.
- De app start de MainActivity, en de gebruiker kan vervolgens de app gebruiken.

2.2.1.2 Use Case Bericht Verzenden Als een gebruiker een bericht in typed en op verzenden drukt, moet dat bericht ook daadwerkelijk naar de server gestuurd worden en daar op de goede manier verwerkt worden.

2.2.1.2.1 Actors

- De gebruiker

2.2.1.2.2 Preconditions

- Wifi of 3G (of een andere vorm van mobiel internet) is actief op het mobiele apparaat dat de gebruiker gebruikt.
- De gebruiker heeft valide inlog credentials.

2.2.1.2.3 Happy Path

- De gebruiker drukt op het tekstvakje rechtsboven in beeld.
- De gebruiker selecteerd de tab recente gesprekken / De gebruiker selecteerd de tab contactpersonen / De gebruiker selecteerd de tab groepen.
- De gebruiker drukt op het gesprek / de persoon / de groep waarin / waarnaar hij een bericht wil sturen.
- De gebruiker drukt op het tekstvak onderaan in beeld.
- De gebruiker typed zijn bericht in.
- De gebruiker drukt op verzenden.
- De app stuurt het bericht dat de gebruiker ingevoerd heeft naar de server.

2.2.1.3 Use Case Contact Toevoegen Gebruikers kunnen contacten toevoegen aan hun contactenlijst.

2.2.1.3.1 Actors

- De gebruiker
- Een andere gebruiker

2.2.1.3.2 Preconditions

- Wifi of 3G (of een andere vorm van mobiel internet) is actief op het mobiele apparaat dat de gebruikers gebruiken.
- De app is opgestart bij beide gebruikers

2.2.1.3.3 Happy Path

- De gebruiker drukt op het tekstvakje rechtsboven in beeld.
- De gebruiker selecteerd de tab Zoeken.
- De gebruiker typed de naam in van de gebruiker die hij wil toevoegen aan zijn contactenlijst.
- De gebruiker selecteerd de button om naar het profiel van de toe te voegen gebruiker te gaan.
- De gebruiker dukt op de knop ‘Add user’.
- De andere gebruiker krijgt een melding dat hij is uitgenodigd door de gebruiker om toegevoegdte worden aan diens contactenlijst, en accepteerd deze.

2.2.2 Non-Functional Requirements

- Security

  privacy moet bewaard blijven als er messages heen en weer gestuurd worden door users. Dit wordt gedaan dmv SSL/TLS connecties, waarmee berichten van en naar de server niet afgeluisterd kunnen worden. Belangrijker nog: wij slaan geen gegevens op die wij niet nodig hebben. Zo doen wij niet aan user usage tracking of targeted user analysis. Ook heeft het systeem geen reclames.

- Robustness

  Messages die door de client (de Android app) verstuurd worden, moeten bij de eerste mogelijkheid (als er een connectie is met de server) naar de server worden verstuurd. Tevens moeten we zeker weten dat ze aan zijn gekomen. Dit gebeurt door onderliggend TCP connecties te gebruiken, en een interne buffer die bijhoud welke berichten nog verstuurd moeten worden.

2.3 Productverantwoording

Oghma Chat is een chat app, met als basis de functionaliteit die in principe in elke chat app te vinden is. Toch zijn wij er van overtuigd dat onze app een bepaald gat in de markt vult, omdat het is gemaakt vanuit een (door anderen bevestigde) ontevredenheid over het huidige aanbod:

- Het merendeel van de huidige chat apps zijn niet intuitief/makkelijk te gebruiken.
- Veel apps zijn niet veilig in gebruik. Een duidelijk voorbeeld hiervan zijn What'sapp en Telegram. Deze chat apps gebruiken beide in-house ontwikkelde beveiligings protocollen, waar aardig wat lekken in zitten.
- Veel chat apps zijn niet of nauwelijks te customizen, en de functionaliteit die een chat biedt is hier vaak zeer beperkt door.
- Veel chat apps bieden geen belofte over het naleven van privacy-gerelateerde wetten. Dit omdat het breken van privacy vaak een middel is om geld te verdienen.

De 4 bovengenoemde punten zijn de belangrijkste punten achter het (systeem)ontwerp van onze applicatie. Er zijn veel apps die mogelijk voor één, misschien twee van de bovenstaande problemen een oplossing bieden, maar vooral punt 3 en 4 zijn ver te zoeken in het huidige aanbod van apps.

Wij hebben een systeem ontwikkeld dat in de toekomst aan de 4 bovenstaande eisen moet gaan voldoen (door de korte tijsspanne van dit project zijn we eigenlijk alleen toegekomen aan de basisfunctionaliteit van de app), en zijn er derhalve van overtuigd dat onze app een positieve toevoeging is aan het huidige marktaanbod. Het bouwen van de Android app is een belangrijke stap

naar het aanspreken van een groter publiek met ons chat systeem; het zorgt voor een meer gebruiksvriendelijke ervaring voor mensen die van het systeem gebruik willen maken op hun smartphone.

2.4 Namespaces

Als je inlogt bij Project ArteMisc, log je in met je account. Aan dat account zijn (mogelijk) meerdere namespaces gelinkt (denk bij namespaces aan gescheiden omgevingen voor bijvoorbeeld je gamevrienden, collega's of familie), en bij elke namespace hoort een User. Je kunt dus met één account meerdere Users hebben.

Het doel van namepaces is om een privacy vriendelijke omgeving to creëeren, die de eigenaar kan configureren zoals hij of zij wil. De eigenaar kan ervoor kiezen om een namespace open te maken (beschikbaar voor iedereen, bijvoorbeeld geschikt voor gameclans), of gesloten (dus alleen toegankelijk voor personen die hij of zij uitnodigd, bijvoorbeeld geschikt voor bedrijven), of een mix daartussen. Op deze manier is het mogelijk om met één Project ArteMisc account de chat met je collega's gescheiden te houden van de chat met je vrienden. Bovendien zijn we van plan om in de toekomst custom bots te ontwerpen die je kunt toevoegen aan je namespace (niet aan de openbare namespace, waar een gebruiker standaard in zit).

Deze namespaces zijn in deze app nog niet erg ver uitgewerkt, maar in de code zijn al sporen te vinden van voorbereidingen op de implementatie ervan. Op het moment zullen alle users alleen toegang hebben tot de standaard namespace. Dit is de namespace die vrij te gebruiken is en waar deze chat applicatie net zo functioneert als andere chatapplicaties.

Omdat de term namespace vrij veel voorkomt in dit verslag, leek het ons een goed idee om even uit te leggen wat het concept erachter is. Namespaces zijn één van de componenten die Project ArteMisc een succes moeten maken: ze zijn een belangrijk onderdeel van het systeem.

3 Ontwerp

3.1 Globaal ontwerp

We hebben de app al snel opgedeeld in drie grote modules. Deze zijn ten eerste de GUI (Graphical User Interface), ten tweede de Server, en als laatste de module Communicatie. Elke module is uitgewerkt door één of twee teamleden: Niek Janssen en Thijs Werrij waren verantwoordelijk voor de GUI, wat het visueel ontwerp van de app inhoud. Jan van de Molengraft heeft zich vooral beziggehouden met het implementeren van de Server, maar heeft daarnaast nog erg veel werk gedaan als teamleider en heeft de rest van het team geholpen bij problemen. De module communicatie was toegewezen aan Wouter van der Linde. Dit onderdeel is de schakel tussen de client (de app) en de server: het zorgt er bijvoorbeeld voor dat als een gebruiker van de app een bericht invoert en op de knop 'verzenden' drukt, dat bericht ook daadwerkelijk naar de server

gestuurd wordt, en dat berichten die de server naar de app stuurt op de goede manier binnenkomen en worden behandeld door de app.

Deze verdeling van de onderdelen zorgde ervoor dat iedereen kon werken aan zijn eigen onderdeel, zonder erg afhankelijk te zijn van code van de andere teamleden. Bovendien was het door deze indeling niet bijzonder lastig om de afzonderlijke modules, toen deze voor een groot deel af waren, in elkaar te schuiven: bij het ontwikkelen van de GUI was het niet nodig om een connectie te hebben met de server, dummy data was voldoende om te testen. En toen alle knoppen en tekstvelden gecrëerd waren, was het in principe geen probleem om daar vervolgens de functionaliteit uit het onderdeel Communicatie aan te verbinden.

Het loskoppelen van het onderdeel communicatie en de server was wat lastiger: de data die de app naar de server moet sturen, moet begrepen worden door de server, en andersom. Een nauwe samenwerking tussen deze twee onderdelen was daarom belangrijk, maar voor het grootste deel was dit probleem opgelost door het opstellen van een specificatie over hoe berichten opgebouwd en verstuurd moesten worden.

3.2 Detailontwerp

3.2.1 GUI

In onze code-structuur hebben we ervoor gekozen om de gui en de backend strict van elkaar gescheiden te houden. Dit doen we d.m.v. een mvc model. De views zitten voor het grootste gedeelte in de xml files, maar af en toe ook een beetje in de gui klassen. De controllers zitten geheel in de java klassen van het package 'gui'. De models zitten in weer andere packages. Hierover meer in 3.2.3.

3.2.1.1 Fragments We hebben bij onze app gebruik gemaakt van fragments. Hierdoor gebruiken we maar twee activities: de LogInActivity en de MainActivity. De app start altijd in de LogInActivity. Deze activity is voorzien van tekstbalkjes om je inloggegevens in te vullen en een knop. Bij een succesvolle log-in gaat de app via een intent naar de MainActivity.

Hier maakt onze app gebruik van fragments. We doen dit door middel van een URI-achtige string: deze string lijkt op het adres in de adresbalk van een browser als hij naar een website navigeert. De app gebruikt deze dus om door de verschillende fragments te navigeren. Dit kan de gebruiker doen door de links in de drawers te gebruiken. De bestaande fragments in onze app zijn:

3.2.1.1.1 Home Met de URI '/home' navigeer je naar het homefragment (FragNavHandlerHome). Dit is de fragment die automatisch geopend wordt als je van de LogInActivity naar de MainActivity gaat. Dit fragment bevat jouw eigen gebruikersprofiel: je user icon, je volledige naam en je username. Ook staat hier een ListView waarin berichten te zien zijn.

3.2.1.1.2 User Het user fragment (FragNavHandlerUser) toont een profiel. De fragment heeft hiervoor een userid nodig, die door middel van de URI wordt doorgegeven in de vorm '/user/[namespace]/[userid]'. Afhankelijk van de situatie zijn er verschillende soorten knoppen beschikbaar:

- Als je een profiel bezoekt van iemand die nog niet in je contacten staat en wie je ook nog geen uitnodiging hebt verstuurt, verschijnt er een knop 'Add user' om deze gebruiker aan je contactlijst toe te voegen. Als je iemand hebt uitgenodigd, maar die gebruiker heeft hier nog niet op gereageerd, krijg je dezelfde knop te zien, maar deze knop is dan disabled.
- Als een gebruiker al in je contactenlijst staat, is er een knop 'Remove user' om die persoon uit je lijst te verwijderen.
- Als een gebruiker jou een uitnodiging heeft gestuurd om jou aan zijn/haar contactlijst toe te voegen, dan verschijnen er twee knoppen; een om het verzoek te accepteren (Accept) of af te wijzen (Decline).

3.2.1.1.3 Room Dit fragment is een chatroom, bestaande uit een ListView, een tekstvak en een knop. Hier kan een gebruiker berichten ontvangen van andere gebruikers. Een room kan gebruikt worden door individuele gebruikers of een groep. Als de gebruiker een bericht intikt in het tekstvak en op de knop drukt, wordt het bericht verstuurd aan de andere gebruikers in de room.

3.2.1.1.4 Group Dit fragment toont groeppagina's. De URI lijkt op die van de user fragment, namelijk '/group/[namespace]/[groupid]'. Deze haalt de groepsicoon, groepsnaam, de beschrijving en de leden van de groep op die bij de groupid horen. De beschrijving is uitvouwbaar door een tap. Deze kun je hierna ook weer inklappen.

3.2.1.2 Drawers Een van de moeilijkheden die we tegenkwamen waren de drawers. Het automatisch gegenereerde project bevatte een hele hoop automatisch gegenereerde code waarvan we geen idee hadden wat het deed. Daarnaast wilden we graag twee drawers: aan beide zijden van het scherm één. Dit was mogelijk, maar de buttons die samenwerkten met de drawers begonnen raar gedrag te vertonen. Uiteindelijk besloten we deze buttons gewoon zelf te schrijven. Dit bleek uiteindelijk niet al te moeilijk, maar intussen hadden we al veel tijd verloren. Maar we hebben nu twee drawers met werkende buttons. Aan de linkerzijde zit nu alleen nog een logout knop. Later worden hier meer items aan toegevoegd zoals settings, namespacebeheer (Zie: 2.4) en accountbeheer. Aan de rechterkant bevindt zich een tabhost met daarin vier tabs:

- Recent conversations
- Contacts
- Groups

- Search

In de eerste drie tabs verschijnen alleen je eigen contacten en groepen. Iin de tab Search verschijnen ook de users die je nog niet toe hebt gevoegd.

3.2.1.3 Tabhost De tabhost is een widget in de standaard android libraries die het mogelijk maakt tabs in een fragment te krijgen. Dit doe je door de tabhost per tab een string of image en een klasse mee te geven. Een vereiste van deze klasse is wel dat deze Fragment extend. Wij hebben ervoor gekozen om vier keer dezelfde klasse mee te geven en vervolgens met getTag() erachter te komen welke tab nu eigenlijk actief is. Zodra we dit weten wordt er beslist:

- Welke view er inflated wordt
- Of de vlag bigItems wordt geset (Alleen in de tabs recent en search)
- Welk bericht er wordt weergegeven bij een lege dataset
- Of de groupsbutton wordt geïnitialiseerd (Alleen in tab Groups)
- Of de searchbar wordt geïnitialiseerd (Alleen in tab Search)

Deze klasse zorgt er ook voor dat de data wordt opgehaald en aan de listview wordt doorgegeven. Door tijdsgebrek update deze klasse helaas niet bij server events. Dit zou later geimplementeerd kunnen worden door de interface GUIUpdatable te implementeren. Meer hierover in 3.2.1.5

3.2.1.4 ListViews Listviews zijn de android widgets die ervoor zorgen dat grote lijsten aan data efficient worden weergegeven. Dit wordt gedaan met een ListAdapter. We hebben geprobeerd deze listadapter zo efficient mogelijk te implementeren. Dit was niet bijzonder moeilijk, want in tegenstelling tot het grootste deel van android was deze documentatie wel duidelijk. Dit kwam vooral door een video over dit onderwerp.

We hadden in totaal twee adapters nodig: een voor contacten/groepen en een voor messages. Dit hebben we opgelost door een abstracte klasse te maken die het meeste implementeerd en de subklassen hiervan konden de overige functies implementeren.

Het meest ingewikkelde van deze ListAdapters is het weergeven van verschillende soorten views door elkaar, en voor deze views de goede data ophalen. Het mixen van verschillende soorten views wordt gedaan met de methode getViewTypeCount() en getItemViewType(int pos). Met getViewTypeCount() wordt het aantal verschillende soorten views opgevraagd en met getItemViewType(int pos) wordt een integer 0 <= x <getViewTypeCount() opgevraagd. Deze wordt bepaald aan de hand van de type klassen in de List met data.

Vervolgens wordt getView geïmplementeerd. Aan deze functie wordt de positie en een view van het goede type meegegeven. Als deze view null is, wordt verwacht dat je een nieuwe view inflate. Als deze al bestaat wordt verwacht dat je hem hergebruikt. Op deze manier vergroot de efficientie van de listview met zo'n 300%.

3.2.1.4.1 Self-refreshable views De views die gebruikt worden zijn subklassen van LinearLayout en implementeren de interface gui.ListView. Deze interface vereist dat de methode refreshData geïmplementeerd wordt. Deze klassen halen zelf de overige benodigde data op. Op deze manier houden we de listAdapters zelf overzichtelijk. Verder zitten alle listview item klassen als volgt in elkaar:

- Elke klasse heeft een constructor die de parameters Context en AttributeSet accepteren. Dit is de constructor die gebruikt wordt door de inflater.
- Vervolgens Override elke klasse de methode onFinishInflate. In deze methode worden alle views die data bevatten opgeslagen in variabelen. Op deze manier hoeft deze data maximaal ongeveer 7 keer (het aantal items dat er ongeveer in de listview past) opgehaald te worden met findViewById. Dit voorkomt dat bij elke refreshData opnieuw gezocht moet worden naar deze views.
- Ten slotte implementeerd elke klasse de methode refreshData. Deze ontvangt als parameter één object. De meeste klassen kunnen dit object direct casten naar een object uit de package model (Zie **??**). De klasse die controller speelt voor de view die contacten/groepen weergeeft moet echter meerdere klassen ondersteunen.

  Deze refreshData methode haalt vervolgens de benodigde overige data op. Een voorbeeld hiervan is de controller voor de messageview. Deze zal een object van het type User op moeten halen om de username boven de message te kunnen weergeven.

  Uiteindelijk update deze methode alle views met de nieuw verkregen data.

3.2.1.5 Reacting to server events Natuurlijk zal de server ook events naar de client pushen, bijvoorbeeld wanneer er een message naar de user wordt verzonden. De service zorgt ervoor dat deze gegevens correct worden verwerkt. Vervolgens kan op de MainActivity de functie onServerEvent() worden aangeroepen met als parameter een Bundle. Deze wordt vervolgens doorgestuurd naar alle gui componenten die op dit moment actief zijn.

Deze componenten moeten zelf aangeven dat ze deze updates willen ontvangen door de functie subscribe (van GUIUpdatable) aan te roepen. Als GUIUpdatable geven ze over het algeeen gewoon zichzelf mee. Als dit component vervolgens door het systeem afgebroken wordt kan het unsubscribe (GUIUpdatable) aanroepen.

3.2.2 Server

De server van Oghma Chat is helaas te complex en deels ongerelateerd aan dit project. Toch zullen we proberen een zo duidelijk mogelijke weergave te geven van de rol die de server speelt in de Android versie van de app.

De server is op te delen in verschillende onderdelen:

- API server
- WebSocket server
- WebSocket Event Handlers (volgens eigen protocol)
- Event Logs

We zullen bespreken hoe deze onderdelen een link hebben met de Android app.

3.2.2.1 API server De API server reageert op HTTP requests van de app. De enige URI die relevant is voor de app is '/login'. Bij een request naar Login stuurt de app een door de gebruiker ingevoerde Username/Password combinatie in. Als deze combinatie overeen komt met een bestaand account binnen Project ArteMisc wordt er voor de user een session aangemaakt op de server. Deze wordt vervolgens in de vorm van een set Token cookies aan de app teruggestuurd. Hiermee kan de user zich vervolgens authenticeren tegen de server, zonder nogmaals een username/ wachtwoord in te moeten voeren. Dit wordt verder door de app afgehandeld (Zie het Session model in 3.2.3.1 voor meer uitleg).

3.2.2.2 WebSocket Server De WebSocket Server is het deel van de server dat verantwoordelijk is voor het accepteren van connecties die vanuit de client geïnitialiseerd worden. Bij het maken van de connectie hoort (in onze app) een stukje user authenticatie. Bij het maken van de connectie moeten de cookies (die ontvangen zijn in een eerdere request naar de API server op '/login') worden meegestuurd, zodat de server weet van wie de connectie is, en of die user ook daadwerkelijk ingelogd is. De rest van het onderdeel WebSocket Server wordt besproken in het stuk over Event Handlers op de server, en in het onderdeel Data Handlers van de Android app.

3.2.2.3 WebSocket Event Handlers WebSocket Event Handlers zijn de server side controllers (in de context van MVC). Om te zorgen dat de data makkelijk van client naar server en terug gaat, zowel in de Android app als in de web app, hebben we gekozen zelf een protocol te ontwikkelen dat onafhankelijk van de taal die we moeten gebruiken makkelijk te verwerken is. Dit is een voornamelijk text-based protocol, dat gebruik maakt van JSON data objecten om op een gestructureerde manier data heen en weer te sturen. De format van een protocol is vrij simpel, namelijk:

<EventName >:<JSON Object >

<EventName >kan elke combinatie van alfanumerieke karakters zijn, <JSON Object >kan alles zijn dat voldoet aan de specificatie van een JSON object. Wat er daadwerkelijk gebeurt op de server als er een event binnenkomt is buiten de scope van dit verslag.

Tevens kan de server kiezen om een event naar de client te sturen. Dit is bijvoorbeeld het geval als er een personal message verstuurd word door een andere user die de actieve user moet kunnen zien.

3.2.2.4 Event Logs Een belangrijk deel van de server, dat een zeer grote rol heeft gespeeld in het ontwerpen van de Android app, is het principe van Event Logs. Elke keer dat de server een event naar een client probeert te sturen (web of Android) wordt er een Event Log opgeslagen, waarin op een door de server begrijpbare manier beschreven staat over wat voor event het gaat, wanneer deze verstuurd had moeten zijn en welke data (voor het JSON Object bijvoorbeeld) ermee geassocieerd is. Zo kan de server bij het connecten van een user de gemiste events afspelen mits de client hier via het protocol (de INIT procedure) interesse voor uitspreekt.

We hebben gekozen voor deze aanpak omdat in de meeste gevallen het een flink significante hoeveelheid resources scheelt om alleen de gemiste events af te spelen voor de client, tegenover het volledig herladen van alle data van de server elke keer dat een connectie wordt gemaakt. Dit heeft vooral invloed op gebruikers die veel actief zijn, gebruikers die niet al te regelmatig contacten en/of groepen toevoegen/verwijderen, en voornamelijk: mensen die op een (slechte/instabiele) internet connectie zitten zoals 3G op een Android smartphone.

3.2.3 Communicatie

Om het ontwerp van het deel Communicatie duidelijk uit te leggen, breken we dit onderdeel op in de volgende drie delen:

- Datamodellen

  Alle data die de gebruiker invoert of die de app ontvangt van de server (denk aan nieuwe berichten, contactpersonen etc.) hebben we, in Object Geöriënteerde stijl, gemodelleerd met behulp van klassen. We leggen uit welke datamodellen we hebben, wat ze bevatten aan informatie en functionaliteit, en hoe we de lokale database opgebouwd hebben waarin we onze data opslaan.

- Data Handlers

  Voor onze app is het noodzakelijk dat we zowel data kunnen verzenden als ontvangen. Dit onderdeel beschrijft hoe we de datastromen binnen de app ontworpen hebben, zowel van de app naar de server als andersom.

- Transport

  In dit derde en laatste onderdeel van Communicatie bespreken we hoe we berichten uitwisselen met de server. We gebruiken twee methodes voor het ontvangen en verzenden van informatie van en naar de server: HTTP requests bij de login procedure, en een websocket voor al het andere verkeer.

Voordat we dieper ingaan op het ontwerp van de module Communicatie, lijkt het ons een goed idee om een beeld te geven van hoe de data die de app verzend en ontvangt eruit ziet. Alle data die ontvangen en verzonden wordt heeft de volgende structuur:

- <eventName >:{'action': <action >, 'data': <data >, 'data': <data >... }

De eventName geeft aan om wat voor soort object deze data gaat (dus Message, Contact, Group etc). De action geeft aan wat er met deze data moet gebeuren (toevoegen van nieuwe data, updaten van bestaande data etc) waarna de data zelf komt. Deze data bestaat uit alle attributen van het event object die naar de server gestuurd moeten worden/ die de server naar de app stuurt. Hieronder staat een lijst met alle events, en alle acties die bij zo'n event kunnen plaatsvinden.

- Event Contact
  - Action invite
  - Action pending
  - Action status
  - Action accept
  - Action remove
  - Action update
- Event Group
  - Action create
  - Action delete
  - Action update
- Event Groupmember
  - Action add
  - Action remove
  - Action role
  - Action update
- Event Message
  - Action add
  - Action delete
  - Action update
- Event Init

  - Init is een speciaal event: het start de init procedure, die bestaat uit het ontvangen van alle Users die bij het ingelogde account horen, en daarna alle data die de app nog niet heeft van al die Users. Init heeft dus geen actie.

- Event User
  - Action delete
  - Action update
- Event Search
  - Ook Search is een speciaal geval. Dit event heeft geen actie, maar alleen een waarde waarop gezocht wordt, en als de server hem terugstuurd het resultaat van de zoekopdracht.

3.2.3.1 Datamodellen We zijn bij het ontwerpen van de datamodellen begonnen met het maken van twee interfaces: Sendable en Receivable. Deze interfaces maken de code overzichtelijker en duidelijker. Het is de bedoeling dat elk object dat we naar de server kunnen sturen de Sendable interface implementeerd, en elk object dat we van de server kunnen ontvangen de Recievable. Deze interfaces hebben beiden één te implementeren methode, respectievelijk de sendObject()-methode en de receiveObject()-methode (parameters voor deze methoden laten we voor nu achterwege). In de huidige versie van de app implementeren alle datamodellen (behalve het Session datamodel) beide interfaces.

3.2.3.1.1 Sendables/Receivables

- Contact

  Een Contact is een contactpersoon. Contactpersonen zijn in Oghma Chat niet direct gelinkt aan een account, maar aan een User: het verschil is dat een account meerdere Users kan bevatten. Zie 2.4) voor uitleg.

  Een contactpersoon heeft de volgende attributen in ons systeem:

  - UUID userId (ID van de User waar deze contactpersoon aan gelinkt is)
  - UUID namespace (de namespace waarin deze persoon contactpersoon is)
  - UUID contactId (ID van deze contactpersoon)
  - boolean invite (true als deze contactpersoon jou een invite heeft gestuurd)
  - boolean pending (true als je deze contacpersoon een invite hebt gestuurd)
  - boolean status (geeft aan of deze contactpersoon online is of niet).

- Group

  Het datamodel van een Group bevat in ons systeem de volgende attributen:

  - UUID groupId (ID van de groep)
  - UUID namespace (de namespace waarin de groep bestaat)
  - String icon (pad naar de afbeelding van deze groep)
  - String subject (onderwerp/beschrijving van de groep)
  - String title (de titel/naam van de groep)

  Zoals u kunt zien, slaan we in dit datamodel niet de groepsleden op. Dit is een slecht idee omdat, als meerdere groepsleden anderen toevoegen binnen korte tijd, de server niet snel genoeg is om alle wijzigingen door te voeren. Dit zou betekenen dat alleen de laatste wijziging dan bij alle groepsleden zou aankomen, zonder dat de andere wijzigingen doorgevoerd zijn. Groepsleden hebben een eigen datamodel, genaamd Groupmember, en ook een eigen tabel in de database. Deze extra tabel geeft ons bovendien ook de mogelijkheid om efficiënt op te zoeken bij welke groep een gebruiker hoort (en of een gebruiker überhaupt bij een groep hoort).

- Groupmember

  Een groepslid heeft de volgende attributen:

  - UUID groupId (ID van de groep waar deze persoon lid van is)
  - UUID userId (ID van de pesoon zelf)
  - UUID namespace (de namespace waarin deze persoon groepslid is van een groep)
  - boolean admin (geeft aan of deze persoon de administrator is van de groep)

- Message

  Een Message is een tekstbericht. Messages kunnen verstuurd worden tussen twee Users, of naar een groep. Berichten mogen voor Oghma Chat maximaal 350 karakters lang zijn, en hebben de volgende attributen:

  - UUID id (ID van dit bericht)
  - UUID namespace (de namespace waar dit bericht toe behoord)
  - UUID from (ID van de afzender van het bericht)
  - UUID userId (ID van de ontvanger van het bericht)
  - UUID groupId (ID van de groep waar het bericht naar verzonden is)
  - String time (timestamp, geeft aan wanneer het bericht verzonden is)
  - String text (de textuele inhoud van het bericht)

Doordat een bericht óf naar een andere gebruiker óf naar een groep gestuurd kan worden, is één van de twee attributen userId en groupId altijd null.

- Session

  Dit datamodel is nogal anders de andere, omdat Session (in tegenstelling tot wat eerder gezegd is) niet de interfaces Sendable en Receivable implementeerd. Dit datamodel zorgt ervoor dat de User die op dat moment actief is, dus ingelogd is in de app, beschikbaar is voor alle klassen van onze app. Dit gebeurt door het gebruik van Shared Preferences, data die opgeslagen wordt en vervolgens in de gehele app beschikbaar is. We hebben voor Shared Preferences gekozen omdat het toegankelijke gegevens zijn die al bij het opstarten van de app gebruikt kunnen worden (in tegenstelling tot bijvoorbeeld het aanmaken van een file of SQL tabel). In ons geval slaan we de UserID en namespace van de actieve User op. Het datamodel Session heeft methodes om deze data op te slaan, en om die uit te lezen. Naast het opslaan van een actieve User, kan dit datamodel ook aangeven of de init procedure (de procedure van het ophalen van alle belangrijke data na het inloggen in de app) is afgerond of niet.

- User

  Een User is, zoals al eerder uitgelegd, deel van een Project ArteMisc account. Alle Users die verbonden zijn aan het account waarmee is ingelogd op de app, slaan we op in de lokale database. Een User heeft de volgende attributen:

  - UUID userId (ID van deze User)
  - UUID namespace (namespace waarin de User bestaat)
  - String username (de username die aan dit account gebonden is, niet de naam die anderen kunnen zien, maar waarmee ze je kunnen vinden)
  - String display (de displayname die aan dit account gebonden is, de naam die anderen kunnen zien)
  - String avatar (pad naar het profielplaatje van dit account)

Alle DataModels (behalve Session dus) hebben naast hun attributen nog de methoden sendObject() en receiveObject() geïmplementeerd. De sendObject() methode doet wat de naam impliceert: hij stuurt het huidige object naar de server (hoe dit gebeurt werken we verder uit in de delen 3.2.3.2) en 3.2.3.3). De methode receiveObject() zet een JSONObject om in een nieuw object van zijn klasse. Een voorbeeld; de receiveObject() methode van een Contact object wordt aangeroepen. De methode checkt eerst of alle parameters die nodig zijn voor een Contact object aanwezig zijn in het JSONObject. Als dit zo is, dan maakt de methode een nieuw Contact object aan, en zet de attributen van dat object gelijk aan de parameters van het JSONObject. Vervolgens wordt het nieuwe Contact object gereturned.

3.2.3.1.2 Interactie met de database Om data lokaal op te slaan, maken we gebruik van een SQLite database implementatie die standaard in Android ingebouwd is. Hiervoor hebben we een klasse SQLHelper, die de klasse SQLiteOpenHelper uitbreidt. Deze klasse creëert de database, met de volgende tabellen:

- TABLE_MESSAGES
- TABLE_USERS
- TABLE_CONTACTS
- TABLE_GMEMBERS
- TABLE_GROUPS
- TABLE_INIT

De laatste tabel, TABLE_INIT, wordt gebruikt om alle Users die horen bij het ingelogde account op te slaan. Deze tabel wordt alleen gebruikt bij de init procedure (vandaar de naam).

Voor het opslaan van objecten als een Contact of Message in de database hebben we hulpklassen gemaakt, die we DataSources genoemd hebben. Elk datamodel heeft een eigen DataSource, met in ieder geval de volgende methoden:

- saveNew() om een nieuw element in de database op te slaan.
- update() om een bestaand element te updaten.
- delete...() om een bstaand element te verwijderen.
- load...() om een element uit de database te laden.

Elke DataSource breidt de klasse ModelDataSource uit, die een aantal attributen en methoden heeft die elke DataSource nodig heeft (waarvan de belangrijkste: een database object, de context van de applicatie, en een methode om de database te sluiten). Naast de methoden hierboven hebben alle DataSources (behalve UserDataSource) nog andere methoden waarmee ze met de database communiceren:

- ContactDataSource
  - loadContacts() om alle contacten van een User te laden (zonder alle pending en invited contacten).
  - loadPending() om alle contacten te laden die op pending staan (door de User uitgenodigd).
  - loadInvited() om alle contacten te laden die op invited staan (die de User een uitnodiging gestuurd hebben).
- GroupDataSource

  - loadGroups() om alle groups van een bepaalde User (in een bepaalde namespace) te laden.

- GroupMemeberDataSource
  - loadGroupMembers() om alle groepsleden van een bepaalde groep te laden.
  - isGroupMember() om te checken of een bepaalde User lid is van een bepaalde groep.
  - isAdmin() om te checken of een bepaald groepslid de administrator van de groep is.

- MessageDataSource
  - loadGroupMessages() om alle berichten van een bepaalde groep de laden.
  - loadUserMessages() om alle berichten van een bepaalde User te laden.
  - loadMessages() om aangegeven berichten te laden (wordt door de DataSource zelf gebruikt, met parameters doorgegeven door andere methoden van de DataSource, om de API overzichtelijker te maken).

- SessionDataSource
  - exists() om te checken of een bepaalde User (in een bepaalde namespace) bestaat in de database.
  - latestUpdate() om te checken wanneer een bepaalde User voor het laatst geupdate is.
  - loadSessionUsers() om alle Users in de database te laden (wordt gebruikt voor de init procedure).

- UserDataSource (heeft geen andere methodes dan de hierboven als standaard aangegeven)

3.2.3.2 Data Handlers Data kan twee kanten op: van de app naar de server, en van de server naar de app. Om de data goed door het systeem te leiden, hebben we daarom twee soorten handlers geïmplementeerd: IntentHandlers (app - server) en EventHandlers (server - app). Als de app data naar de server wil sturen, wordt deze data verpakt in een Intent (vandaar de naam IntentHandler), dat vervolgens naar de EventService gestuurd wordt (het centrale punt in de app waar alle data die verzonden en ontvangen wordt naartoe gaat om op de goede manier verwerkt te worden, zie 3.2.3.3). De EventService zorgt er vervolgens voor dat de data op de goede manier wordt verzonden, oftewel dat de goede IntentHandler dat Intent verwerkt. Als data van de server naar de app gestuurd wordt, wordt die data ontvangen door een Transporter (de klasse die de connectie onderhoud met de server, meer hierover in het deel 'Transport'), die de data naar de goede EventHandler stuurt, aan de hand van het Event dat meegegeven is aan de data (vandaar de naam EventHandler). Daar aangekomen wordt de data op de correcte manier verwerkt.

3.2.3.2.1 Intent Handlers Alle IntentHandlers breiden de abstracte klasse GUIEventHandler uit. Deze klasse houdt de context van de applicatie en een Transporter bij (een object dat data naar de server kan sturen, zie paragraaf Transport). Ook heeft deze klasse de abstracte methode handle(). Alle IntentHandlers implementeren deze abstracte methode, waarin de IntentHandler de goede stappen neemt om de data in de meegegeven intent te verzenden naar de server. We hebben ervoor gekozen om alle IntentHandlers deze abstracte klasse te laten implementeren, omdat de Handlers allemaal de context van de applicatie en een transporter nodig hebben. Bovendien moeten ze allemaal een methode handle() implementeren, maar allemaal op hun eigen manier.

Oghma Chat heeft de volgende IntentHandlers:

- IntentHandlerContact
- IntentHandlerGroup
- IntentHandlerGroupmember
- IntentHandlerMessage
- IntentHandlerSearch
- IntentHandlerUser

De handle() methode van elke IntentHandler zit als volgt in elkaar: ten eerste wordt gecheckt of het meegegeven intent de goede informatie bevat om door die IntentHandler te kunnen worden behandeld. Als dit dit zo is, wordt er een nieuw object gemaakt van de data die naar de server gestuurd moet worden (dus als de IntentHandlerContact wordt aangeroepen, maakt die een nieuw Contact object van de meegegeven data). Vervolgens wordt dat object voorbereid voor het verzenden; naast de attributen die besproken zijn in de paragraaf DataModellen, heeft elk datamodel een attribuut action. Dit attribuut geeft aan wat de server moet doen met het object dat binnenkomt. Nadat deze actie op de goede waarde gezet is (die ook uit de intent wordt gehaald), wordt het object daadwerkelijk verzonden.

Dit geldt voor alle IntentHandlers, behalve voor IntentHandlerSearch. Deze heeft geen model nodig, maar hoeft alleen een search value aan een JSONObject mee te geven. De rest van het proces is hetzelfde als bij de andere IntentHandlers.

3.2.3.2.2 Event Handlers De EventHandlers behandelen alle data die de app ontvangt van de server. We hebben de volgende EventHandlers:

- EvtHandlerContact
- EvtHandlerGroup
- EvtHandlerGroupmember
- EvtHandlerMessage

- EvtHandlerSearch
- EvtHandlerUser

Al deze EventHandlers breiden de abstracte klasse EvtHandler uit. Deze klasse houdt, net als de GUIEventHandler, de context van applicatie bij, maar in plaats van een Transporter object houdt elke EvtHandler een Service object bij. EvtHandler implementeerd weer de klasse TransportEventHandler, die slechts één te implementeren methode heeft: handle(). Deze methode accepteerd een JSONObject, en zorgt ervoor dat de data in dat JSONObject op de goede manier wordt verwerkt binnen de app. Dat gaat als volgt: ten eerste wordt het type event (Contact, Message etc) uit de json gelezen, en wordt de json naar de goede EventHandler gestuurd. Daarna wordt geprobeerd om uit het ontvangen JSONObect de actie te halen, dus wat er met de data gedaan moet worden. Deze acties verschillen per soort event (dus Contact/Group/Groupmember/Message/User) zoals eerder al uitgelegd is. Nadat de actie bekend is, slaat de EvtHandler de data op de goede manier op in de lokale database. Hierna stuurt de EvtHandler een bericht naar de Service (zie 3.2.3.3) dat er nieuwe data beschikbaar is. De Service zorgt er vervolgens voor dat deze notificatie in het goede deel van de GUI terechtkomt, zodat de nieuwe data op de goede plek geladen wordt.

Ook bij de EvtHandlers werkt Search net wat anders. Search heeft geen actie en data, maar bevat de data waarop gezocht is en het resultaat. Dit resultaat wordt niet opgeslagen in de database, maar gaat direct naar de GUI.

3.2.3.3 Transport In deze paragraaf leggen we concreet uit hoe we in de Oghma Chat app data naar de server sturen en van de server ontvangen, en hoe we notificaties sturen naar de GUI om aan te geven dat er nieuwe data beschikbaar is. We kunnen dit het beste uitleggen aan de hand van de twee klassen die hier voor verantwoordelijk zijn:

- EventService

  We bespreken hier alleen het deel van de EventService dat berichten van en naar de server in goede banen leidt. De communicatie tussen EventService en GUI is namelijk al uitgelegd in de module 'GUI'.

- Transporter

  De implementatie van deze interface vormt de connectie tussen de app en de server. We hebben de app zo opgezet dat het gemakkelijk is om van implementatie van de Transporter te switchen. Momenteel maken we gebruik van een WebSocket, maar dat is dus gemakkelijk aan te passen mocht dat ooit nodig zijn.

3.2.3.3.1 EventService De EventService is het hart van alle communicatie van onze app. Alle berichten van en naar de server en van en naar de GUI komen op een bepaald moment door de EventService. Vanwege dit ontwerp (één centraal punt voor alle communicatie), ontstond het probleem dat de

applicatie zou kunnen blijven hangen bij het verzenden of ontvangen van data als dat in dezelfde Thread als de GUI zou gebeuren; een centraal punt betekend in dit geval dus een bottleneck. Om dit probleem te voorkomen, hebben we ervoor gekozen om gebruik te maken van een Service. Dit is een component van Android dat tijdrovende acties in de achtergrond van de app kan uitvoeren, op een eigen Thread (dus niet de UI thread).

Er zijn twee soorten Services: Started en Bound Services. Started Services zijn gemaakt om één zware berekening of actie uit te voeren. Het uitvoeren van deze berekening of actie gaat net zolang door tot hij klaar is, zelfs als de klasse die de actie opgestart heeft al lang vernietigd is. Zodra de actie of berekening klaar is, vernietigd de Service zichzelf. Omdat wij een connectie willen openhouden is dit voor ons niet de goede oplossing. Het alternatief is een Bound-Service.

Een Bound-Service stelt klassen in staat om te 'binden' met de Service. Dit betekend dat de Service actief blijft zolang er maar een klasse aan gebonden is. Bovendien stelt dit de Service in staat om status berichten naar de klassen te sturen die aan hem gebind hebben. Omdat een Bound-Service net zo lang open blijft totdat hij expliciet wordt afgesloten, hebben we voor ons ontwerp hiervoor gekozen.

Zodra de EventService opstart, maakt deze een TransportController en een GUIEventController aan. Beiden zijn interfaces, die geïmplementeerd worden door respectievelijk de WebSocketController en de IntentController. Een TransportController zorgt ervoor dat data die van de server komt naar de goede EventHandler wordt gestuurd. Om dit te kunnen doen, geeft de EventService bij het aanmaken van de TransportController alle EventHandlers aan de addEventHandler()-methode. Alle EventHandlers worden in de TransportController opgeslagen in een lijst. Zodra de Transporter (momenteel de WebSocket dus) data krijgt van de server, roept deze de methode handleTransporterEvent() van de TransportController aan, die vervolgens in de lijst van Handlers kijkt of er voor het event dat in de ontvangen data staat een handler aanwezig is. Als dit zo is, dan wordt de handle()-methode van deze EventHandler aangeroepen.

Onze huidige implementatie van de TransportController is de klasse WebSocketController. We hebben deze zo genoemd omdat hij door de WebSocket gebruikt wordt. De huidige implementatie van de GUIEventController is de IntentController: hij verwerkt Intents, en stuurt ze naar de goede IntentHandler.

Bij het aanmaken van de GUIEventController geeft de EventService via de methode addActionHandler() alle IntentHandlers, die, net zoals bij de TransportController, opgeslagen worden in een lijst. En net zoals bij de TransportController wordt, zodra de EventService een Intent krijgt van de GUI, de methode handleAction() aangeroepen, die opzoekt of de GUIEventController een IntentHandler heeft voor de doorgestuurde intent en vervolgens de handle()-methode van die IntentHandler uitvoert.

Een klein overzicht: de EventService is het hart van alle communicatie tussen de app en de server. Zodra de EventService wordt opgestart, maakt hij een GUIEventController (verantwoordelijk voor het verwerken van data van de app naar de server) en een TransportController (verantwoordelijk voor het verwerken van data van de server naar de app). De huidige implementatie van een

GUIEventController is de IntentController, en die van de TransportController is de WebSocketController.

3.2.3.3.2 Transporter De Transporter is de daadwerkelijke connectie tussen de app en de server. Hij wordt opgestart zodra de EventService wordt aangemaakt. De klasse Transporter is een interface met de volgende methoden:

- send(String data) om een String van data naar de server te sturen.
- send(TransporterEvent evt) om een object van de klasse TransporterEvent naar de server te sturen.
- connect() om de connectie met de server op te zetten.
- close() om de connectie met de server te sluiten.
- checkStatus() om de status van de connectie te checken (of er een verbinding is of niet).

TransporterEvent is een klasse waarmee we te verzenden data gemakkelijk in de goede vorm kunnen krijgen om het te verzenden. Een TransportEvent object heeft namelijk de attributen 'event' en 'json'. Event geeft aan wat voor event dit object is (Message, Contact etc), en json bevat de data die verzonden moet worden. Het gebruik van deze klasse houdt de code overzichtelijk en duidelijk in onze ogen.

Als de methode send(TransporterEvent evt) van de Transporter wordt aangeroepen, zet deze methode het TransporterEvent om in een String, en wordt de methode send(String data) aangeroepen met die string als parameter. De methode close() doet precies hij op het eerste gezicht lijkt te doen: hij sluit de connectie tussen app en server. De methode connect() is wat complexer. Hierin wordt de connectie tussen app en server tot stand gebracht.

De huidige implementatie van de interface Transporter is WebsocketTransport. Deze klasse zet een WebSocket op tussen de server en de app, met behulp van een externe library die we gevonden hebben; Android Async, te vinden op github via deze link (https://github.com/koush/AndroidAsync). Om de WebSocket open te houden, sturen we elke 20 seconden een heartbeat bericht naar de server, om te kijken of de verbinding nog in stand is. Momenteel hebben we nog niet echt een goede implementatie voor zwakke connecties,dat wil zeggen: connecties die vaak kort wegvallen. Daar hopen we binnenkort wat aan te doen.

Het eerste dat gebeurt zodra de WebSocket is opgezet, is het starten van de Init procedure. Dit houdt in dat de app een bericht naar de server stuurt me de boodschap dat hij een init procedure wil starten. De server stuurdt dan een lijst terug met accounts die gelinked zijn aan het account van de ingelogde gebruiker (bij het opstarten van de WebSocket worden cookies meegegeven die ontvangen zijn na het inloggen in de app, waardoor de server weet wie er is ingelogd). Vervolgens antwoordt de app met een bericht waarin wordt aangegeven van welke Users er data geladen moet worden. Daarna stuurt de server al die data, en als hij klaar is een bericht met de boodschap dat de init procedure klaar is.

Momenteel wordt de communicatie tussen server en app NIET versleuteld. We hopen dit z.s.m. te implementeren, maar om dat goed te doen moeten we een certificaat krijgen waarmee we onze publieke sleutel kunnen vastleggen. Vanwege de korte tijdsspanne van dit project hebben we dat nog niet voor elkaar gekregen, maar het is zeker een belangrijk punt voor ons, aangezien we ons focussen op privacy. Todat we de informatie kunnen versleutelen, raden we dan ook sterk af om deze app de gebruiken voor gesprekken met gevoelige inhoud.

3.3 Ontwerpverantwoording

De belangrijkste reden dat we voor deze gui hebben gekozen is omdat deze het meest overeenkomt met de gui die we voor de web-app hebben gekozen. Het is een schoon en simplistisch ontwerp, en we denken dat de app hier in zijn geheel duidelijker van wordt. Er zijn echter toch een paar puntjes anders dan in de web versie.

3.3.1 Drawers

We hebben gekozen voor twee drawers, aan beide kanten één, terwijl deze in de web-versie nergens te vinden zijn. Aan de rechterkant is echter wel een side-bar te vinden waarin dezelfde informatie staat als in de rechter drawer. Omdat een telefoon vaak een klein scherm heeft, vonden we de drawer een goede oplossing om deze side-bar toch in de app te houden. Een bijkomende factor is dat dit al een gebruikte oplossing is in andere apps, en dat werkt goed.

In de web-versie zijn dingen als accountbeheer en namespacebeheer te vinden in onze eigen versie van de action bar. Hier is op het kleine telefoonscherm helaas geen ruimte voor. Het zou een soort drop-down menu moeten worden om dit goed kwijt te kunnen. Hiervoor zijn drie basis-opties:

- Een klein drop-down menu met een knop om hem te openen, meestal geplaatst aan de rechter kant van de action bar, die ook luistert naar de hardware-matige knop opties.

  In ons ontwerp zat aan de rechter kant van de action bar echter al een knop om de rechter drawer te openen. Deze knop op een andere plek plaatsen breekt met de android-conventies. Het zou een optie kunnen zijn, maar hij heeft geen voorkeur.

- Een tabhost-achtig menu net onder de action bar.

  Ten eerste zou door het plaatsen van deze knop nog meer ruimte verloren gaan aan de action bar. Dit zou vooral op kleine telefoons een probleem worden. Daarnaast wordt dit, vooral als er later wat knoppen bij zouden komen (wat wel ons plan is) zeer irritant om te gebruiken. Deze optie valt dus af.

- Een navigation drawer aan de linker zijde van het scherm.

Dit is tot nu toe de beste optie. Ten eerste zit de knop om de opties te openen nu op een relatief logische plek (de linker bovenhoek) en ten tweede staan alle opties zo direct onder elkaar. Daarnaast is het met dit ontwerp mogelijk om met hooguit drie taps (en misschien een beetje scrollen) naar elke mogelijke chat-room te komen en in twee taps op elke andere locatie in de app.

Wij denken hierom dat we er goed aan hebben gedaan om de opties in de linker drawer te plaatsen, vooral omdat dit groei van de app toestaat. Groei die we zeker van plan zijn waar te maken. De reden dat we hiervoor gekozen hebben is dus voor een groot deel theoretisch, maar ook gebaseerd op andere apps en een beetje experimenteren.

3.3.2 Fragments

In ons ontwerp hebben we ervoor gekozen om fragments te gebruiken om onze gui in elkaar te zetten. Dit heeft een aantal redenen gehad. Ten eerste is het veel gemakkelijker om fragments te hergebruiken. Bijvoorbeeld: op een tablet kunnen we de drawer fragments er gewoon naast zetten in plaats van ze als drawer te laten functioneren. Ten tweede wordt de app hier veel overzichtelijker van. Bij het gebruik van activities is het veel lastiger de componenten over verschillende klassen te verspreiden en is de kans op code-duplicaten veel groter. De reden dat we hiervoor gekozen hebben is dus puur theoretisch, maar wel goed onderbouwd.

3.3.3 Actionbar

We hebben ervoor gekozen om als actionbar een custom fragment te gebruiken in plaats van de native actionbar. Dit is omdat we de native actionbar veel te autoritair vonden. Met deze nieuwe actionbar werd het uiteindelijk ook veel gemakkelijker om twee verschillende knoppen voor de drawers erin te zetten. In de native actionbar was dit veel lastiger (dat wil zeggen: het is ons niet gelukt. Android ondersteund blijkbaar geen twee drawers tegelijkertijd in samenwerking met de native actionbar.). Deze keuze is dus voor het grootste gedeelte gebaseert op experimenteren.

4 Reflectie

Over het algemeen zijn we het erover eens dat dit project goed is gegaan. Waar we het ook over eens zijn, is dat we voor de acht weken die we voor dit project hadden, misschien wel een erg groot project gekozen hebben. Dat bleek vooral bij het inleveren van het prototype: we hadden al enorm veel code geschreven, maar de verschillende modules van de app (GUI, Communicatie en Server) waren eigenlijk nog op geen enkele wijze echt aan elkaar gelinked. Dat we nog veel werk te doen hadden bleek ook toen we groep PRLJ onze app gaven voor het Usability-onderzoek. PRLJ kon vrij weinig van de chat functionaliteit testen,

en kon eigenlijk alleen maar beoordelen hoe de app eruit zag en hoe het werken met de app aanvoelde. Desalniettemin denken we dat we, aan het einde van deze acht weken, toch veel bereikt hebben (wat ook wel mag met bijna 7000 regels gestructureerde en werkende Java-code).

Aan het begin van dit project kwamen we erachter dat Eclipse en Git niet goed samenwerken; het kostte ons veel tijd om het project, dat door het pushen en pullen met Git steeds kapot ging, te herstellen. Dit heeft ons uiteindelijk doen besluiten om over te stappen op Android Studio. Android Studio is een IDE die gemaakt is om Android op te programmeren (in tegenstelling tot Eclipse, war je plugin nodig hebt om in Android te programmeren). Dit zorgde ervoor dat niet meer tijd kwijt raakten aan het herstellen van projecten en computers, en gewoon konden werken aan de app.

Doordat we de app opgesplitst hebben in drie delen die los van elkaar te programmeren zijn, was communicatie van het grootste belang; we moesten elkaar natuurlijk op de hoogte houden van elkaars voortgang, en bovenal moesten we elkaars code ook kunnen begrijpen (wat vooral belangrijk was bij het in elkaar schuiven van de verschillende modules). Dit hebben we redelijk goed opgelost door elke week ongeveer twee keer bij elkaar te komen om samen aan het project te werken. Daarnaast hadden we een zeer actieve facebook chat waar we elkaar vragen konden stellen en vertellen wat er allemaal gebeurt was en nog moest gebeuren. Wat ook een grote hulp was, was het gebruik van Git en Bitbucket. Git is een version control programma, en Bitbucket is een website waar je m.b.v. Git code kunt oploaden, managen en downloaden. We hebben hier intensief gebruik van gemaakt, waardoor iedereen altijd toegang had tot de nieuwste code.

We hebben erg veel geleerd van het werken aan dit project. Technische dingen als het werken met SQLite database implementaties, WebSockets, het ontwerpen van een systeem, programmeren in Android, het gebruik vn XML etc., maar ook op organisatorisch vlak hebben we een hoop geleerd, zoals het managen van code, en het stellen van haalbare deadlines (wat bij ons misschien wat mislukt is). Daarnaast was het een leerzame ervaring om een langere periode in teamverband te werken aan een wat grotere opdracht. Voorheen hebben we vooral kleinere programmeeropdrachten gehad, die je alleen of met zijn tweeën moest maken.

In het begin is het lastig om aan een groot project te beginnen, omdat je eigenlijk niet weet waar je aan moet beginnen; er moet immers zo ontzettend veel gebeuren. Hier hadden vooral wij, Thijs en Wouter, wat moeite mee. Gelukkig konden Jan en Niek ons op weg helpen: zij hadden al wat meer ervaring met het opstarten van grote projecten, en ze konden ons uitdagen om eens wat nieuws te doen. En na een tijdje merk je dan dat het werken aan het project steeds makkelijker gaat, en dat je toch enorm veel geleerd hebt.