

Konzeption und Implementierung einer nativen iOS-App zum Nachrichtenaustausch mit innovativen Funktionalitäten

**Bachelor-Thesis
zur Erlangung des akademischen Grades B.Sc.**

**Vincent Friedrich
2326998**



Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüfer: Prof. Dr. Andreas Plaß
Zweitprüfer: Prof. Dr. Torsten Edeler

Hamburg, 26. 08. 2019

Abstract

The basic concept behind digital conversation has remained unchanged for some time now. Messages are often sent while the other person does not require active presence. This can be useful, but it creates an impersonal kind of communication. This work will deal with the development and implementation of an innovative concept for messaging on the mobile platform iOS that focuses on the opposite side of the conversation and draws the user's attention to the conversation.

Zusammenfassung

Das grundlegende Konzept hinter der digitalen Konversation bleibt seit einiger Zeit unverändert. Nachrichten werden oft abgeschickt während das Gegenüber keine aktive Präsenz erfordert. Dies kann sich als nützlich erweisen, doch es entsteht eine unpersonliche Art von Kommunikation. Diese Arbeit wird sich mit der Anfertigung und Umsetzung eines neuartigen Konzeptes zum Nachrichtenaustausch auf der mobilen Plattform iOS auseinandersetzen, welches die gegenüberliegende Seite der Konversation in den Mittelpunkt stellt und die Aufmerksamkeit des Nutzers auf die Konversation lenkt.

Inhaltsverzeichnis

1 Einleitung	7
1.1 Motivation	7
1.2 Idee	7
1.3 Zielsetzung	8
1.4 Aufbau der Arbeit	8
2 Grundlagen	10
2.1 Digitale Kommunikation	10
2.2 Instant Messaging	10
2.3 Digitale Barrierefreiheit	11
3 Anforderungsanalyse	12
3.1 Zielgruppe	12
3.2 Plattform	13
3.3 Markt und Wettbewerb	13
3.4 Funktionen	17
3.4.1 An- und Abmeldung	17
3.4.2 Freundschaftsanfragen	20
3.4.3 Nachrichtenanfragen	24
3.4.4 Nachrichtenaustausch	26
4 Design	30
4.1 Richtlinien	30
4.2 Benutzeroberfläche	31
4.3 Name	34
4.4 Logo	34
5 Softwarearchitektur	35
5.1 Entwicklungsstrategie	35
5.2 Entwurfsmuster	35
5.2.1 MVC	35
5.2.2 Observer Pattern	37
5.2.3 ViewController	40
5.2.4 MassiveViewController	41
5.2.5 MVVM	41

Inhaltsverzeichnis

5.3	Architektur der App	43
5.3.1	Entwurfsmuster	43
5.3.2	Models	44
5.3.3	ViewController	45
5.3.4	ViewModels	45
5.3.5	ListViewModels und CellViewModels	46
5.3.6	Worker	47
5.3.7	Provider	48
6	Entwicklung	49
6.1	Entwicklungsumgebung	49
6.2	Swift	49
6.3	Bibliotheken	50
6.3.1	Apple	51
6.3.2	CocoaPods	51
6.3.3	Alamofire	52
6.3.4	Firebase	53
6.3.5	Weitere Bibliotheken	53
6.4	Backend	55
6.4.1	HTTP	55
6.4.2	Stream	57
6.4.3	JSON	57
6.4.4	SQL	58
6.5	Zusammenfassung	59
7	Implementierung	60
7.1	Kommunikation zwischen App und Backend	60
7.2	Hierarchie der ViewController	61
7.3	Login	63
7.4	Darstellung von Listenansichten	64
7.4.1	Überblick	64
7.4.2	Animierte Aktualisierung des Listeninhaltes	65
7.5	HTTP-Requests	66
7.6	Stream-Verbindung	68
7.6.1	Übersicht	68
7.6.2	Senden von Daten	69
7.6.3	Empfangen von Daten	69
7.7	Freundschaftsanfragen	70
7.7.1	Versenden einer Freundschaftsanfrage	70
7.7.2	Empfangen einer Freundschaftsanfrage	71
7.7.3	Beantworten einer Freundschaftsanfrage	72
7.8	Nachrichtenanfragen	72

Inhaltsverzeichnis

7.9	Nachrichtenaustausch	74
7.9.1	Textnachrichten in Echtzeit	78
7.9.2	Sprachnachrichten mit Transkription	80
7.9.3	Soundeffekte	81
7.9.4	Datenpersistenz	81
7.10	Technische Schwierigkeiten	82
7.10.1	Hoher Datenverkehr	82
7.10.2	Spracherkennung	83
8	Fazit	84
8.1	Auswertung	84
8.2	Ausblick	84
8.2.1	Weiterentwicklung des bestehenden Konzeptes	84
8.2.2	Technische Weiterentwicklung	86
8.2.3	RTT	86
A	Material	87
A.1	CD	87
A.2	Informationen zum Code	87
A.3	Abbildungen	88
A.4	Tabellen	91
Abbildungsverzeichnis		96
Tabellenverzeichnis		99
Quellcodeverzeichnis		100
Literaturverzeichnis		101

Inhaltsverzeichnis

Disclaimer

Die in dieser Arbeit gewählte männliche Form bezieht sich stets zugleich auf weibliche, männliche und anderweitige Geschlechteridentitäten. Auf eine Mehrfachbezeichnung wurde zugunsten einer besseren Lesbarkeit verzichtet.

1 Einleitung

1.1 Motivation

Die Art und Weise, wie wir heutzutage Nachrichten über unsere Mobiltelefone austauschen, hat sich im Laufe der letzten Jahre nicht wirklich verändert. Große Marktführer wie WhatsApp oder Facebook Messenger aktualisieren ihre Applikationen laufend. Es werden Funktionen wie der Austausch von GIF-Animationen¹, Sticker² oder sich wieder zerstörende Nachrichten ergänzt. Der Erfolg von Emojis³ zeigt, wie wichtig eine persönliche Note beim digitalen Nachrichtenaustausch sein kann. Das grundlegende Konzept hinter der digitalen Konversation bleibt allerdings immer gleich: Nachrichten werden getippt und abgeschickt. Der Nutzer empfängt sie - unabhängig von seinem aktuellen Anwesenheitsstatus. Es ist im Grunde weniger wie eine reale Konversation und mehr wie ein beschleunigter, digitaler Postverkehr in Echtzeit. Überlegt man sich die Anforderungen an eine echte Konversation und versucht diese in eine digitale Form umzuwandeln, wird man allerdings schnell feststellen, dass es ein angestrebtes Ziel sein sollte, nicht nur die ausgetauschten Wörter per se, sondern auch den Zeitpunkt zu dem sie ausgedrückt werden wollten so realitätsnah wie möglich darzustellen.

1.2 Idee

Idee dieser Arbeit ist die Konzeption und Implementierung einer nativen iOS-App mit dem Namen »M2«⁴. M2 ist eine neuartige Form von Messenger, soll dabei aber nicht den alltäglichen Messenger ersetzen. Die Art und Weise des Kommunikationsablaufs in M2 lässt gezielt keine mal eben schnell abgesendeten Nachrichten zu und erfordert beidseitige Aufmerksamkeit. Es soll sich von beiden Teilnehmern Zeit für die Konversation genommen werden. Für eine höhere Privatsphäre werden Freundschaften per Freundschaftsanfragen geschlossen, bevor eine Anfrage zur Konversation gesendet werden kann. Nach Beginn der Konversation ähnelt die Erfahrung einem Telefonat in Textform. Um diese Idee zu unterstützen, werden Nachrichten durch buchstabenweise Darstellung dem gegenüberliegenden Nutzer angezeigt während sie eingetippt werden. Die Nutzer können sich gegenseitig unterbrechen, ein Nutzer kann schon eine Antwort formulieren, während der andere Nutzer noch tippt, usw. Es entsteht eine

¹Bewegte, sich oft wiederholende Bilder in Sekundenabständen

²Illustrationen von Charakteren oder Emotionen

³Piktogramme von Charakteren oder Emotionen

⁴Der Name „M2“ steht für „Messenger 2.0“. Erläuterung in Abschnitt 4.3

neue Dynamik in der Konversation. Ebenfalls werden die eigenen Nachrichten gezielt nicht angezeigt. Nach Beendigung der Konversation wird der Nachrichtenverlauf verworfen, um die Erfahrung an die eines Telefonates anzugeleichen. Sprachnachrichten können außerdem als Text angezeigt werden. Eine genauere Beschreibung des Ablaufs zum Nachrichtenaustausch findet sich in Abschnitt 3.4.4.

1.3 Zielsetzung

Die Konzeption und Implementierung der aus dieser Arbeit resultierenden iOS-App M2 konzentriert sich auf eine gezielte Verschiebung der Aufmerksamkeit des Nutzers von den üblicherweise umgebenden Funktionalitäten ganz auf die Konversation selbst und insbesondere den gegenüberliegenden Nutzer. Es soll unter Aufgreifen bestehender Konzepte, und einer Weiterentwicklung dieser, eine neuartige Form von Nachrichtenaustausch entwickelt werden. Die neu überdachten Abläufe für die Nutzung der App, Konversationsstart und Nachrichtenaustausch verschmelzen moderne Mechanismen wie Freundschaftsanfragen, die bereits in sozialen Netzwerken zum Einsatz kommen, mit altbekannten, wie Nachrichtenaustausch aus dem Instant Messaging⁵ und innovieren den üblichen Konversationsverlauf mit Chatanfragen und buchstabenweiser Übertragung der Nachrichten, während sie eingetippt werden. Ziel ist es, eine optisch ansprechende App zu schaffen, die in ihren Kernfunktionalitäten stabil läuft und eine gute Basis zur Veranschaulichung der beschriebenen Mechanismen bietet. Den vollen Funktionsumfang vergleichbarer Messenger wie WhatsApp wird sie jedoch aufgrund des begrenzten Entwicklungszeitraums nicht bieten. Der gezielte Anwendungsfall der resultierenden App soll ebenfalls nicht den eines alltäglichen Messengers wie WhatsApp abdecken, sondern vielmehr für bestimmte Einzelfälle genutzt werden. Auf den vollen Funktionsumfang wird in 3.4 tiefer eingegangen.

1.4 Aufbau der Arbeit

Im Anschluss an dieses Einleitungskapitel folgt ein Kapitel für die Erläuterung der **Grundlagen**. Hier werden zum besseren Verständnis der nachfolgenden Kapitel grundlegende Begriffe wie *Digitale Kommunikation*, *Instant Messaging* und *Digitale Barrierefreiheit* erklärt.

Im Anschluss folgt das Kapitel **Anforderungsanalyse**, welches sowohl nicht-technische Anforderungen wie die *Zielgruppe* als auch technische Anforderungen der iOS-App wie die Wahl der *Plattform* definiert. In diesem Kapitel wird außerdem jede einzelne Funktionalität der iOS-App analysiert und konzipiert.

Nachdem die Anforderungen an die iOS-App festgelegt wurden, wird im Kapitel **Design** auf die designtechnische Umsetzung der Benutzeroberfläche in Bezug auf die konzipierten Funktionalitäten eingegangen.

⁵Begriffserklärung in Abschnitt 2.2

1 Einleitung

Im folgenden Kapitel **Softwarearchitektur** wird erläutert, welche Softwarearchitektur geplant wird, um die konzipierten Funktionalitäten umzusetzen. In einem Teil werden grundlegende Begriffe wie *MVC* und *MVVM* erklärt. In einem weiteren Teil wird dann erläutert, wie genau die *Architektur der App* geplant ist.

Im Kapitel **Entwicklung** wird erläutert, welche Programmiersprachen, Entwickler-tools und Bibliotheken für die Umsetzung der iOS-App verwendet werden. Hier wird außerdem auch auf grundlegende Merkmale des Backends eingegangen, welches für die Gesamtumsetzung des Projektes erforderlich ist.

Als Hauptteil dieser Arbeit folgt das Kapitel **Implementierung**, welches auf die programmatische Umsetzung aller geplanten Funktionalitäten der iOS-App in Zusammenarbeit mit dem Backend eingeht. Hier wird im ersten Schritt erklärt, wie die *Kommunikation zwischen App und Backend* funktioniert. Danach werden grundlegende Merkmale der iOS-Entwicklung wie die *Hierarchie der ViewController* und die *Darstellung von Listenansichten* betrachtet. Da alle weiteren Abschnitte des Kapitels sich immer wieder auf die Nutzung von Funktionalitäten der Netzwerkkommunikation beziehen werden, folgen die Abschnitte *HTTP-Requests* und *Stream-Verbindung*. Darauf bauen dann mehrere Abschnitte auf, in denen auf die Implementierung der Funktionalitäten der iOS-App eingegangen wird. Im letzten Abschnitt des Kapitels werden *Technische Schwierigkeiten*, die bei der Implementierung aufgetreten sind, beleuchtet.

Abschließend folgt ein **Fazit**, welches das Gesamtergebnis dieser Arbeit auswertet und einen Ausblick auf eine mögliche Weiterentwicklung der entstandenen iOS-App aufzeigt.

2 Grundlagen

2.1 Digitale Kommunikation

„Digitale Kommunikation bedeutet zunächst Kommunikation mithilfe digitaler Medien. Unter den digitalen Medien steht das Internet mit seinem vielfältigen Angebot an Publikation und Wechselrede an erster Stelle. Dass die Kommunikation selbst digital genannt wird, ist aber keine sprachliche Nachlässigkeit, sondern bringt zum Ausdruck, dass die Kommunikation über digitale Medien eine andere wird.“ ([Grimm & Delfmann 2017: 1](#)).

Digitale Kommunikation lässt sich im Modell „*Sender-Empfänger-Botschaft-Medium*“ betrachten ([Grimm & Delfmann 2017: 3](#)).

Sender und Empfänger sind hierbei die zwei Nutzer der Konversation.

Die Botschaft ist eine Nachricht, die vom Sender zum Empfänger transportiert wird. Das Medium ist der Kanal des Transportes und der Darstellung dieser Nachricht.

Digitale Kommunikation besitzt somit die Fähigkeit, aufgrund ihrer Aufbereitung durch das Medium eine Konversation beim Empfänger anders darstellen zu können, als beim Absender. Auf Ebene dieser Arbeit steht der Transport der Botschaft und dessen Darstellung durch die App M2 als Medium im Mittelpunkt, da der Transport durch das Medium nicht nur den Wortlaut, sondern auch den Sinn der Botschaft beeinflusst ([Grimm & Delfmann 2017: 3](#)). Durch die buchstabenweise Darstellung der Botschaft in M2 entsteht eine neue Möglichkeit für den Empfänger, die Nachricht in ihrer Entstehung zu verfolgen und diese interpretieren zu können.

2.2 Instant Messaging

Instant Messaging¹ bezeichnet im Bereich der digitalen Medien Software zum sofortigen Nachrichtenaustausch. Kurze Textabschnitte werden über ein Netzwerk ausgetauscht und ermöglichen direktes Antworten innerhalb einer digitalen Konversation. Software dieser Art wird als „Messenger“ bezeichnet. Bekannte Beispiele für Messenger in heutiger Zeit sind Skype oder WhatsApp². Die Entwicklung des ersten Messengers „ICQ“ vom israelischen Unternehmen Mirabilis in 1996 ([Kessler 2008](#)) zielte vorerst ausschließlich auf den Desktopbereich ab. Das erste iPhone wurde von

¹abgekürzt IM

²Marktanalyse in Abschnitt 3.3

der Firma Apple in 2007 ([Apple 2007](#)) veröffentlicht. Das mobile, internetfähige Telekommunikationsgerät eroberte innerhalb kürzester Zeit den Massenmarkt und stellt heutzutage zusammen mit konkurrierenden Geräten für viele einen Alltagsgegenstand dar. Es ermöglichte somit eine Nutzung von mobilen Messengern von überall aus und ersetzte bald die SMS³.

2.3 Digitale Barrierefreiheit

Barrierefreiheit in der digitalen Welt spielt eine wichtige Rolle. Es geht um die Gleichstellung aller Menschen. Egal ob eine Behinderung vorliegt oder nicht sollte jeder die gleiche Möglichkeit bekommen, von digitalen Systemen profitieren zu können. Oft erhält dieser Aspekt bei der Entwicklung von Software aus Gründen des Aufwandes leider wenig Aufmerksamkeit. Für gewisse Bereiche, wie digitale Produkte von Trägern öffentlicher Gewalt, ist es Pflicht, Barrierefreiheit zu gewährleisten (§ 12a Abs. 1 BGG.). Dies gilt bspw. für Museen, Behörden oder Ämter mit Webauftritt und/oder App. [Apple \(2019\)](#) empfiehlt die Realisierung von Barrierefreiheit in Apps durch Einstellungsmöglichkeiten wie Farbanpassungen im Display, Einstellungsmöglichkeit der Schriftgröße, große Elemente in der Benutzeroberfläche, verschiedene Spracheinstellungen oder Sprachführung für Menü und Funktionalitäten. Die Idee hinter M2 bietet einen Ansatz, der auch für Menschen mit einer starken Beeinträchtigung im Hörvermögen interessant sein kann, da die Konversation durch das buchstabenweise Übertragen der Nachrichten mehr einer nicht-digitalen Konversation ähnelt und sich Sprachnachrichten in Text konvertieren lassen. Übrige Funktionalitäten, die für offizielle Barrierefreiheit gewährleistet sein müssen, sind aktuell aus Zeitgründen und Gründen von Schwerpunktsetzung nicht in M2 implementiert. Aber in einer Weiterentwicklung ist darüber durchaus nachzudenken (siehe Abschnitt 8.2.1).

³Short Message Service: Beim Telekommunikationsanbieter in Anspruch zu nehmender Nachrichtendienst ohne Nutzung eines mobilen Datentarifs

3 Anforderungsanalyse

3.1 Zielgruppe

Bei Untersuchung einer Statistik von Faktenkontor (Abb. 3.1), die sich auf die Nutzung von Social-Media-Angeboten verschiedener Altersgruppen bezieht, fällt auf, dass ein etablierter Messenger mit einem altbewährten Konzept in allen Altersgruppen deutlich erfolgreicher ist, als ein modernerer Ansatz wie von Snapchat¹. Da das Konzept von M2 ebenfalls einen moderneren Ansatz verfolgt, wird die angestrebte Altersgruppe bei 14-39 festgesetzt. Dies muss allerdings nicht bedeuten, dass andere Altersgruppen damit ausgeschlossen sind. Bspw. für Menschen mit Beeinträchtigungen des Hör- oder Sprechvermögens, unabhängig von der Altersgruppe, kann der Ansatz hinter M2 interessant sein. Hierauf wird im Abschnitt 3.4.4 tiefer eingegangen. Eine Geschlechterbeschränkung wird bei der Festsetzung der Zielgruppe für M2 nicht vorgenommen.

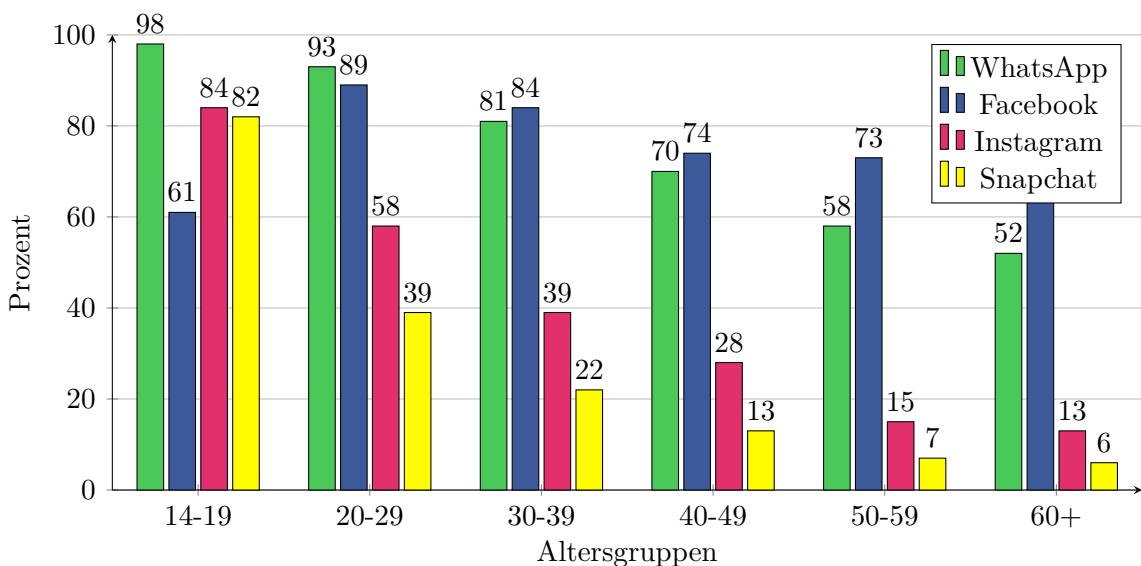


Abbildung 3.1: Nutzung von Social-Media-Angeboten (in Prozent)
Quelle: Faktenkontor (2018), modifiziert

¹Mehr Informationen zu der Funktionalität hinter WhatsApp und Snapchat in Abschnitt 3.3

3.2 Plattform

Vergleichbare Messenger wie WhatsApp und Snapchat verwenden hauptsächlich die beiden mobilen Plattformen iOS ([Apple 2019](#)) und Android ([Google 2019](#)). Über 80% der Menschen in Deutschland in der Altersgruppe zwischen 14 und 65 nutzen zumindest hin und wieder ein Smartphone ([Bitkom Research 2018](#)). Außerdem bietet es sich aufgrund der ständigen Erreichbarkeit überall an, eine mobile Plattform für M2 zu verwenden. Aufgrund von langjähriger eigener Berufserfahrung im Bereich der App-Entwicklung für iOS, fiel die Wahl bei der Zielplattform von M2 ebenfalls auf iOS.

3.3 Markt und Wettbewerb

Die Nutzung von Instant Messaging auf Smartphones hat sich innerhalb der letzten Jahre stark verbreitet. Während einige große Apps sich mittlerweile als Standard etabliert haben, versuchen trotzdem viele neue Plattformen sich ebenfalls auf dem Markt zu behaupten.

Im Folgenden wird eine Liste mit Features aufgezeigt, die die meisten Messenger auf mobilen Plattformen besitzen (inklusive der Features von M2). Ohne Zweifel gibt es noch etliche weitere Features wie beidseitiges Löschen oder Bearbeiten von Nachrichten, öffentliche Chats, Umfragen, etc. aber zugunsten der Übersicht und aufgrund von geringerem Vorkommen besagter Features zwischen den einzelnen Messengern wurden diese außen vor gelassen.

Gängige Features von Messengern auf mobilen Plattformen
Versenden und Empfangen von Textnachrichten
Versenden und Empfangen von Sprachnachrichten
Versenden und Empfangen von Bildern und Videos
Versenden und Empfangen von GIF-Animationen
Versenden und Empfangen von Stickern
Erstellen von Gruppen
Nutzername über Telefonnummer
Sprachanrufe
Videoanrufe
Stories ²
Teilen des Standortes
Freundschaftsanfragen
Chatanfragen
Textnachrichten in Echtzeit beim Eintippen
Verschlüsselter Nachrichtenaustausch möglich

Tabelle 3.1: Gängige Features von Messengern auf mobilen Plattformen

3 Anforderungsanalyse

Feature	WA	T	V	FM	Sn	K	Th	Sk	Bo	M2
Textnachrichten	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
Sprachnachrichten	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Nein	Ja
Bildern und Videos	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Nein	Nein
GIF-Animationen	Ja	Ja	Ja	Ja	Nein	Ja	Nein	Ja	Nein	Nein
Sticker	Ja	Ja	Ja	Ja	Ja	Ja	Nein	Ja	Nein	Nein
Gruppen	Ja	Ja	Ja	Ja	Nein	Ja	Ja	Ja	Nein	Nein
Anm. o. Mobilnummer ³	Nein	Ja	Nein	Ja	Nein	Ja	Nein	Ja	Nein	Ja
Sprachanrufe	Ja	Ja	Ja	Ja	Ja	Nein	Ja	Ja	Nein	Nein
Videoanrufe	Ja	Ja	Ja	Ja	Ja	Nein	Ja	Ja	Nein	Nein
Stories	Ja	Nein	Nein	Ja	Ja	Nein	Nein	Nein	Nein	Nein
Teilen Standort	Ja	Ja	Ja	Ja	Ja	Nein	Nein	Ja	Nein	Nein
Freundschaftsanfragen	Nein	Nein	Nein	Ja	Ja	Nein	Nein	Ja	Nein	Ja
Chatanfragen	Nein	Nein	Nein	Teils ⁴	Teils ⁵	Nein	Nein	Nein	Nein	Ja
Textnachrichten Echtzeit	Nein	Nein	Nein	Nein	Nein	Nein	Nein	Nein	Ja	Ja
Verschlüsselung	Ja	Ja	Ja	Ja	Nein	Ja	Ja	Ja	k.A.	Nein

Tabelle 3.2: Vergleich gängiger Features von Messengern auf mobilen Plattformen (zuletzt geprüft: 26.07.2019)

Legende

- WA: WhatsApp
- T: Telegram
- V: Viber
- FM: Facebook Messenger
- Sn: Snapchat
- K: Kik
- Th: Threema
- Sk: Skype
- Bo: Bolt

²Das Teilen von Kurzvideos und Bildern für alle vernetzten Kontakte auf der jeweiligen Plattform, oftmals innerhalb eines Verfügbarkeitszeitraums von 24 Std

³Anmeldung ohne Mobilnummer

⁴Nur für Unbekannte

⁵Benachrichtigung bei Start

3 Anforderungsanalyse

Wie in Tabelle 3.2 erkennbar, ähneln sich die meisten Messenger auf mobilen Plattformen stark. Die Features überschneiden sich zumeist und der Unterschied zwischen den Messengern ist oft nur CI⁶, Markenführung oder schneller, besser, sicherer, etc. umgesetzte Features. Der Reiz zur Nutzung besteht oft durch persönliche Vorliebe oder Markensympathie.

Alle im Folgenden genannten Apps sind in Tabelle 3.3 aufgeführt.

Einer der ersten mobilen Messenger war die 2009 veröffentlichte Anwendung WhatsApp. WhatsApp machte sich als einer der Ersten die, damals noch neue, Plattform iOS zu Nutzen. Innerhalb weniger Jahre hat es WhatsApp geschafft, sich als Standard in der digitalen Kommunikation zu etablieren und der Anwendungsbereich geht inzwischen weit über die reine Kommunikation hinaus. Im Jahr 2018 nutzten zwischen 81% und 98% der Deutschen im Alter zwischen 14 und 39 Jahren zumindest hin und wieder WhatsApp (Abb. 3.1). Trotzdem beruhte das Konzept von WhatsApp aufgrund seines Ursprungs lange Zeit auf den minimalsten Funktionalitäten. Neben Text- und Sprachnachrichten und Versenden von Bildern und Videos blieb weitere Innovation lange aus.

Der Erfolg von WhatsApp animierte Konkurrenten ebenfalls zur Entwicklung ähnlicher Apps. Im Laufe der letzten Jahre wurden diverse Messenger auf den Markt gebracht. Ein Teil der Konkurrenz konzentriert sich auf denselben Funktionsumfang, aber mit dem Anspruch, diese Features besser oder auf mehr Plattformen umzusetzen als WhatsApp. Hierzu zählen bspw. Telegram, Viber oder Kik.

Da WhatsApp viele Jahre unverschlüsselt Nachrichten verschickte, entstanden bspw. Apps wie Threema, deren Fokus ganz auf der sicheren Zustellung einer Nachricht liegt.

Andere Apps wie Snapchat versuchten sich an neuen Funktionsansätzen.

In Snapchat wird z.B. der Nachrichtenverlauf nur solange gespeichert, wie die Konversation geöffnet bleibt. Des Weiteren fokussiert sich das Konzept dort auf das Versenden von Bildern und Kurzvideos mit kurzweiligem Verfügbarkeitszeitraum. Snapchat etablierte zudem die sogenannten Stories, welche heutzutage auch in vielen anderen Messengern zu finden sind.

WhatsApp hat sich im Laufe der letzten Jahre zu einer Art Hybrid dieser Apps entwickelt. Nach Übernahme von Facebook in 2014 ([Reuters 2014](#)) kamen mit der Zeit neue Features wie Stories, GIFs, Sticker oder Verschlüsselung hinzu.

Der Messenger Bolt verfolgt ein ähnliches Konzept wie M2, laut der Beschreibung in Apples App Store werden die Nachrichten so zugestellt, wie sie eingetippt werden. Leider konnte Bolt nicht vollständig evaluiert werden, da die App seit einigen Jahren nicht mehr gepflegt wurde und beim Start abstürzt.

Beam Platform bietet hingegen ein SDK⁷ zum Chatten in Echtzeit, sowohl in Be-

⁶ „Corporate Identity“: Der designtechnische Auftritt einer Anwendung inkl. Farbpalette, Schriftart, etc.

⁷Software Development Kit: Sammlung vorgefertigter Algorithmen und Programmbibliotheken zur

3 Anforderungsanalyse

zug auf Textnachrichten als auch für Telefonie. Das Beam SDK ermöglicht es laut Website des Anbieters u.a., ein Telefonat in Echtzeit zu transkribieren und als Chat darzustellen.

Messenger wie Skype, die bereits vor WhatsApp als Desktopanwendung erfolgreich waren, haben ihren Fokus innerhalb der letzten Jahre stark auf Apps gelegt und verfügen über einen ähnlichen Funktionsumfang.

Nach Übernahme von WhatsApp durch Facebook fing auch der Facebook Messenger an, sich in seiner Funktionalität WhatsApp anzugeleichen. Wie die New York Times Anfang 2019 berichtete, plant der Geschäftsführer von Facebook Mark Zuckerberg auf lange Sicht eine Zusammenschließung u.a. der beiden Apps (Isaac 2019).

M2 macht sich ebenfalls einige der bereits beschriebenen Features der Konkurrenz zu Nutze, um die Nutzererfahrung zu optimieren. Freundschaftsanfragen wie in Facebook als Konzept der Vernetzung und Sicherheitsmechanismus für mehr Privatsphäre, Nachrichtenübertragung in Echtzeit wie in Bolt - in anderer Form weiterentwickelt, Transkription wie in Beam - nur für Sprachnachrichten statt Telefonate, der Versand von Nachrichten wie in WhatsApp und der Verzicht auf eine Mobilnummer als Nutzernname für mehr Privatsphäre wie in Threema. In M2 werden diese Features in sorgfältiger Arbeit zu einem eigenständigen Konzept verbunden.

Markt und Wettbewerb (letzter Zugriff: 26. 07. 2019)

1 WhatsApp	https://www.whatsapp.com
2 Telegram	https://telegram.org
3 Viber	https://www.viber.com/de/
4 Facebook Messenger	https://www.messenger.com
5 Snapchat	https://www.snapchat.com/l/de-de/
6 Kik	https://www.kik.com
7 Threema	https://threema.ch/de
8 Skype	https://www.skype.com/de/
9 Beam	https://www.beamplatform.com/
10 Bolt	https://apps.apple.com/de/app/bolt-real-time-live-messaging/id1060154196

Tabelle 3.3: Markt und Wettbewerb (letzter Zugriff: 26. 07. 2019)

3.4 Funktionen

3.4.1 An- und Abmeldung

Empfang

Beim ersten Start der App bzw. wenn kein Nutzer angemeldet ist, startet die Anwendung in der Empfangsansicht (Abb. 3.2). Von hier aus kann der Nutzer einen Account anlegen, falls noch keiner existiert, oder sich mit einem bestehenden Account anmelden.

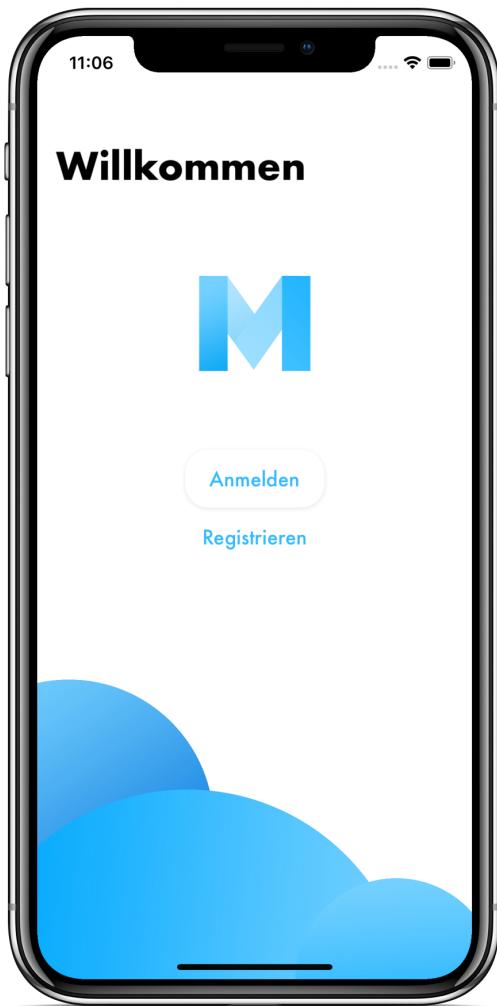


Abbildung 3.2: Empfangsansicht, wenn kein Nutzer angemeldet

Anmeldung und Registrierung

Um die Anmeldung für den Nutzer so unkompliziert wie möglich zu halten, kann sich der Nutzer mit einem Nutzernamen und Passwort registrieren oder anmelden (Abb. 3.3). Von der Nutzung der Mobilnummer als Nutzername wird abgesehen, um eine Bindung des Accounts an ein Gerät oder eine Plattform in zukünftigen Versionen von M2 zu vermeiden. Außerdem sollen aus Datenschutzgründen so wenig nutzerbezogene Daten wie möglich gespeichert werden.

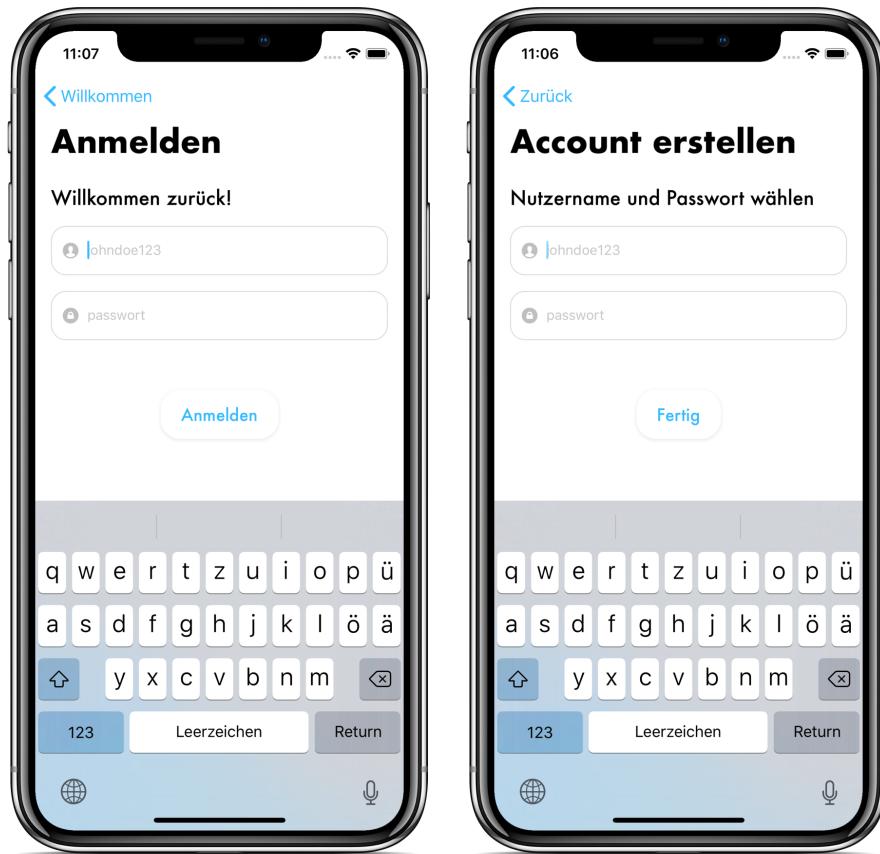


Abbildung 3.3: Anmelde- und Registrierungsansicht

Chatliste

Nach erfolgreicher Anmeldung landet der Nutzer in der Hauptansicht von M2: Der Chatlistenansicht. Ist ein Nutzer bereits angemeldet, startet die App mit dieser Ansicht.

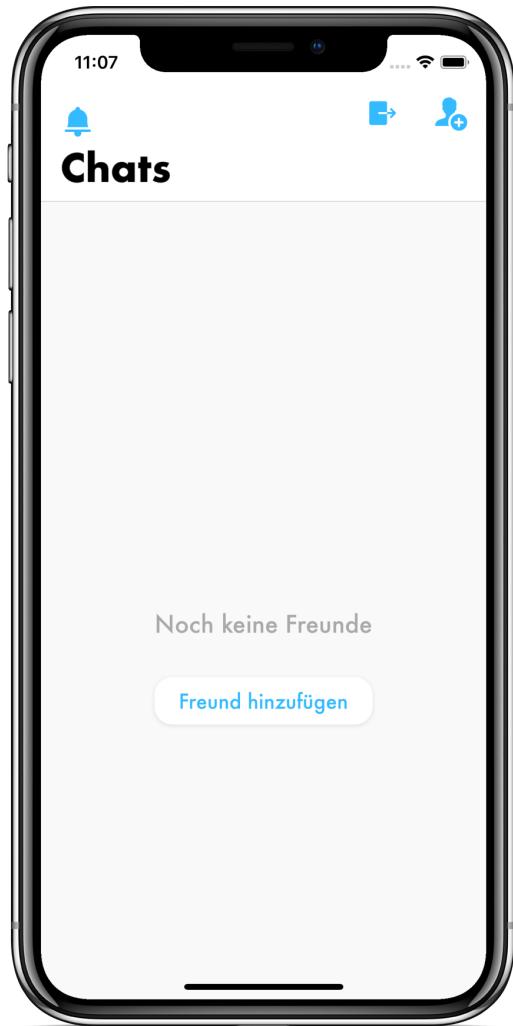


Abbildung 3.4: Leere Chatliste

Abmeldung

Möchte sich ein Nutzer wieder abmelden, betätigt er einen Knopf oben rechts in der Navigationsleiste.

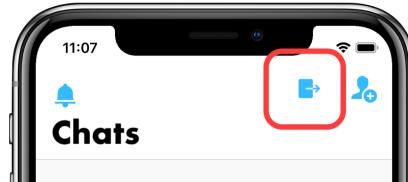


Abbildung 3.5: Knopf zur Abmeldung

Die endgültige Abmeldung erfordert aus Sicherheitsgründen noch eine Bestätigung.



Abbildung 3.6: Hinweis vor Abmeldung

3.4.2 Freundschaftsanfragen

Für die Gewährleistung von Privatsphäre soll die Möglichkeit einer Kontaktaufnahme erst nach einer bestätigten Freundschaftsanfrage gegeben sein.

In einer Liste aus bestehenden Freundschaften werden mögliche Chats dargestellt. In Abbildung 3.4 wird die Liste der Chats für einen Nutzer ohne jegliche gesendete oder bestätigte Freundschaftsanfragen visualisiert.

Freundschaftsanfrage versenden

Um neue Kontaktaufnahmen zu ermöglichen, kann der Nutzer Freundschaftsanfragen versenden. Hierzu befindet sich oben rechts in der Navigationsleiste der Chatlistenanansicht ein Knopf. Hat der Nutzer noch keine bestehenden Freundschaften, befindet sich zusätzlich ein Knopf in der Mitte der Ansicht, um dem Nutzer mitzuteilen, warum diese Ansicht nicht gefüllt ist.

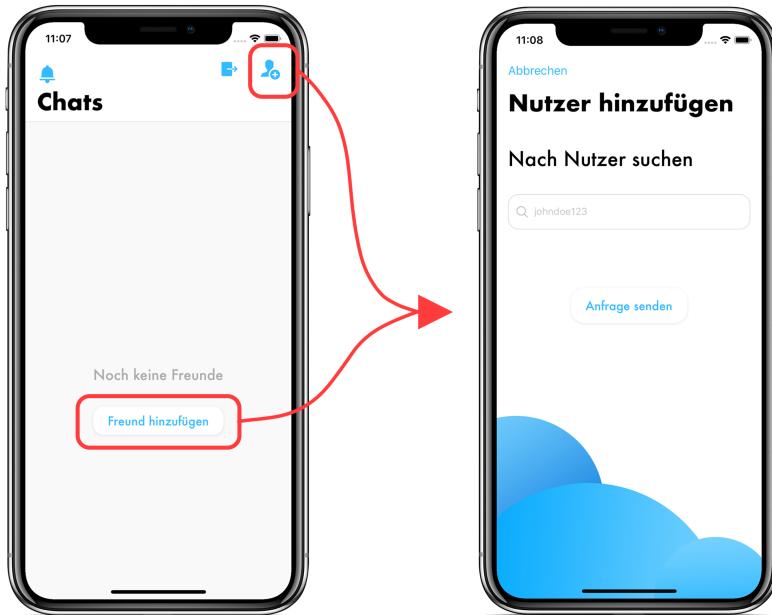


Abbildung 3.7: Öffnen der „Nutzer hinzufügen“-Ansicht

In der folgenden Ansicht wird während der Nutzereingabe überprüft, ob der eingegebene Nutzernname bereits existiert. Wenn ja, wird dies visuell bestätigt (Abb. 3.8). Wird ein anderer Nutzer gefunden, kann die Anfrage durch Betätigen des „Anfrage senden“-Knopfes versendet werden. Der Nutzernname muss sich zuvor auf anderem Wege, außerhalb von M2, mitgeteilt werden.

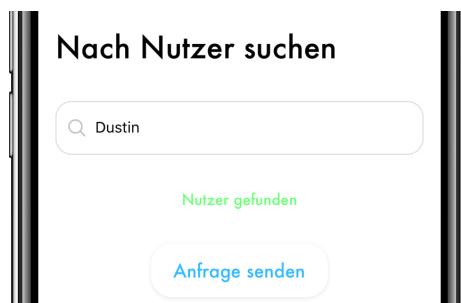


Abbildung 3.8: Bestätigung eines gefundenen Nutzernamens

Freundschaftsanfrage beantworten

Sobald ein Nutzer eine Freundschaftsanfrage erhält, wird dieser benachrichtigt. Befindet sich der Nutzer in der App, erscheint eine kleine Zahl zur Repräsentation der noch offenen Anfragen neben dem Knopf zum Öffnen der Freundschaftsanfragenliste. Befindet sich der Nutzer zu dem Empfangszeitpunkt der Anfrage nicht in der App, erhält dieser eine Push-Benachrichtigung⁸ (Abb. 3.9).



Abbildung 3.9: Benachrichtigung einer Freundschaftsanfrage als Push-Benachrichtigung (l) und innerhalb der App (r)

Der Empfänger kann die Freundschaftsanfrage nun bestätigen oder ablehnen. Hierzu befindet sich oben links in der Navigationsleiste der Chatlistenansicht ein Knopf (Abb. 3.10). Nach der Bestätigung der Freundschaftsanfrage erhält der Absender ebenfalls eine Push-Benachrichtigung (Abb. 3.11).

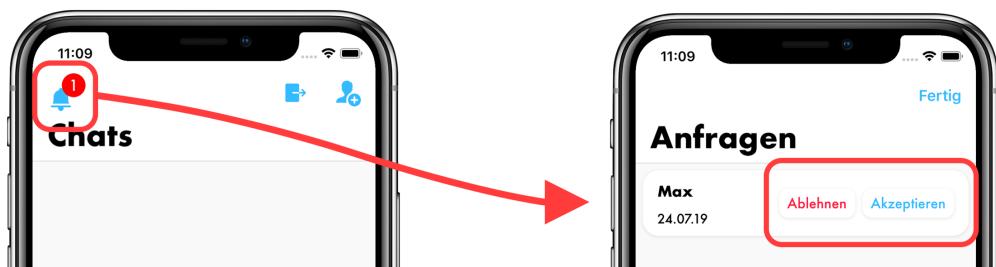


Abbildung 3.10: Beantworten einer Freundschaftsanfrage

⁸Systemmitteilung, die außerhalb einer App im Sperrbildschirm, oder im oberen Bildschirmbereich angezeigt werden kann

3 Anforderungsanalyse

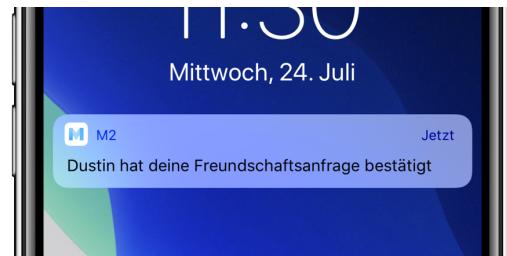


Abbildung 3.11: Push-Benachrichtigung beim Absender nach Bestätigung einer Freundschaftsanfrage

Nach Bestätigung der Freundschaftsanfrage wird der Chat nun bei beiden Nutzern in der Chatliste angezeigt (Abb. 3.12).

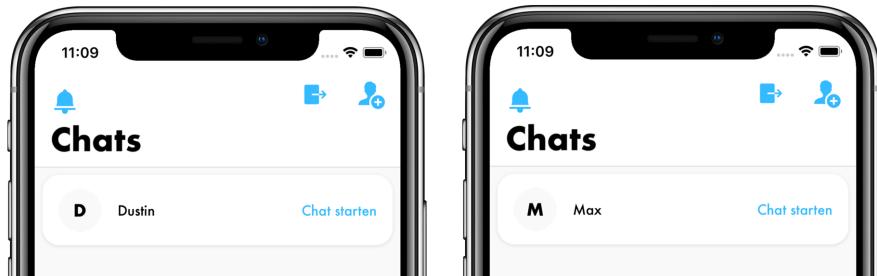


Abbildung 3.12: Chatlistenansicht bei bestehender Freundschaft zwischen Nutzer A (l) und B (r)

Freundschaft rückgängig machen

Soll eine bestätigte Freundschaft rückgängig gemacht werden, kann der Nutzer in der Chatlistenansicht den jeweiligen Chat mit einem Wischen nach links rückgängig machen. Dies ist eine unter iOS übliche Interaktion, die ein Nutzer in der festgesetzten Zielgruppe gewohnt sein sollte ([Apple 2017](#)).

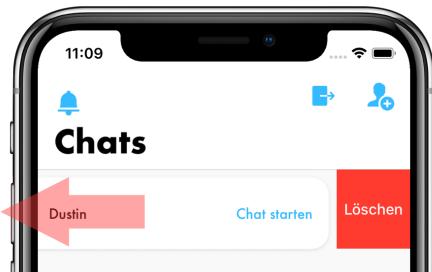


Abbildung 3.13: Rückgängigmachen einer bestehenden Freundschaft (Interaktion mit rotem Pfeil dargestellt)

3.4.3 Nachrichtenanfragen

Sind zwei Nutzer miteinander befreundet, kann eine Konversation begonnen werden. Hierzu existiert in jeder Zeile für einen Chat ein Knopf. Durch Betätigung gelangt der Nutzer in die Nachrichtenansicht (Abb. 3.14).

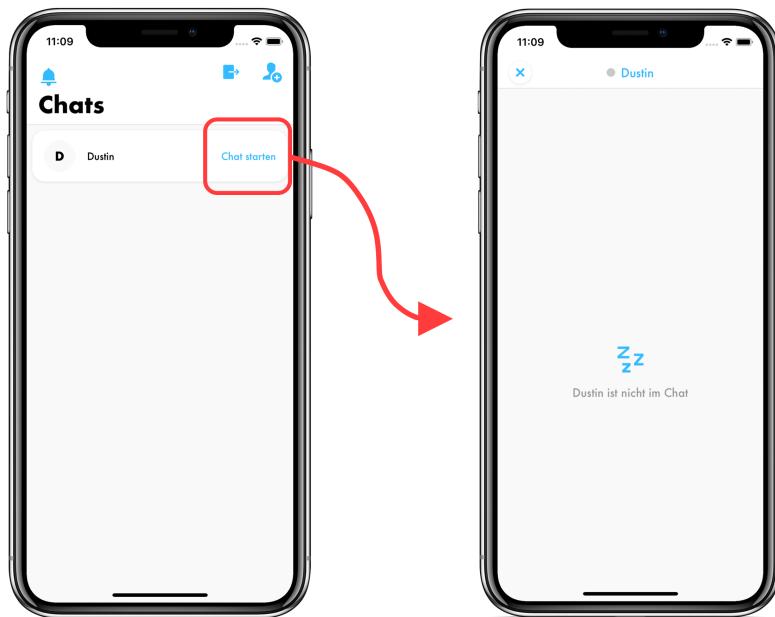


Abbildung 3.14: Öffnen der Nachrichtenansicht

Hat der gegenüberliegende Nutzer nicht ebenfalls bereits die Nachrichtenansicht für diese Konversation geöffnet, wird dies dem ersten Nutzer in der Mitte des Bildschirms mitgeteilt. Es existiert solange kein Textfeld zur Nachrichteneingabe, wie nicht beide Nutzer gleichzeitig in der Nachrichtenansicht sind (Abb. 3.15).



Abbildung 3.15: Nachrichtenansicht, wenn gegenüberliegender Nutzer nicht verfügbar (Platzhalter und ausgeblendetes Textfeld mit rotem Rahmen markiert)

3 Anforderungsanalyse

Damit der gegenüberliegende Nutzer Bescheid weiß, dass ein Freund mit ihm eine Konversation starten möchte, erhält dieser eine Push-Benachrichtigung (Abb. 3.16).



Abbildung 3.16: Push-Benachrichtigung bei gegenüberliegendem Nutzer nach Erhalt einer Nachrichtenanfrage

Öffnet der gegenüberliegende Nutzer nun ebenfalls die Nachrichtenansicht der Konversation, verändert sich die Ansicht des ersten Nutzers. Der Hilfetext in der Mitte des Bildschirms erklärt, dass noch keine Nachrichten gesendet wurden und für beide Nutzer wird nun ein Textfeld zur Nachrichteneingabe angezeigt (Abb. 3.17). Beim Verlassen der Nachrichtenansicht eines Nutzers verändert sich der Zustand wieder zurück.

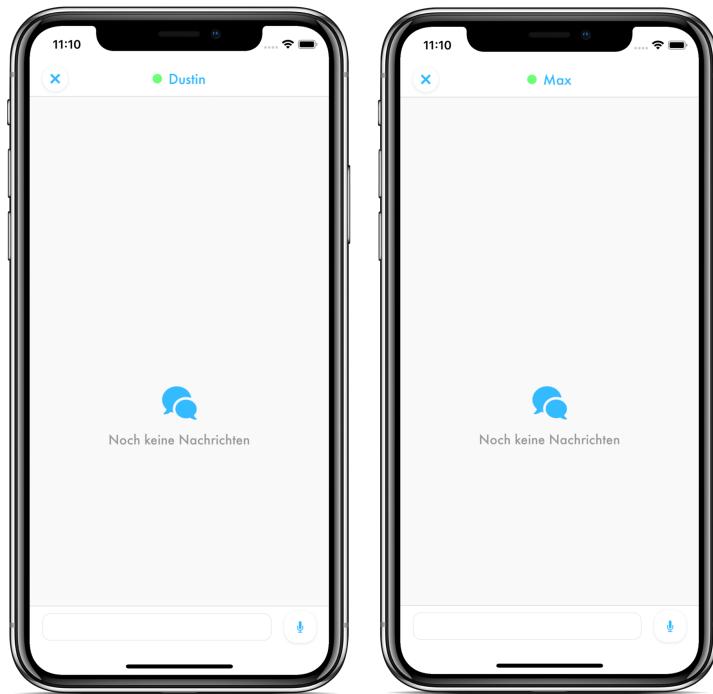


Abbildung 3.17: Nachrichtenansicht, wenn beide Nutzer verfügbar sind

3.4.4 Nachrichtenaustausch

Sind beide Nutzer in der Nachrichtenansicht der Konversation, kann die Konversation begonnen werden.

Textnachrichten

Zum Versenden einer Textnachricht, kann die Eingabe in einem Textfeld begonnen werden.



Abbildung 3.18: Nachrichtenansicht für Nutzer A (l) und B (r)

Fängt ein Nutzer an zu tippen, erscheint die Nachricht Stück für Stück wie sie eingetippt wird bei dem gegenüberliegenden Nutzer.

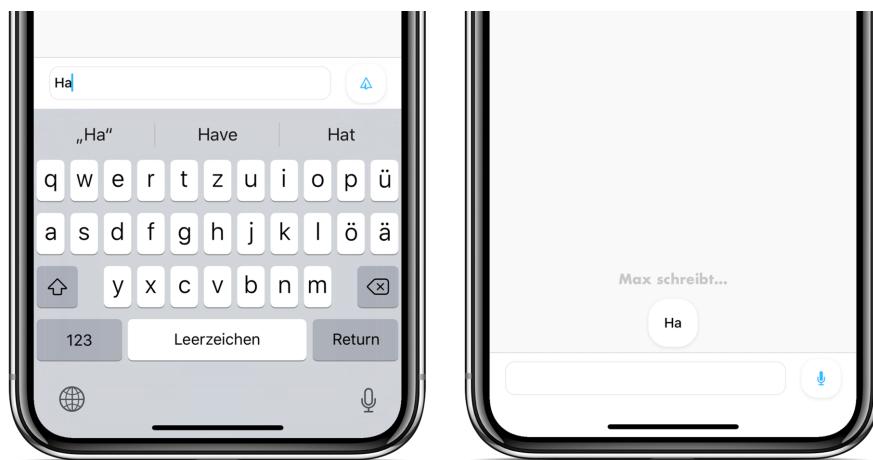


Abbildung 3.19: Eintippen einer Nachricht von Nutzer A (l) an B (r)

3 Anforderungsanalyse

Ist die Nachricht fertig eingetippt, kann sie über einen Knopf neben dem Textfeld abgeschickt werden. War das Versenden der Nachricht erfolgreich, wird die Nachricht in Form einer Zelle dargestellt. Der Inhalt der eigenen Nachricht bleibt für den Nutzer allerdings verborgen und wird nur dem Empfänger angezeigt, um den Fokus der Konversation auf die gegenüberliegende Seite zu richten.

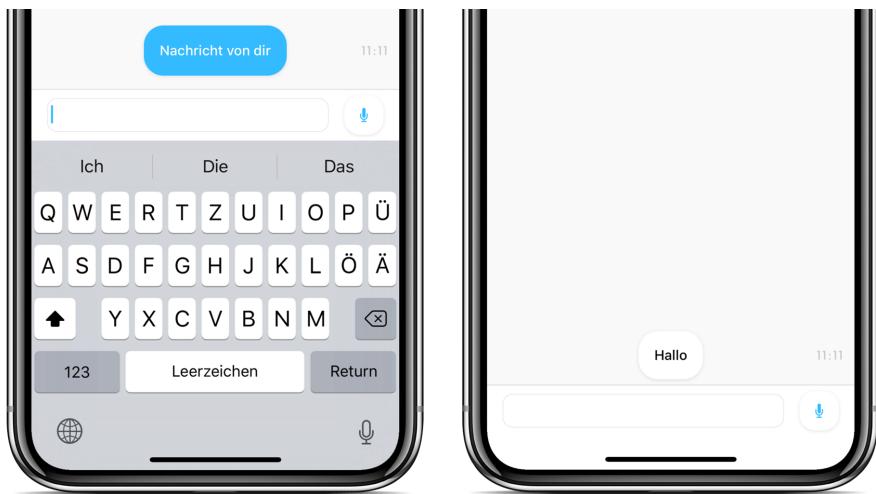


Abbildung 3.20: Abgeschickte Textnachricht von Nutzer A (l) an B (r)

Sprachnachrichten

Der Knopf zum Versenden einer Textnachricht verändert während der Texteingabe seinen Zustand. Wurde noch keine Texteingabe begonnen, dient er nicht zum Absenden einer Textnachricht, sondern zum Aufnehmen einer Sprachnachricht. Dies wird dem Nutzer mithilfe eines dazugehörigen Symbols signalisiert. Die Sprachnachricht wird solange aufgenommen, wie der Knopf gedrückt gehalten wird. Lässt der Nutzer den Knopf los, wird die Sprachnachricht abgesendet. Auch hier bleibt dem Absender der Inhalt der Nachricht verborgen (Abb. 3.21).



Abbildung 3.21: Abgeschickte Sprachnachricht von Nutzer A (l) an B (r)

Nach Erhalt einer Sprachnachricht kann der Empfänger sich diese nun über einen Wiedergabeknopf anhören. Der Abspielpunkt innerhalb der Audiomeldung wird mittels eines Fortschrittsbalkens angegeben.

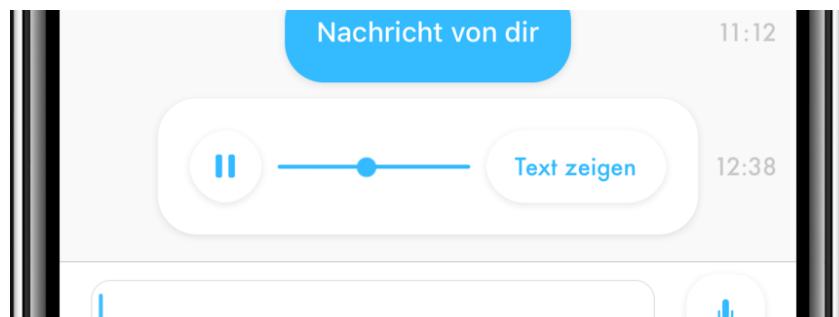


Abbildung 3.22: Anhören einer Sprachnachricht

3 Anforderungsanalyse

Des Weiteren hat der Empfänger die Möglichkeit, den Inhalt der Sprachnachricht als textuelle Transkription zu sehen. Hierzu befindet sich ein Knopf rechts vom Fortschrittsbalken der Sprachnachricht (Abb. 3.23). Bei wiederholtem Betätigen des Knopfes wird die Transkription wieder versteckt.

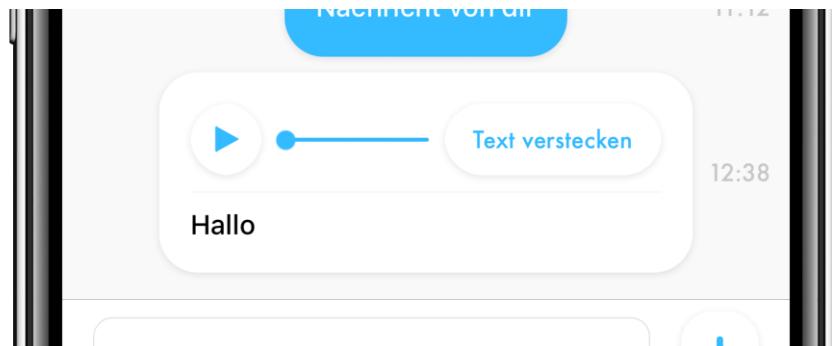


Abbildung 3.23: Transkription einer Sprachnachricht

Konversation verlassen

Möchte ein Nutzer die Konversation verlassen, kann er dies durch Nutzung eines Knopfs in der oberen linken Ecke der Nachrichtenansicht.

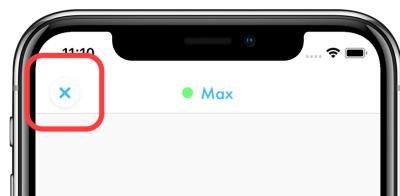


Abbildung 3.24: Verlassen einer Konversation

Das endgültige Verlassen erfordert aus Sicherheitsgründen noch eine Bestätigung, da der Verlauf der Konversation dadurch verloren geht.

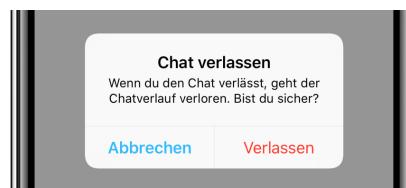


Abbildung 3.25: Verlassen einer Konversation

4 Design

4.1 Richtlinien

Während der Entwicklung einer iOS-App ist es wichtig, die Designrichtlinien zu beachten. Apple hält hierfür einen umfangreichen Ratgeber bereit, der dem Entwickler helfen soll, die optimale Nutzererfahrung zu erreichen ([Apple 2019](#)). Wird geplant, die App später über den Apple App Store zum Download anzubieten, wird sie einer Prüfung durch Apple unterzogen. Apple Mitarbeiter prüfen jede einzelne App persönlich, bevor sie zum Download im App Store freigegeben wird. Verstößt eine App zu gravierend gegen Apples Designrichtlinien, besteht sie diese Prüfung nicht und der Entwickler muss die vorliegenden Mängel ausbessern. Ein Verstoß könnte bspw. sein, ein UI-Element¹ auf eine ganz andere Art und Weise zu nutzen, als es vorgesehen wurde. Nutzt man stets die vorgegebenen Programmbibliotheken für UI-Elemente, verzichtet auf die Nutzung von „Private APIs“² und versucht sich grob an die Designrichtlinien zu halten, sollte dies jedoch kein Problem darstellen. Die Programmbibliotheken für UI-Elemente unter iOS sind außerdem so umfangreich, dass es äußerst selten vorkommt, dass ein Element für einen Anwendungsfall fehlt. Und wenn dies doch eintreten sollte, ist es ebenfalls erlaubt, eigene UI-Elemente zu entwickeln. Auch wenn M2 vorerst nicht für die Nutzung in einer Produktionsumgebung ausgelegt ist, ist es durchaus sinnvoll, gegebene Richtlinien zu beachten. Diese stellen eine gute Orientierung für die Umsetzung intuitiv bedienbarer Interaktionen dar und vereinfachen die Fertigstellung für eine Produktionsumgebung in einem späteren Entwicklungsstand der App.

¹ „User Interface“-Element: Element zur Gestaltung der Benutzeroberfläche

² Versteckte „Application Programming Interfaces“: Teile der, von Apple vorgegebenen, Schnittstellen der Programmbibliotheken, die offiziell nicht zur Nutzung freigegeben wurden, aber über Umwege trotzdem erreichbar sind

4.2 Benutzeroberfläche

Typografie

Riesen Titel	Futura Bold	32.0pt
Großer Titel	Futura Medium	27.0pt
Titel	Futura Bold	17.0pt
Knopf	Futura Medium	17.0pt
Kleiner Titel	Futura Medium	14.0pt
Text	SF Pro Text Regular	14.0pt

Abbildung 4.1: Übersicht genutzter Schriftarten

Eine wegweisende designtechnische Entscheidung bei der Konzeption einer App ist die Wahl der Schriftart. Für die Nutzung in Titeln kann eine aufwendiger gestaltete Schriftart genutzt werden, während es sich für Fließtexte eher empfiehlt, eine schlichtere Schriftart zu verwenden.

Für Titel (bspw. in Navigationsleisten) kam in M2 „Futura“ in diversen Schriftschnitten³ zum Einsatz.

Für Texte (bspw. Nachrichten in der Konversation) wurde die iOS-Systemschriftart „San Francisco“ eingesetzt. Diese Schriftart wurde eigens von Apple entwickelt und bietet besonders gute Lesbarkeit bei modernem Aussehen in Apps.

³Verschiedene Ausführungen einer Schriftart mit unterschiedlichen Breiten der genutzten Strichführung

Farbpalette



Abbildung 4.2: Übersicht genutzter Farben

Die Farbwahl in M2 ist schlicht gehalten. Ein helles Blau wirkt freundlich und angenehm, während weiße oder hellgraue Hintergründe klare Kontraste bilden. Die in ähnlichem Kontext üblicherweise verwendeten Farbtöne grün und rot für bestätigende oder ablehnende Interaktionen schaffen eine intuitive Bedienung.

UI-Elemente

Um M2 ein modernes und freundliches Design zu geben, sind viele UI-Elemente eigens gestaltet. Es wurden dafür zwar Elemente aus der UI-Bibliothek von Apple⁴ verwendet, aber Hintergrundfarben, Texte, Eckenrundung, etc. angepasst.

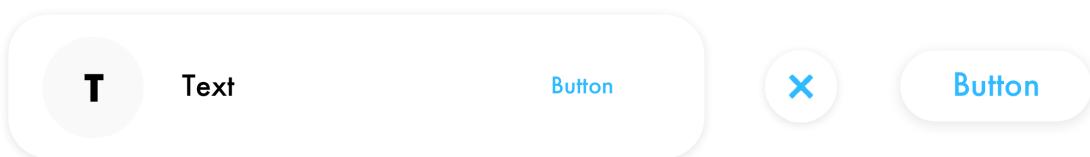


Abbildung 4.3: Übersicht der eigens gestalteten UI-Elemente (Teil 1)

⁴UIKit (Abschnitt 6.3.1)

4 Design

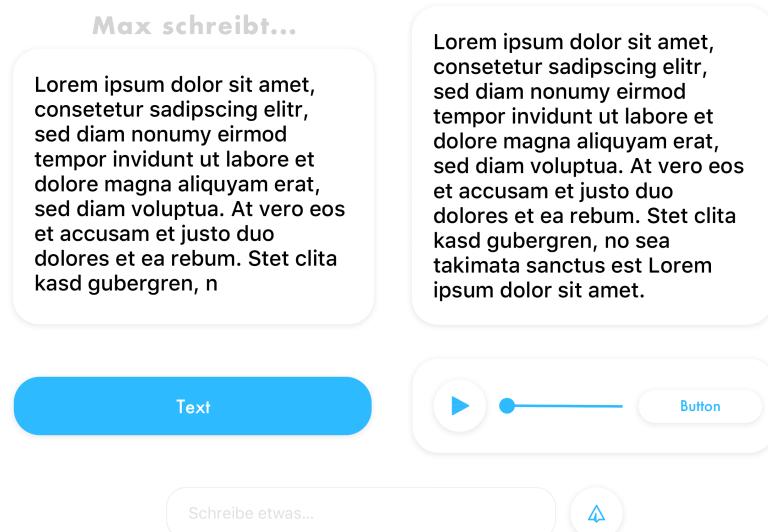


Abbildung 4.4: Übersicht der eigens gestalteten UI-Elemente (Teil 2)

Der grundlegende Aufbau aller Elemente basiert auf einem kartenähnlichen Design mit abgerundeten Ecken und leichtem Schlagschatten. Diese Designform findet sich ebenfalls in Apples aktuellen Apps wieder.



Abbildung 4.5: Apple Health (l) und Apple App Store (r) unter iOS 12
Quellen: [Apple \(2019\)](#) und [Apple \(2019\)](#)

Für die Symbole in Buttons (z.B. der Wiedergabe-Knopf in Abb. 4.4) wurden Icons des Anbieters [Icons8 \(2019\)](#) verwendet.

4.3 Name

Der Name der aus dieser Arbeit resultierenden iOS-App lautet »M2«. Er steht für „Messenger 2.0“ und bezeichnet die Weiterentwicklung der altbekannten Form eines Messengers⁵ mit innovativen Funktionalitäten. Die Ziffer »2« im Namen steht außerdem für die Konzentration beider Nutzer aufeinander. Der Name ist kurz, leicht zu merken und spiegelt in seiner niedergeschriebenen Veranschaulichung das moderne Gesicht der App wider.

4.4 Logo



Abbildung 4.6: Entwicklungsphasen des Logos

Bei der Entwicklung des Logos wurde die Farbpalette der festgelegten CI übernommen und der Name so minimalistisch wie möglich dargestellt. Auf eine vollständige Ausschreibung des Namens wurde außerdem verzichtet. Vielmehr kam ab Entwicklungsschritt zwei (mittig in Abb. 4.6) die Idee, die Ziffer »2« des Namens in römischer Zahlschrift darzustellen und durch Variation von Kontrasten trotzdem mit dem Buchstaben »M« zu verknüpfen.

⁵In der Informatik werden erste Versionen oft mit der Nummer 1.0 versehen, größere Weiterentwicklungen oft mit 1.0er-Sprüngen, demnach 2.0

5 Softwarearchitektur

5.1 Entwicklungsstrategie

Es wurde nach der Entwicklungsstrategie „Bottom-Up“ ([Lackes & Siepermann 2019](#)) gearbeitet, welche die Struktur des Codes während der Entwicklung entstehen lässt. Die Hauptbestandteile des entstehenden Projektes werden vorab definiert, weitere Klassen und Funktionen werden während der Entwicklung in dem Moment ergänzt, in dem sie benötigt werden. Dies ermöglicht eine schnelle Entwicklung, birgt allerdings die Gefahr, öfter Ansätze auch wieder verwerfen und umstrukturieren zu müssen. Im Gegensatz dazu steht die Variante „Top-Down“ ([Müller 2019](#)), welche eine exakte Planung der Struktur im Voraus beschreibt. Dies kann insbesondere für Großprojekte an denen viele Leute gleichzeitig arbeiten sehr sinnvoll sein. Für diese Arbeit hat es sich allerdings nicht angeboten, da einige Entwurfsmuster und Bibliotheken getestet werden mussten, bevor feststand wie genau sie in die Struktur integriert werden.

5.2 Entwurfsmuster

5.2.1 MVC

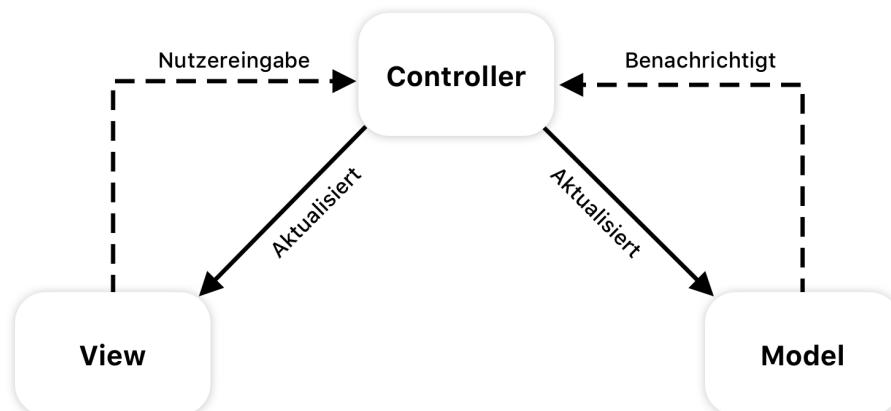


Abbildung 5.1: Visualisierung einer oftmals verwendeten Ausführung des Entwurfsmusters MVC (indirekte Assoziationen mit gestrichelten Pfeilen)

MVC¹ wurde 1979 von dem norwegischen Forscher und Informatiker Trygve Reenskaug in der Programmiersprache Smalltalk beschrieben ([Wikipedia 2019](#)) und ist heutzutage einer der bewährtesten Entwurfsmuster der objektorientierten Programmierung ([Lahres & Rayman 2009](#)). In Abbildung 5.1 ist eine oftmals verwendete Ausführung des Entwurfsmusters dargestellt. Diese und alle weiteren Grafiken dieser Arbeit sind selbst erstellt, sofern keine Quellenangaben im Bilduntertitel vorliegen. Laut [Lahres & Rayman \(2009\)](#) ist MVC keine vollständige Softwarearchitektur. Wie MVC implementiert wird steht unabhängig von dem Muster. Im Folgenden wird zum besseren Verständnis für den weiteren Verlauf der Arbeit ein grundlegender Überblick gegeben.

Das Entwurfsmuster teilt die Codestruktur in drei Teile auf: **Model** ([Datenmodell](#)), **View** ([Präsentation](#)) und **Controller** ([Programmsteuerung](#)) ([Wikipedia 2019](#)). Es soll somit eine übersichtlichere Wartung des Codes erwirken. Es gibt diverse Ausführungen des MVC, welche sich zumeist im Aufbau der Interaktion der drei Teile untereinander unterscheiden. In diesem Kapitel wird die Verwendung unter iOS behandelt.

Model (Datenmodell)

Das Model beschreibt das darzustellende Objekt. Es ist der Ursprung der angezeigten Daten und sollte darstellbar sein ([Lahres & Rayman 2009](#)). Das Model hat nichts mit der Präsentation zu tun und kann daher auch mehr Informationen als angezeigt werden müssen enthalten.

View (Präsentation)

Die View ist die darzustellende Ansicht. In gängigen Programmen heutzutage beinhaltet diese Bedienelemente wie Textfelder, Knöpfe, Regler, usw. Im Entwurfsmuster des MVC wird der Inhalt des Models in der View dargestellt. Dies erlaubt durchaus die Darstellung desselben Models auf mehreren, unterschiedlichen Views.

Controller (Programmsteuerung)

Der Controller ist das Bindeglied zwischen Model und View (Abb. 5.1). Er hat die Aufgabe, Eingaben durch die View zu verarbeiten und an das Model weiterzugeben ([Lahres & Rayman 2009](#)). Umgekehrt werden Veränderungen des Models verarbeitet und an die View weitergegeben.

¹Model View Controller

5.2.2 Observer Pattern

Zur optimalen Nutzung von MVC müssen dem Controller die Änderungen des Models mitgeteilt werden, damit die View entsprechend aktualisiert werden kann. Hierzu dient das **Observer Pattern**. Auch dieses findet seinen Ursprung in der Entwicklung des MVC und erlangte seinen Bekanntheitsgrad durch die häufige Nutzung in der Entwicklung mit der Programmiersprache Java ([Ullenboom 2011](#)).

Im Observer-Pattern gibt es zwei maßgebliche Objektarten: **Observable** und **Observer**.

Observable

Das **Observable**-Objekt ist ein beobachtbares Objekt. Es kann von beliebig vielen **Observer**-Objekten beobachtet werden und benachrichtigt diese, wenn es sich verändert hat.

Observer

Das **Observer**-Objekt ist Beobachter eines **Observable**-Objektes. Es wird vom **Observable**-Objekt über eine Änderung benachrichtigt.

Um dieses Pattern in Verbindung mit der Nutzung von MVC zu veranschaulichen, folgt ein Beispiel.

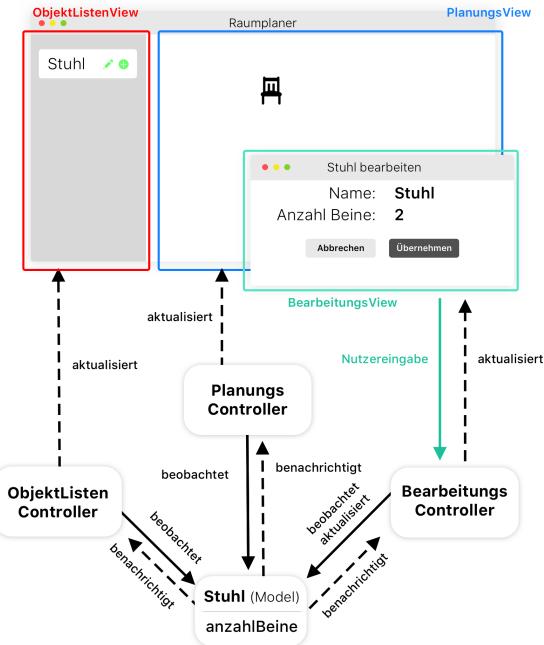


Abbildung 5.2: Visualisierung des Observer-Patterns

In diesem Beispiel wird die Entwicklung einer Software zur Raumplanung konzipiert. Wie genau innerhalb der konzipierten Software interagiert wird ist für dieses Beispiel nicht von Bedeutung. Zugunsten einer übersichtlicheren Veranschaulichung wurde auf die Darstellung irrelevanter Aktionen verzichtet. Es wurden außerdem teils deutsche, teils englische Bezeichner gewählt, um einige Aspekte besser hervorheben zu können, in einer tatsächlichen Implementierung würde eher eine komplette englische Schreibweise bevorzugt werden.

Die Software hat drei Ansichten:

- **ObjektListView** zur Darstellung einer Listenansicht aus Objekten zur Raumplanung.
- **PlanungsView** zur Darstellung der eigentlichen Planung und Nutzung der Objekte aus **ObjektListView**.
- **BearbeitungsView** zur Darstellung einer Bearbeitungsansicht für die Objekte aus **ObjektListView**.

Des Weiteren gibt es für jede der drei Ansichten die dazugehörigen Controller **ObjektListenController**, **PlanungsController** und **BearbeitungsController**, welche die Änderungen der Views verarbeiten und an das Model **Stuhl** weiterreichen.

Das Model **Stuhl** repräsentiert das darzustellende Datenobjekt. Es hat einen veränderbaren Namen und eine veränderbare Stuhlbeinanzahl.

Als Teil des Observer-Patterns sind die drei Controller Beobachter des Objektes **Stuhl**. Das Objekt ist so implementiert, dass es all seine Beobachter über jede Veränderung des Namens oder der Stuhlbeinanzahl informiert.

In folgender Visualisierung 5.3 wurden durch den Nutzer Objektname und Stuhlbeinanzahl modifiziert.

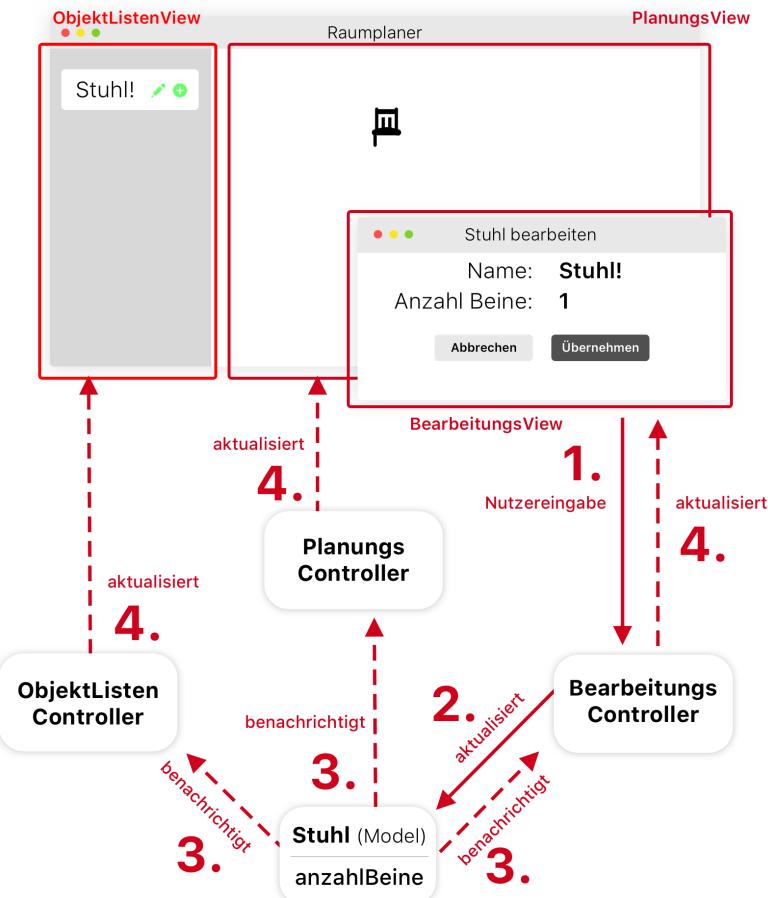


Abbildung 5.3: Ausgelöster Ablauf durch Nutzereingabe in BearbeitungsView mit Observer-Pattern

Vorteile

Das Observer-Pattern vermeidet die händische Aktualisierung der Darstellung, welche bei der Implementierung schnell vergessen werden kann. Ist ein Beobachter erst

einmal registriert, muss sich der Entwickler keine Gedanken mehr um das Anstoßen der Darstellungsaktualisierung machen. Eine Änderung wird im Model vorgenommen und der Rest geschieht von allein.

Nachteile

Oft werden Views unnötig aktualisiert, da sie über jede Änderung im Model informiert werden, aber einige Veränderungen im Model gar nicht in dieser View dargestellt werden. In dem Beispiel (Abb. 5.3) werden sowohl `ObjektListView` als auch `PlanungsView` durch eine Änderung im Model aktualisiert. Die Veränderung des Attributs `Name` braucht `PlanungsView` aber beispielsweise nicht zu interessieren, da die Information dort nicht dargestellt wird. Die View wird trotzdem aktualisiert. Ähnliches gilt für das Attribut `Anzahl Beine`, welches wiederum nicht für `ObjektListView` interessant ist und trotzdem auch dort eine Aktualisierung hervorruft. Dies kann insbesondere problematisch werden, wenn z.B. in `ObjektListView` noch eine Animation bei Veränderung der angezeigten Daten hinzugefügt werden würde, denn dann würde bei einer Veränderung der Stuhlbeinanzahl durch den Nutzer eine Animation auftreten, obwohl sichtlich für den Nutzer in der Darstellung nichts verändert wurde. Dies könnte dann als verwirrendes Flackern interpretiert werden.

5.2.3 ViewController

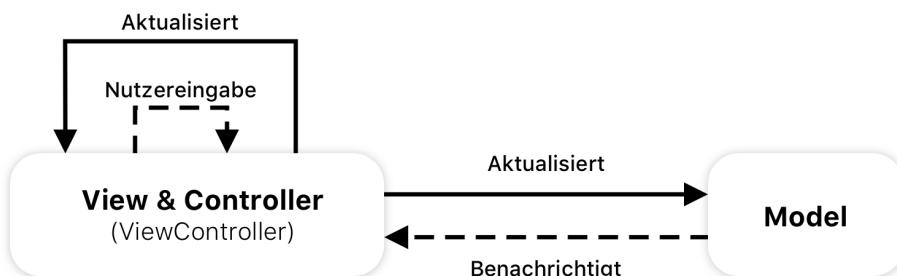


Abbildung 5.4: Visualisierung von MVC mit einem ViewController

Ein ViewController ist die Vereinigung einer View und eines Controllers. Die zwei Teile stellen in diesem Fall ein einheitliches Objekt dar, die View ist Teil des Controllers. Dieses Objekt ist als `UIViewController` Bestandteil der UI-Bibliothek von Apple (2019)².

²UIKit (Abschnitt 6.3.1)

5.2.4 MassiveViewController

In der iOS-Entwicklung kann die Vereinigung von View und Controller (Abschnitt 5.2.3) bei mittelgroßen bis großen Projekten zu einem Problem werden. Der Controller agiert nicht mehr als eigenständiges Objekt, stattdessen wird seine Logik im gleichen Code mit der View implementiert. Damit ist der ViewController sowohl für die Präsentation als auch für die Programmsteuerung zuständig und lässt Implementierungen schnell über 5000 Codezeilen groß werden. Die Wartbarkeit des Codes wird somit bei steigender Projektgröße immer schlechter und unübersichtlicher. Eine angenehmere Wartbarkeit ist bei einer maximalen Anzahl von ca. 400 Codezeilen gewährleistet³. Aus diesem Grund bekam die Nutzung des MVC Entwurfsmusters unter iOS den Spitznamen „*MassiveViewController*“ ([Andersson 2018](#)).

5.2.5 MVVM

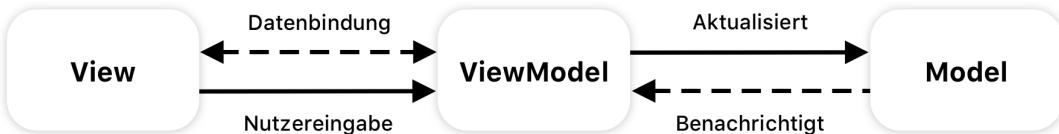


Abbildung 5.5: Visualisierung des MVVM Entwurfsmusters

Zur Vermeidung des „*MassiveViewControllers*“ (Abschnitt 5.2.4) unter iOS hat sich innerhalb der letzten Jahre immer mehr das MVVM⁴ Entwurfsmuster als Variante des MVC durchgesetzt. Es findet seinen Ursprung in einem Blog-Post des Microsoft MVP⁵ Grossman (2005), als eine Spezialisierung des von Fowler (2004) entwickelten „*Presentation Model*“. Das Pattern stammt ursprünglich aus der Softwareentwicklung im Bereich von WPF⁶-Anwendungen und wird zur Vermeidung von Abhängigkeiten eingesetzt ([Kühnel 2012](#)).

ViewModel

Eingeordnet wird das ViewModel an ähnlicher Stelle wie der Controller in MVC und bildet das Bindeglied zwischen View und Model. Es trägt allerdings nicht deshalb seinen Namen, sondern weil es die View in ihrem Aussehen als Struktur, also als Model beschreibt. Alle Bedienelemente einer View werden mithilfe des ViewModels beschrieben. Es enthält das darzustellende Model als Instanz und trennt so klar die Ansicht von dem Model.

³Diese Werte stammen aus eigener langjähriger Berufserfahrung als iOS-Entwickler

⁴Model View ViewModel

⁵Microsoft Most Valuable Professional; Die höchste Auszeichnung von [Microsoft](#) (2019)

⁶Windows Presentation Foundation

Beispielhaft wird in folgender Darstellung auf eine mögliche Repräsentation der **BearbeitungsView** durch ein **ViewModel** eingegangen.

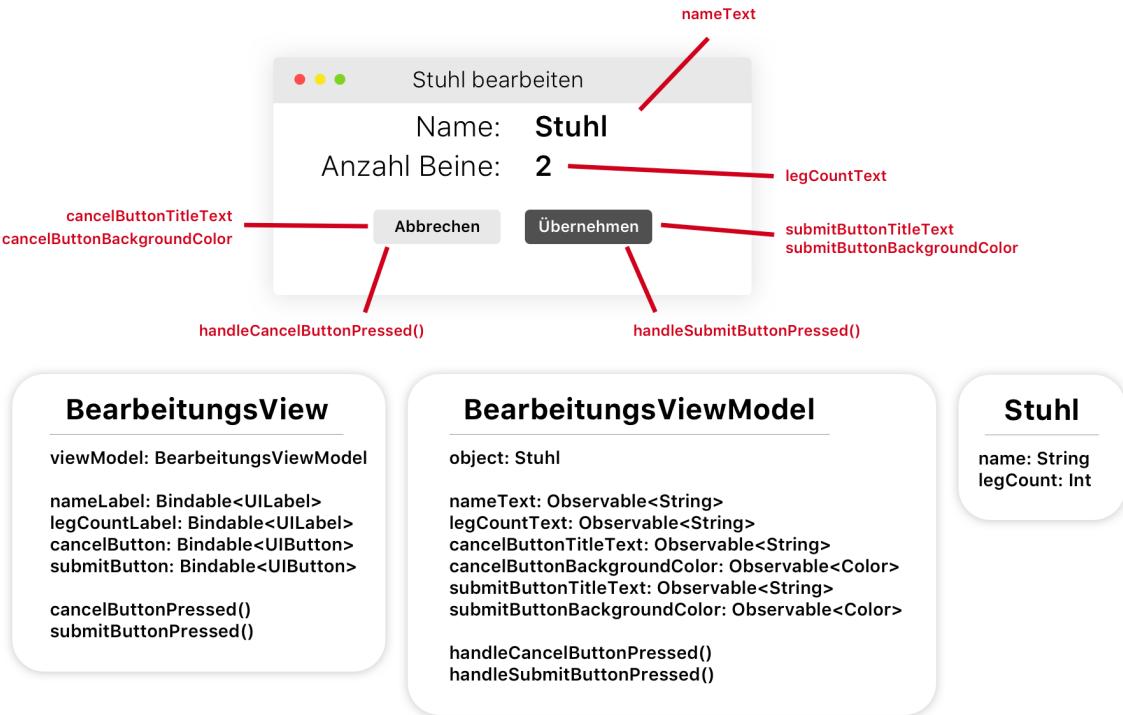


Abbildung 5.6: Visualisierung eines ViewModels für die BearbeitungsView

Wie in Abbildung 5.6 erkennbar, beinhaltet das ViewModel sowohl das Objekt **Stuhl** als auch Variablen für jedes Attribut der View und Funktionen zur Behandlung von Aktionen der beiden Knöpfe *Abbrechen* und *Übernehmen*. Das ViewModel wiederum ist Teil der View. Die einzelnen Attribute des ViewModels sind außerdem **Observable**, dies ist ein Teil des Datenbindungsmechanismus.

Datenbindung

Damit Änderungen des ViewModels an das Model weitergegeben werden, benötigt es einen Datenbindungsmechanismus zwischen View und ViewModel.

Ein gängiger Ansatz zu diesem Zweck ist das **Two-Way-Binding** (Katoch 2017). Jedes der Attribute des ViewModels muss **Observable** sein, also beobachtbar. Des Weiteren müssen alle Bedienelemente der View **Bindable** sein, also die Möglichkeit bieten, an ein Attribut gebunden werden zu können. Somit kann eine Bindung zwischen jedem Bedienelement mit dem dazugehörigen Attribut aus dem ViewModel erzeugt werden. Auf der einen Seite werden so Änderungen von Attributen des ViewModels an die View weitergegeben, auf der anderen Seite werden auch Veränderungen

in der View direkt an das ViewModel weitergegeben. Der Zustand zwischen View und ViewModel wird so synchron gehalten.

Ein solcher Datenbindungsmechanismus wird von iOS in der aktuellen Version leider nicht angeboten. Es gibt zwar Bibliotheken Dritter wie [RxSwift \(2019\)](#), welche diese Funktionalität durch eine Swift-Implementierung reaktiver Programmierparadigmen bieten, diese sind jedoch sehr umfangreich und für die Nutzung innerhalb dieses Projektes zu speziell. In der aktuellen Vorabversion von Apples neuem Betriebssystem iOS 13, welches im Herbst 2019 erscheinen soll ([Apple 2019](#)) gibt es allerdings eine neue Bibliothek mit dem Namen **Combine**, welche genau diesen Mechanismus bereitstellt ([Apple 2019](#)). Auf die Nutzung dieser Bibliothek wurde in diesem Projekt allerdings ebenfalls verzichtet, da diese zum Zeitpunkt dieser Arbeit noch nicht final veröffentlicht wurde.

5.3 Architektur der App

5.3.1 Entwurfsmuster

Der Großteil der Views in M2 besteht aus Listenansichten. Diese kommen u.a. in der Chatlistenansicht (Abb. 3.12), der Freundschaftsanfragenansicht (Abb. 3.10) oder der Nachrichtenansicht (Abb. 3.21) zum Einsatz. Hier benötigen nicht viele unterschiedliche Attribute eine Aktualisierung und ein Ansatz mit **Two-Way-Binding** (Abschnitt 5.2.5) wäre aufgrund der aktuell noch fehlenden Implementierung von Apples Seite und der Menge an daraus resultierendem eigenem Code unverhältnismäßig viel Aufwand. Nichtsdestotrotz soll das Entstehen eines **MassiveViewControllers** (Abschnitt 5.2.4) auf jeden Fall vermieden werden.

Daher wird in M2 das MVVM Entwurfsmuster verwendet. Aus in Abschnitt 5.2.5 genannten Gründen wird für den Datenbindungsmechanismus aber kein **Two-Way-Binding**, sondern das Observer-Pattern eingesetzt.

Im folgenden Sequenzdiagramm 5.7 wird der Ablauf zwischen ViewController, ViewModel und Model in M2 visualisiert.

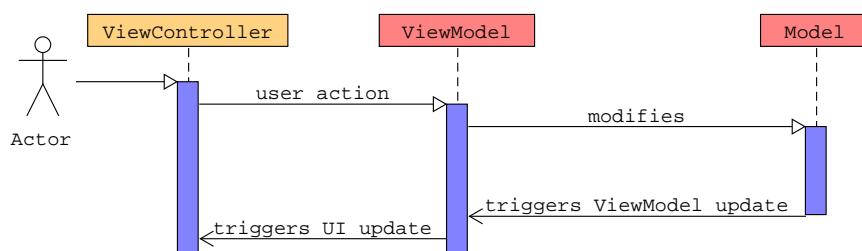


Abbildung 5.7: Sequenzdiagramm für die Nutzung des MVVM-Entwurfsmusters in M2

5.3.2 Models

In M2 werden folgende Objekte als Models definiert:

- Chat
- User
- Message
- TextMessage
- VoiceMemoMessage
- Sound

Die Objektarten TextMessage und VoiceMemoMessage nutzen Vererbung.

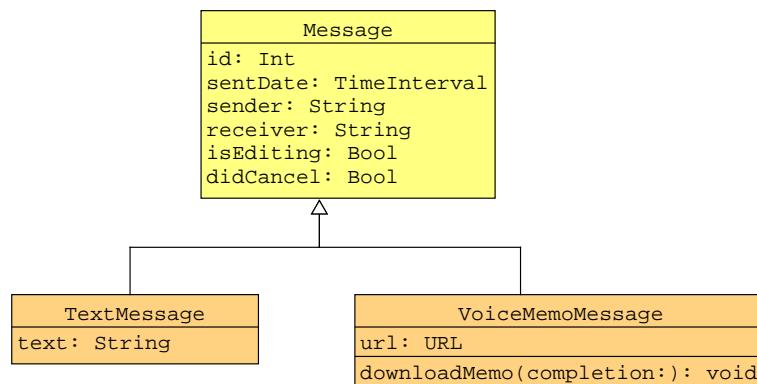


Abbildung 5.8: Klassendiagramm für die Model-Klasse einer Nachricht

Außerdem gibt es diverse Model-Klassen für die Repräsentation der Befehle zwischen App und Backend, welche aufgrund ihrer Vielzahl hier nicht aufgelistet werden. Im Anhang auf der beigefügten CD (Abschnitt A.1) findet sich eine umfangreiche HTML-Dokumentation aller Klassen. Auf die Entwicklung des Backends wird außerdem in Abschnitt 6.4 tiefer eingegangen.

5.3.3 ViewController

Ansichten werden in Form von **ViewControllers** dargestellt.
In M2 werden folgende Objekte als ViewController definiert:

Liste der ViewController in M2	
Name	Beschreibung
IntroViewController	Empfangsansicht beim ersten App-Start
LoginViewController	Anmeldeansicht
ChatsViewController	Chatlistenansicht
MessagesViewController	Nachrichtenansicht
AddUserViewController	Ansicht zum Versenden einer Freundschaftsanfrage
FriendsRequestsViewController	Freundschaftsanfragenliste

Tabelle 5.1: Liste der ViewController in M2

5.3.4 ViewModels

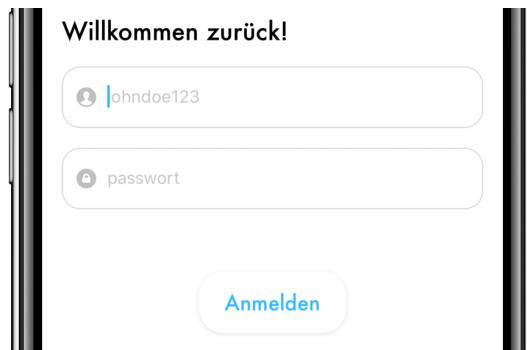


Abbildung 5.9: Anmeldeansicht in M2

Zu jedem ViewController aus der Tabelle 5.2 existiert ein dazugehöriges ViewModel, welches die Attribute des ViewControllers beinhaltet und ausgelöste Nutzeraktionen behandelt. In der obigen Abbildung 5.9 ist die Anmeldeansicht in M2 zu sehen. Diese wird mit dem LoginViewController dargestellt. Das folgende Beispiel veranschaulicht mit einem Klassendiagramm einen Entwurf vom LoginViewController und einem dazugehörigen LoginViewModel.

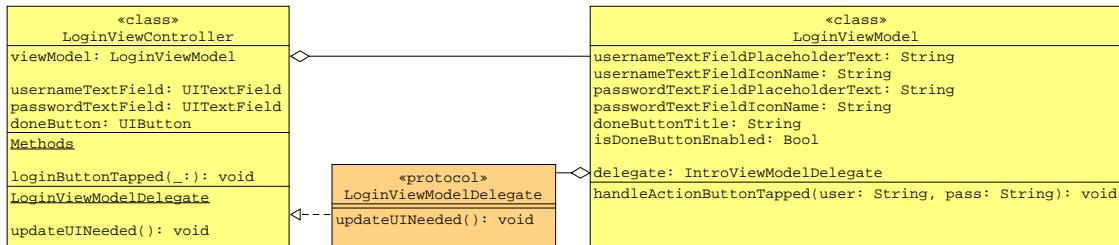


Abbildung 5.10: Klassendiagramm für einen Entwurf von LoginViewController und LoginViewModel

Über das Protokoll `LoginViewModelDelegate` kommuniziert das `LoginViewModel` mit dem `LoginViewController` und teilt ihm so mit, dass es sich verändert hat. Das obige Beispiel lässt sich für beliebige Anwendungsfälle adaptieren und implementieren.

5.3.5 ListViewModels und CellViewModels

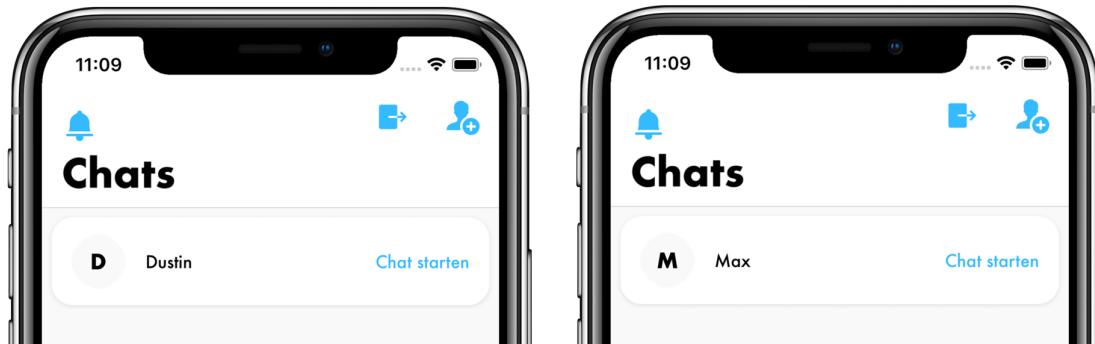


Abbildung 5.11: Chatlistenansicht in M2

Um das MVVM-Entwurfsmuster auf eine Liste von Objekten beziehen zu können, erfordert es eine Aufteilung zwischen Liste und Listeneintrag. In der obigen Abbildung 5.11 ist die Chatlistenansicht in M2 zu sehen. Diese wird mit dem ChatsViewController dargestellt. Im folgenden Beispiel wird die Struktur hinter dem Entwurf aus Abbildung 5.10 auf die Chatlistenansicht unter Nutzung eines ListViewModels angewandt.

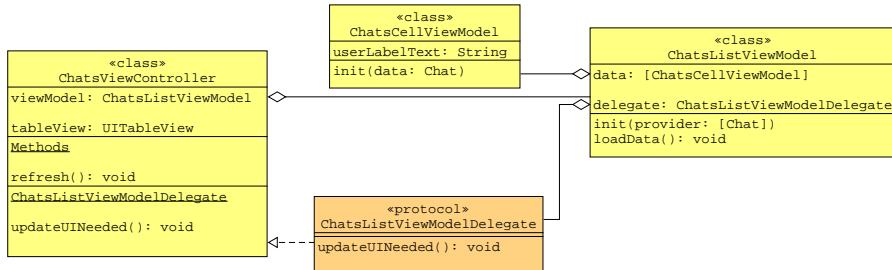


Abbildung 5.12: Klassendiagramm für einen Entwurf von ChatsViewController, ChatsListViewModel und ChatsCellViewModel

Der Aufbau zur Kommunikation zwischen ChatsViewController und ChatsListViewModel ist ähnlich wie zwischen LoginViewController und LoginViewModel, mit dem Unterschied, dass das ChatsListViewModel wiederum eine Liste von ChatsCellViewModels enthält, welche die Darstellung eines einzelnen Listeneintrags repräsentieren.

5.3.6 Worker

Worker-Klassen dienen zur Auslagerung von vereinzelten Verarbeitungsmechanismen. Genutzte Klassen externer Bibliotheken werden verkapselt, der Inhalt des Workers ist austauschbar. Muss aus irgendeinem Grund auf die Nutzung von einer Bibliothek verzichtet werden, würde dies ermöglicht werden, ohne alle Stellen dessen Nutzung im Code aktualisieren zu müssen, da dort nicht direkt die Bibliothek, sondern der BackendWorker eingesetzt wird. Dies ist ein Entwurfsmuster und bekannt als **Facade Pattern** (Eilebrecht & Starke 2019: 87 f.). Folgend werden alle Worker-Klassen in M2 aufgelistet:

Liste der Worker-Klassen in M2	
Name	Beschreibung
UsersWorker	Aufgaben rund um die Nutzerverwaltung
BackendWorker	Aufgaben rund um die Kommunikation zum HTTP-Teil des Backends
StreamWorker	Aufgaben rund um die Kommunikation zum Stream-Teil des Backends
PersistanceWorker	Aufgaben rund um die Speicherung von Werten
AudioRecordingWorker	Aufnahme und Speicherung von Audio
AudioPlayingWorker	Wiedergabe von Audio
SpeechRecognitionWorker	Spracherkennung von Audiodateien

Tabelle 5.2: Liste der Worker-Klassen in M2

In der folgenden Abbildung 5.13 findet sich ein Entwurf des **AudioRecordingWorkers** zur Aufnahme und Speicherung von Audio.

5 Softwarearchitektur

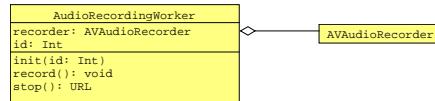


Abbildung 5.13: Klassendiagramm für einen Entwurf vom AudioRecordingWorker

5.3.7 Provider

In Abschnitt 5.3.5 wurde auf den Umgang mit Listenansichten und ViewModels in Form von ListViewModels eingegangen. Die Darstellung in dem aufgeführten Entwurf 5.12 ist jedoch vereinfacht: Das ListViewModel wird direkt mit einer Liste des Models `Chat` initialisiert.

Da diese Objektliste im Fall von M2 mittels BackendWorker aus dem Backend geholt wird, muss dies angestoßen und aufbereitet werden. Dies übernehmen sogenannte **Provider**-Klassen. Diese sind im Kontext der Entwicklung von WPF-Anwendungen auch als **Dienste** bekannt (Kühnel 2012). Sie beinhalten eine Liste der Model-Objekte und Hilfsfunktionen rund um die Listenoperationen. Sie sind außerdem **Observable** und benachrichtigen ihre **Observer** bei Veränderung der enthaltenen Model-Liste. Die ViewModels für Listenansichten werden somit in M2 nicht direkt aus einer Liste von Models, sondern mit einem **Provider**, welcher die Models aufbereitet initialisiert.

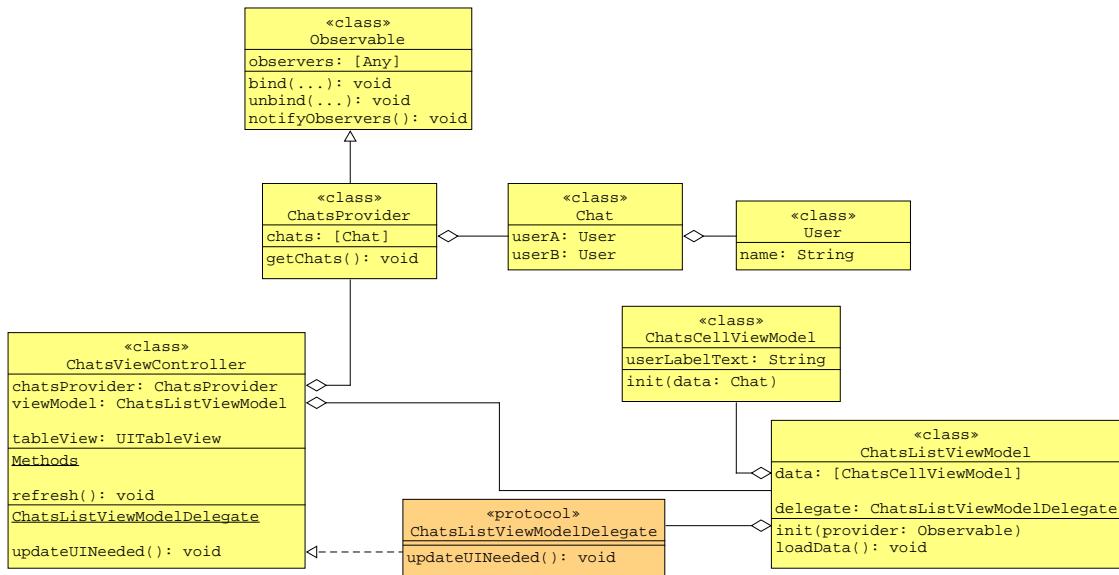


Abbildung 5.14: Klassendiagramm für einen Entwurf von ChatsViewController, ChatsListViewModel und ChatsCellViewModel inklusive ChatsProvider

6 Entwicklung

6.1 Entwicklungsumgebung

Die Entwicklung des gesamten Projektes erfolgt sowohl auf einem Apple iMac 27" als auch auf einem Apple MacBook Pro 13". Beide Geräte stammen aus dem Jahre 2018 und laufen unter dem Betriebssystem macOS Mojave 10.14. Getestet wird die iOS-App auf Apples iOS-Simulator und einem Apple iPhone X.

Für die Programmierung der iOS-App wird Apples Entwicklungsumgebung **Xcode** 10 verwendet ([Apple 2019](#)). Sie ermöglicht u.a. die Entwicklung des Codes, Debugging, Gestaltung der Benutzeroberfläche mit dem **Interface Builder** und die Verwaltung von eingebundenen Bildern in der App mit einem **Asset-Katalog**. Mit **Xcode** können sowohl Apps für die Apple-Plattformen macOS, iOS, tvOS und watchOS als auch C oder C++ Anwendungen entwickelt werden.

Zur Programmierung des Backends wird **IntelliJ IDEA Ultimate 2018.3** von JetBrains mit Plug-Ins für die Nutzung der Programmiersprachen PHP und Go eingesetzt ([JetBrains 2019](#)).

Zur Überprüfung der fertig entwickelten HTTP-Endpunkte wird die Software **Paw** genutzt. Paw ist in der Lage, jegliche HTTP-Requests inklusive ihrer Parameter abzusenden und zu testen ([Paw 2019](#)).

Zur Erstellung der Grafiken und der Konzeption der Benutzeroberfläche wurde das Programm **Sketch** verwendet ([Sketch 2019](#)).

Zur Erstellung der Sequenz- und Klassendiagramme wurde das Tool **UMLet** verwendet ([UMLet 2019](#)).

Bildschirmfotos der verwendeten Entwicklungsumgebungen finden sich im Anhang (Abbildungen A.1, A.2, A.3, A.4 und A.5).

6.2 Swift

M2 wird vollständig in Apples Programmiersprache **Swift** 5 entwickelt. **Swift** wurde im Jahre 2014 von Apple als Nachfolger der Programmiersprache **Objective C** vorgestellt ([Sillmann 2019: 4 f.](#)). **Swift** ist im Gegensatz zu seinem Vorgänger Open

6 Entwicklung

Source¹ und kann zudem sowohl unter macOS als auch unter Linux verwendet werden ([Sillmann 2019: 4 f.](#)).

Zur Programmierung von nativen Apps für iOS, tvOS, macOS oder watchOS kann entweder **Swift** oder sein Vorgänger **Objective C** verwendet werden. Die Wahl der Programmiersprache steht hierbei dem Entwickler frei. Apple selbst spricht keine explizite Empfehlung zur Nutzung von **Swift** aus, allerdings lassen das Marketing und die von Apple aufgewendeten Ressourcen zur Weiterentwicklung von Swift darauf schließen, dass dessen Nutzung in iOS-Apps zukunftsorientierter ist. Nach der Aussage des Chefarchitekten von **Swift** ([Lattner 2019](#)), ist **Swift** u.a. von den Programmiersprachen **Objective-C**, **Rust**, **Haskell**, **Ruby**, **Python**, **C#** und **CLU** beeinflusst. Das Ziel bei der Entwicklung von **Swift** war es, eine Programmiersprache zu schaffen, die zwar mächtig in ihren Mitteln ist, allerdings trotzdem einen unkomplizierten Einstieg ermöglicht und Spaß in ihrer Nutzung macht ([Apple 2019](#)).

Swift ist multiparadigmatisch und erlaubt somit u.a. objektorientierte sowie funktionale Programmierung.

Zu den wichtigsten Konzepten in **Swift** zählen laut [Apple \(2019\)](#):

- Optionals zur Vermeidung von **NullPointerExceptions**.
- Durchdachte Syntax für einen schnellen und unkomplizierten Kontrollfluss in der Programmierung.
- Tupel, welche z.B. mehr als nur einen Rückgabewert einer Funktion erlauben.
- Structs² mit Funktionen, Extensions³ und Protokollen.
- Closures⁴ mit Funktionszeigern.

Seit Anfang 2014 wurden diverse Aktualisierungen veröffentlicht. Die aktuelle Version 5, in der auch **M2** entwickelt wird, wurde Anfang 2019 veröffentlicht ([Sillmann 2019: 5](#)).

6.3 Bibliotheken

Bei der Entwicklung von **M2** werden mehrere Bibliotheken genutzt, um die Programmierung einiger Aufgabenbereiche zu erleichtern.

¹Quellcode ist öffentlich zugänglich

²Verbundsdatentypen

³Erweiterungen

⁴Funktionsabschlüsse

6.3.1 Apple

UIKit und Foundation

UIKit ist zusammen mit Foundation Teil des von Apple entwickelten Cocoa Touch Frameworks ([Apple 2018](#)).

Beide Bibliotheken stellen eine grundlegende Sammlung von Klassen und Strukturen zur Entwicklung von Apps auf Apples Plattformen dar.

UIKit beinhaltet Bedienelemente wie Textfelder, Knöpfe oder Regler und ist aktuell ausschließlich auf den Plattformen iOS, tvOS und watchOS verfügbar ([Apple 2019](#)).

Foundation beinhaltet essentielle Datentypen wie String, Integer oder Systemdienste ([Apple 2019](#)).

Sowohl UIKit als auch Foundation kommen in M2 zum Einsatz.

Im Zuge der diesjährigen WWDC⁵ 2019 wurde eine Vorabversion eines neuen Frameworks von Apple namens Mac Catalyst vorgestellt ([Apple 2019](#)), welches die Möglichkeit bietet, für iOS entwickelte Apps auch unter der Plattform macOS ausführen zu können. Mit dieser Ankündigung wurde ebenfalls die Bibliothek UIKit in einer Vorabversion für die Kompatibilität mit macOS aktualisiert. Auf die Nutzung von Mac Catalyst wird in M2 allerdings verzichtet, da es sich hierbei noch um eine Vorabversion handelt.

Weitere Bibliotheken

Zur Aufnahme und Wiedergabe der Sprachnachrichten in M2 wird die Bibliothek AVFoundation genutzt ([Apple 2019](#)).

Zur Spracherkennung für den Inhalt der Sprachnachrichten als Text wird die Bibliothek Speech genutzt ([Apple 2019](#)).

6.3.2 CocoaPods

Es gibt außerhalb der Bibliotheken von Apple einige Möglichkeiten zur Einbindung von Bibliotheken Dritter. Eine Variante ist CocoaPods ([CocoaPods 2019](#)), ein Paketverwaltungs-Tool für Bibliotheken Dritter in den Programmiersprachen Objective-C und Swift. Es läuft in der Kommandozeile auf macOS und wurde in der Programmiersprache Ruby entwickelt.

Entwickler können ihre entwickelten Bibliotheken als pod zur Verfügung stellen, dieser wird über ein öffentliches Git⁶-Repository bereitgestellt.

Zur Einbindung von Bibliotheken mit CocoaPods in das eigene Projekt wird die gewünschte Bibliothek in einer Podfile eingetragen und mit der Kommandozeilenan-

⁵World Wide Developer Conference: Entwicklerkonferenz von Apple

⁶Software zur Versionierung von Quellcode

wendung ein Befehl zur Installation ausgeführt. Diese erzeugt eine Xcode-Workspace⁷ mit zwei Unterprojekten. Das eine Unterprojekt beinhaltet die eigene App und das andere Unterprojekt die Bibliotheken. Die Bibliotheken können dann direkt in Xcode verwendet werden. Über die Kommandozeilenanwendung können sie außerdem aktualisiert oder wieder deinstalliert werden. **CocoaPods** wird in M2 genutzt, um Bibliotheken Dritter einzubinden.

Vorteile

Natürlich können Bibliotheken Dritter auch manuell heruntergeladen und dem Xcode-Projekt hinzugefügt werden. Dadurch lassen sich die Bibliotheken allerdings deutlich schlechter versionieren und aktualisieren. Außerdem kann **CocoaPods** auch mit einem privaten Git-Repository genutzt werden. Gleiche Vorkommnisse von Code in verschiedenen Projekten können so als Bibliothek zusammengefasst und innerhalb einer Firma verteilt werden.

Nachteile

Die Bibliotheken in **CocoaPods** sind nicht vorkompiliert. Ist die Kompilierung einer Bibliothek also nicht aus vorangegangenen Builds im Cache vorhanden, muss diese Bibliothek komplett neu kompiliert werden. Dies kann bei großen Bibliotheken viel Zeit in Anspruch nehmen. Außerdem kann die Erstellung und Wartung eigener Bibliotheken durch **CocoaPods** sehr zeitaufwändig und kompliziert sein.

Zur Vermeidung des Kompilierungsproblems gibt es bspw. **Carthage**, ein Paketverwaltungs-Tool für Bibliotheken Dritter, ähnlich wie **CocoaPods**, nur auf Basis *vorkompilierter* Bibliotheken. Die Handhabung ist allerdings komplizierter und einige Bibliotheken haben mit **Carthage** Probleme oder sind dafür nicht verfügbar. Deshalb wird **Carthage** in M2 nicht eingesetzt.

6.3.3 Alamofire

Alamofire ist eine Bibliothek zur Vereinfachung von Prozessen in der Netzwerkverwaltung unter Apples Plattformen iOS, macOS, tvOS und watchOS. Da die von Apple zu diesem Zwecke zur Verfügung gestellten Klassen⁸ in ihrer Syntax teils sehr umständlich sind, wurde **Alamofire** kurz nach der Veröffentlichung von Swift von dem Entwickler Mark Thompson veröffentlicht ([Alamofire 2019](#)). **Alamofire** bietet u.a. übersichtlichere Syntax, direktes Parsing aus JSON⁹ und diverse durchdachte Hilfsfunktionen für gängige Netzwerkfunktionen wie Download oder Upload.

Alamofire wird in M2 für die Umsetzung aller Netzwerkfunktionalitäten außerhalb

⁷Sammlung von Xcode-Projekten

⁸NSURLSession ([Apple 2019](#))

⁹JavaScript Object Notation

des Streams (Abschnitt 6.4.2) genutzt. Eine Liste aller über `Alamofire` angesprochenen Netzwerkendpunkte findet sich in Abschnitt 6.4.1.

6.3.4 Firebase

`Firebase` wurde laut [Wikipedia \(2019\)](#) im Jahr 2011 von James Tamplin und Andrew Lee gegründet und 2014 von Google aufgekauft. `Firebase` ist eine Entwicklungs-Plattform mit einer Sammlung diverser, cloudbasierter Dienste, welche über ein SDK u.a. für iOS zur Verfügung gestellt werden. `Cloud Firestore` ist ein Teil von `Firebase` und erlaubt die cloudbasierte Nutzung einer Datenbank und dessen Implementierung in einer mobilen Anwendung zusammen mit durchdachten Schnittstellen, welche sich sowohl um die Netzwerkkommunikation als auch die Persistenz und die Verwaltung der Daten kümmern.

`Cloud Firestore` wurde für die Nutzung in M2 evaluiert und sollte ursprünglich als Datenbank für die Nutzer, Chats und Nachrichten fungieren. Letztlich wurde sich aus folgenden Gründen gegen die Nutzung entschieden:

- `Cloud Firestore` erlaubt im kostenlosen Angebot 20.000 Schreibzugriffe und 50.000 Lesezugriffe am Tag ([Google 2019](#)). Dies ist für die Entwicklung ausreichend, allerdings nicht zukunftsorientiert. Die Zugriffe können bei einer Verwendung von M2 durch mehrere Nutzer schnell aufgebraucht sein, insbesondere unter Anbetracht der Tatsache, dass das Eintippen eines einzelnen Buchstabens bereits sowohl einen Lese- als auch einen Schreibzugriff erzeugen würde.
- `Cloud Firestore` erzwingt zur Nutzung die gesamte Einbindung von `Firebase` und bringt somit viele Klassen zusätzlich mit, die gar nicht für den speziellen Anwendungsfall notwendig gewesen wären.
- Eine eigens erzeugte Veränderung der Bibliothek bei einem spezialisierten Problem ist kompliziert und könnte bei einer Aktualisierung vom Herausgeber überschrieben werden. Dies ist ein gängiges Problem bei der Nutzung von Bibliotheken Dritter. Mit der Datenverwaltung der Nutzer, Chats und Nachrichten durch `Firebase` wäre das Herzstück von M2 somit verkapselt.
- Durch die Nutzung entstünde eine starke Bindung an die Plattform `Firebase` und eine Veränderung dessen Konditionen oder Verfügbarkeit würde somit über die gesamte Zukunft von M2 entscheiden.

6.3.5 Weitere Bibliotheken

TinyConstraints

`TinyConstraints` ist eine Bibliothek zur Vereinfachung der programmatischen Steuerung von Constraints unter iOS. Constraints werden vergeben, um eine Darstellung

6 Entwicklung

eines **ViewControllers** für mehrere Bildschirmgrößen anpassbar zu machen. Constraints werden normalerweise direkt im **Interface Builder** in **Xcode** gesetzt, in einigen Fällen ist dies allerdings nicht ausreichend. Die von Apple zur Verfügung gestellte Schnittstelle ist sehr unleserlich und schwierig zu handhaben. **TinyConstraints** vereinfacht die Ansprache dieser und wird deshalb in **M2** dafür eingesetzt.

XCGLogger

XCGLogger ist eine Bibliothek zum verbesserten Logging unter iOS. Standardmäßig bietet Apple keine andere Möglichkeit als ein vereinfachtes „Drucken“ von Text in die Ausgabe von **Xcode**. Es existieren keine mehrfachen Einstufungen für die Priorität des ausgedruckten Inhaltes wie **Debug**, **Warning** und **Error**. Zudem gibt es keinen Steuermechanismus, der per einfacher Umprogrammierung das gesamte Logging deaktiviert, was für Produktionsumgebungen aus Sicherheitsgründen wichtig ist. All diese Mechanismen sind Teil von **XCGLogger** und werden deshalb in **M2** implementiert.

AppCenter

AppCenter ist eine Verteilungsplattform von Microsoft für iOS- und Android-Apps außerhalb der App Stores. Um die App zu Testzwecken anderen Nutzern zur Verfügung stellen zu können, wird die Implementierung einer dazugehörigen Bibliothek vorausgesetzt. Diese wurde in **M2** ebenfalls implementiert, um den Verteilungsprozess zu vereinfachen.

DifferenceKit

DifferenceKit ist eine Bibliothek zur vereinfachten animierten Aktualisierung der Datensätze von Listenansichten unter iOS. Die standardmäßige Schnittstelle von Apple zur animierten Aktualisierung einer Listenansicht ist sehr fehleranfällig und umständlich. **DifferenceKit** behebt dieses Problem und wird in **M2** in der Chatliste und der Freundschaftsanfragenliste eingesetzt.

SwiftDate

SwiftDate bietet eine große Sammlung von Hilfsmethoden zur Darstellung und Verarbeitung von datumsbasierten Datentypen unter iOS. **SwiftDate** wird in **M2** zur Formatierung von datumsbasierten Attributen verschiedener Objekte eingesetzt.

EmptyDataSet-Swift

EmptyDataSet-Swift bietet eine schnelle und einfache Möglichkeit zur Darstellung von Platzhaltern in Listenansichten unter iOS. Apple selber stellt hierzu keine Schnittstelle zur Verfügung und die eigene Entwicklung einer Schnittstelle hierfür kann sehr

zeitaufwändig und fehleranfällig werden, daher wird `EmptyDataSet-Swift` in M2 zur Darstellung von Platzhaltern in allen Listenansichten eingesetzt.

NGSBadgeBarButton

`NGSBadgeBarButton` erlaubt die Darstellung eines sogenannten **Badges** in einer Navigationsleiste unter iOS. Ein **Badge** ist eine Zahl mit kreisrundem, farblich hinterlegtem Hintergrund, welcher unter iOS oft zur Darstellung von offenen Anfragen zum Einsatz kommt. Bekannt ist eine Implementierung hauptsächlich direkt am App-Icon selbst, beispielsweise in Apples Mail-App für die Anzahl der ungelesenen Mails. In M2 wird mit einem **Badge** unter der Nutzung von `NGSBadgeBarButton` in der Navigationsleiste die Anzahl der noch offenen Freundschaftsanfragen direkt neben dem Knopf zum Öffnen der Freundschaftsanfragenliste dargestellt.

6.4 Backend

Um einen Großteil der Funktionalitäten (Abschnitt 3.4) von M2 gewährleisten zu können, muss zusätzlich zur App auch ein Backend entwickelt werden. Es bietet die Basis der Kommunikation zwischen mehreren Nutzern der App und beinhaltet eine Nutzerdatenbank. Das Backend ist nicht Hauptbestandteil dieser Arbeit und sollte deshalb ursprünglich durch `Firebase` realisiert werden, welches aus bereits genannten Gründen (Abschnitt 6.3.4) aber nicht geschieht. Deshalb wird das Backend eigenständig entwickelt. Bei der Entwicklung werden vor dem Hintergrund der Tatsache, dass der Fokus dieser Arbeit sich auf die Konzeption und Implementierung der iOS-App konzentriert, keinerlei Aspekte im Bereich Sicherheit beachtet. Das Backend ist im Entwicklungsstand dieser Arbeit nicht auf die Nutzung in einer Produktionsumgebung ausgelegt und verschickt seine Informationen bewusst unverschlüsselt.

Das Backend besteht aus zwei Teilen: `HTTP` und `Stream`.

6.4.1 HTTP

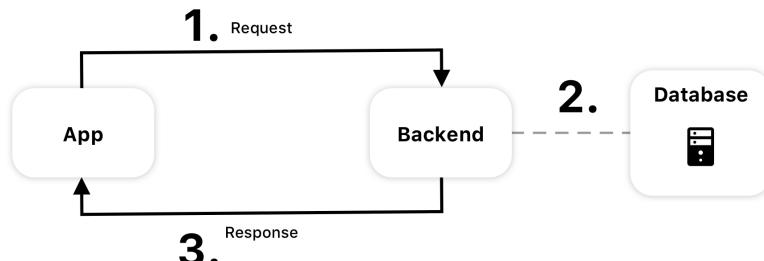


Abbildung 6.1: Visualisierung einer beispielhaften HTTP-Anfrage der App zum Backend

6 Entwicklung

Zur Steuerung datenbankbezogener Informationen in M2 bietet das Backend HTTP-Endpunkte. Diese werden in der Programmiersprache PHP entwickelt. Sie kommunizieren mit einer SQL-Datenbank.

In der Abbildung 6.1 ist der Ablauf einer beispielhaften HTTP-Anfrage an das Backend visualisiert. Zur einheitlichen Benennung werden teils englische Begriffe verwendet.

1. Die App stellt einen Request an das Backend über eine vorgegebene Route.
2. Das Backend durchsucht die Datenbank nach den erfragten Informationen und bereitet diese auf.
3. Das Backend liefert eine Response mit den aufbereiteten Daten.
4. Die App parsed die Response in die eigene Datenstruktur und verwendet diese zur Darstellung der Informationen.

In M2 werden HTTP-Requests für folgende Anwendungsfälle eingesetzt:

- Login, Registrierung, Logout und Löschen eines Nutzers
- Aktualisierung eines Gerätetokens für Push-Benachrichtigungen
- Validierung eines Nutzernamens
- Senden, Abfragen, Beantworten und Löschen von Freundschaftsanfragen
- Abfragen von Chats
- Setzen und Abfragen des aktiven Chats
- Upload, Download, Abfragen und Löschen von Sprachnachrichten

Mit einer Response liefert das Backend bei einem Request auch einen HTTP-Status-Code. Falls es sich um einen Fehlercode handelt, kann dieser von der App interpretiert und als Fehlermeldung angezeigt werden. Hierzu wurden sowohl standardisierte Status-Codes für allgemeinere Meldungen als auch eigene Status-Codes für speziellere Meldungen genutzt.

Eine genaue Spezifizierung der HTTP-Endpunkte und Status-Codes ist dem Anhang zu entnehmen (Tabellen A.1, A.2, A.3, A.4 und A.5).

6.4.2 Stream

Zusätzlich zu den HTTP-Endpunkten wird ein Stream als zweiter Teil des Backends entwickelt. Dieser dient zur Übermittlung von Daten in Echtzeit. Der Stream nutzt das Übertragungsprotokoll TCP¹⁰ und wird in der Programmiersprache Go entwickelt.

Der Stream bietet folgende Funktionalitäten:

- Übermittlung einer Textnachricht während sie getippt wird
- Versenden einer fertig eingetippten Textnachricht
- Versenden einer Sprachnachricht (Audiodatei separat über HTTP-Endpunkt)
- Benachrichtigung bei Veränderung des aktiven Chats eines anderen Nutzers
- Benachrichtigung bei Veränderung der Liste der Freundschaftsanfragen
- Versenden von Push-Benachrichtigungen für Chat- oder Freundschaftsanfragen, wenn ein Nutzer nicht aktiv im Stream ist

Eine genaue Spezifizierung der Stream-Befehle ist dem Anhang zu entnehmen (Tabelle A.7).

6.4.3 JSON

Jegliche HTTP-Requests des Backends senden als Antwort einen JSON-Text. JSON ist ein Datenformat zur Standardisierung des Datenaustauschs zweier Anwendungen ([Wikipedia 2019](#)). Es wird genutzt, um Datenstrukturen, die in unterschiedlichen Anwendungen aufgrund unterschiedlicher Plattformen andere Strukturen besitzen auf abstrakter Ebene miteinander kommunizieren lassen zu können. Ein passendes Beispiel hierfür ist die Kommunikation zwischen der in **Swift** programmierten App und dem in **PHP** programmierten HTTP-Teil des Backends.

Im Folgenden wurde das in Abschnitt 5.2.2 verwendete Beispiel noch einmal aufgegriffen, um das Objekt Stuhl als JSON darzustellen.

```

1 {
2   "name": "Stuhl",
3   "legCount": 2
4 }
```

Quellcode 6.1: Beispielhaftes JSON-Objekt Stuhl

¹⁰Transmission Control Protocol

Ein Objekt beginnt und endet in JSON stets mit gewellten Klammern. Innerhalb eines Objektes werden Werte mittels „Key-Value“-Prinzip dargestellt. Ein Attribut wird durch einen Key definiert (im Quellcode 6.1 z.B. „name“) und dessen hinterlegter Wert mit einem Value dargestellt (im Quellcode 6.1 z.B. „Stuhl“). Die Formatierung des Values entspricht dem enthaltenen Datentypen. JSON erlaubt die Formatierung gängiger Datentypen wie String, Integer oder Boolean. Mehrere Werte werden durch ein Komma getrennt. Eine Liste von Objekten wird mittels eckiger Klammern geöffnet und geschlossen.

```
1 [  
2 {  
3   "name": "Stuhl",  
4   "legCount": 2  
5 },  
6 {  
7   "name": "Zweiter Stuhl",  
8   "legCount": 4  
9 }  
10 ]
```

Quellcode 6.2: Liste von JSON-Objekten

JSON wird in M2 genutzt, um Objekte in der Kommunikation zwischen App und Backend abstrahiert darzustellen.

6.4.4 SQL

SQL ist eine der weltweit am meisten genutzten Datenbanksprachen für relationale Datenbanken. Sie wurde ursprünglich von IBM in den 1970er Jahren entwickelt und basiert auf dem Modell relationaler Datenbanken von E. F. Codd. SQL unterstützt die Abfrage, Manipulation und Administration von Datenbankinhalten in tabellarischer Form ([Chamberlin 2017](#)).

Folgend wird anhand eines kurzen Beispiels die Abfrage von Objekten erläutert.

Als Beispiel dient erneut der Raumplaner, unter Annahme es gäbe eine Datenbank zur Speicherung der enthaltenen Objekte. In der obigen Abbildung 6.2 wird eine Datenbank mit einer Tabelle `furniture` inklusive zweier Einträge dargestellt.

```
1 SELECT * FROM furniture;
```

Quellcode 6.3: Beispielhafter SQL-Befehl zur Abfrage der gesamten Tabelle

Database		
Table furniture		
id	name	leg_count
1	Stuhl	2
2	Zweiter Stuhl	4

Abbildung 6.2: Beispielhafte Tabelle für Möbel im Raumplaner

Im Codebeispiel 6.3 wird der gesamte Inhalt der Tabelle `furniture` abgefragt. Das Symbol * signalisiert die Auswahl aller Spalten in der Tabelle. Das Resultat der Abfrage entspricht also dem Inhalt der Datenbank samt aller Spalten, also der Abbildung 6.2.

```
1 SELECT name FROM furniture WHERE leg_count = 2;
```

Quellcode 6.4: SQL-Befehl zur Abfrage einer einzelnen Spalte mit einer Bedingung

Im obigen Codebeispiel 6.4 wird ausschließlich der Inhalt der Spalte `name` der Tabelle `furniture` für alle Einträge mit dem Wert 2 in der Spalte `leg_count` abgefragt. Das Resultat der Abfrage wäre also eine einzelne Spalte `name` mit dem Inhalt „Stuhl“.

Im Backend von M2 wird eine SQL-Datenbank zur Speicherung der Nutzerinformationen, Freundschaften und Archivierung der Sprachnachrichten genutzt. Eine vollständige Übersicht der Tabellenstruktur der Datenbank ist im Anhang einzusehen (Abbildung A.6).

6.5 Zusammenfassung

Das Projekt M2 besteht aus einer iOS-App und einem Backend, wobei der Fokus dieser Arbeit sich auf die Konzeption und Implementierung der App konzentriert.

Für die Programmierung der App wird Apples Programmiersprache **Swift 5** genutzt. Des Weiteren werden diverse Bibliotheken verwendet, eine vollständige Liste findet sich im Anhang (Tabelle A.8).

Das Backend besitzt sowohl HTTP-Endpunkte, welche der App aufbereitete Antworten mit Informationen aus einer SQL-Datenbank bereitstellen als auch einen Stream für einen Datenaustausch mehrerer App-Nutzer in Echtzeit. Als Abstraktion aller Objekte, die zwischen App und Backend aufbereitet und verschickt werden, wird JSON verwendet. Die HTTP-Endpunkte werden in PHP programmiert, die Programmierung des Streams erfolgt in Go.

7 Implementierung

In Abschnitt 3.4 wurden alle Anforderungen von M2 analysiert. Inhalt dieses Kapitels ist die Erläuterung der Implementierungen dieser Anforderungen. Hierzu werden, wenn nötig, Klassen- und Sequenzdiagramme und Codebeispiele zur besseren Veranschaulichung eingesetzt. Bei deckungsgleichen Implementierungen ähnlicher Funktionalitäten wird darauf allerdings verzichtet.

7.1 Kommunikation zwischen App und Backend

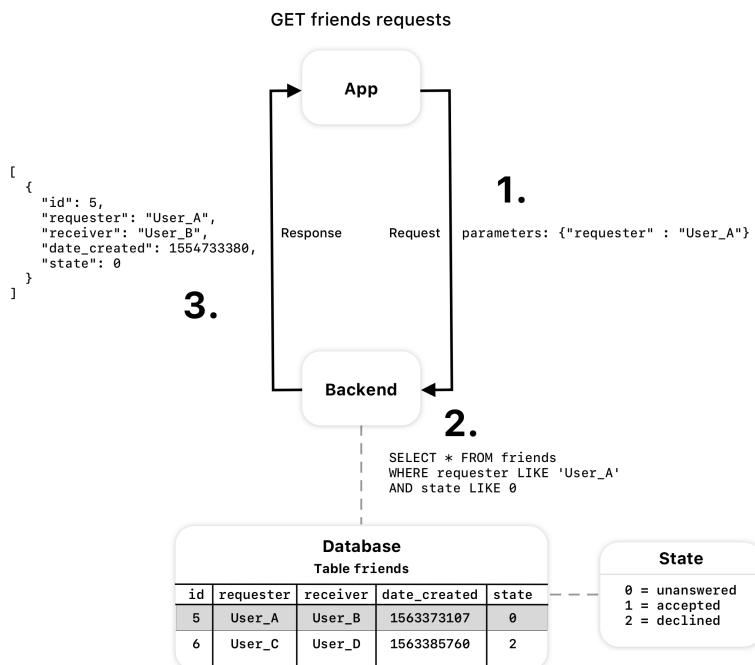


Abbildung 7.1: Visualisierung des gesamten Ablaufs einer HTTP-Anfrage von der App zum Backend

Fast jede Funktionalität in der iOS-App besitzt einen dazugehörigen HTTP-Request im Backend. Sowohl der Stream-Teil als auch der HTTP-Teil des Backends werden auf einem Cloudserver des Anbieters Hetzner¹ gehostet, damit das Backend von überall aus erreichbar ist. Bevor auf die Implementierung der Funktionalitäten in der App

¹<https://www.hetzner.de/cloud>, letzter Zugriff: 19.08.2019

eingegangen werden kann, muss der Ablauf der Kommunikation zwischen App und Backend genauer erläutert werden. In Abbildung 7.1 ist der vollständige Ablauf einer HTTP-Anfrage von der App zum Backend inklusive der Antwort vom Backend zurück zur App visualisiert. In diesem Fall handelt es sich um die Abfrage der offenen Freundschaftsanfragen für einen Nutzer.

1. Die App startet einen GET-Request mit einer passenden Route. Damit das Backend weiß, für welchen Nutzer es offene Anfragen heraussuchen soll, muss die App dies dem Backend übermitteln. Hierzu wird an den HTTP-Endpunkt ein Parameter übergeben. Nicht alle Endpunkte erfordern die Übergabe eines Parameters. Den Tabellen des Anhangs A.1, A.2, A.3, A.4 und A.5 lassen sich die Spezifizierungen entnehmen.
2. Das Backend erhält den GET-Request der App und verarbeitet die Anfrage. Alle Freundschaften sind in der Tabelle `friends` gesammelt. Offene Freundschaftsanfragen haben alle den Eintrag „0“ in der Spalte `state`. Nachdem bspw. eine Freundschaft mit der Nutzung eines anderen HTTP-Requests akzeptiert wurde, verändert sich diese Spalte von „0“ auf „1“. Das Backend durchsucht die Datenbank nach einem passenden Eintrag. Wird ein Eintrag gefunden, geht es weiter zu Schritt 3. Wird kein Eintrag gefunden, wird ein passender Status-Code im HTTP-Header ausgegeben. Eine Liste aller Status-Codes lässt sich im Anhang finden (Tabelle A.5).
3. Das Backend bereitet einen gefundenen Eintrag als JSON-Text auf und sendet diesen als Antwort zurück zur App. Diese parsed die Antwort dann in die eigene Datenstruktur und verwendet diese zur Darstellung.

Dieser Ablauf wiederholt sich für den Aufruf aller HTTP-Endpunkte zwischen App und Backend, eine Visualisierung jedes individuellen Endpunktes ist daher nicht notwendig.

7.2 Hierarchie der ViewController

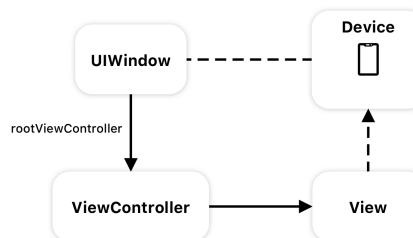


Abbildung 7.2: Hierarchie der Darstellung eines ViewControllers unter iOS

7 Implementierung

Den Ursprung aller ViewController bildet das `UIWindow`. Dieses besitzt einen Haupt-ViewController, den `rootViewController` (Abb. 7.2). Die Wahl des `rootViewController`s entscheidet über den ViewController, der beim App-Start angezeigt wird (Apple 2018).

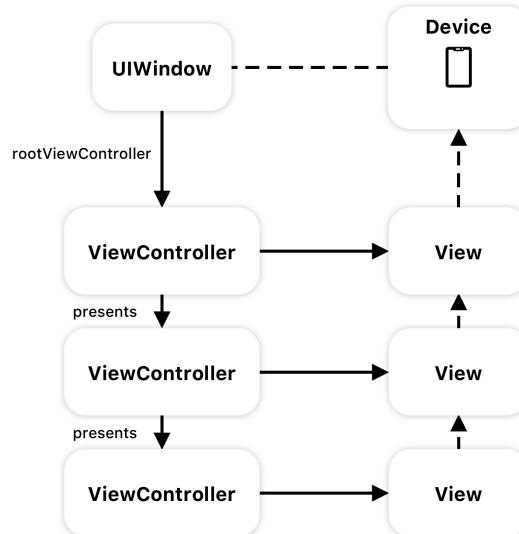


Abbildung 7.3: Hierarchie von mehreren, auf dem `rootViewController` präsentierten ViewControllern

Auf dem `rootViewController` können dann weitere ViewController präsentiert werden (Abb. 7.3).

In der folgenden Abbildung 7.4 ist die Hierarchie aller ViewController in M2 dargestellt.

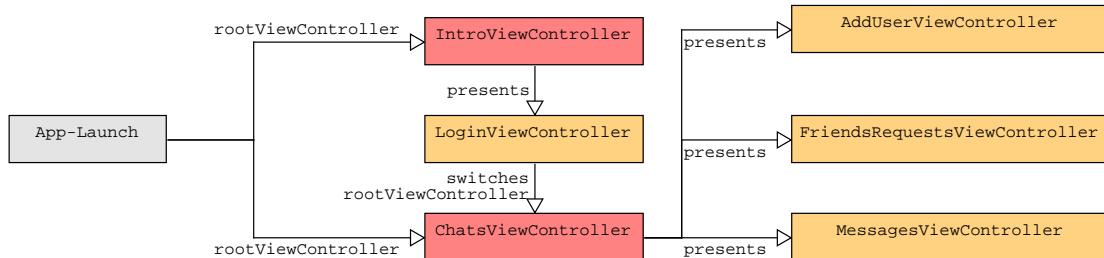


Abbildung 7.4: Hierarchie aller ViewControlller in M2

7.3 Login

Beim App-Start überprüft die App, ob ein Nutzer bereits angemeldet ist. Ist kein Nutzer angemeldet, wird der **IntroViewController** als **rootViewController** gesetzt. Von diesem aus kann der Nutzer sich anmelden oder registrieren.

Da sich der Aufbau der Ansicht für die zwei verschiedenen Anwendungsfälle nicht unterscheidet, sondern lediglich die Texte der UI-Elemente, wird der **LoginViewController** für beide genutzt, aber mit zwei verschiedenen ViewModels initialisiert. Im Sequenzdiagramm 7.5 ist der Ablauf visualisiert.

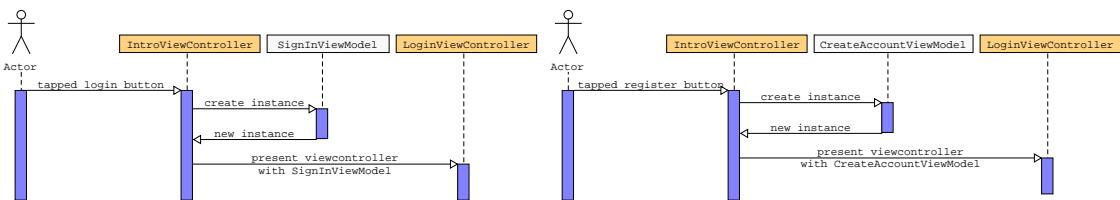


Abbildung 7.5: Sequenzdiagramm für Anmeldung (l) oder Registrierung (r) in M2

Für den Nutzer äußert sich die Interaktion dann wie in der folgenden Abbildung 7.6.

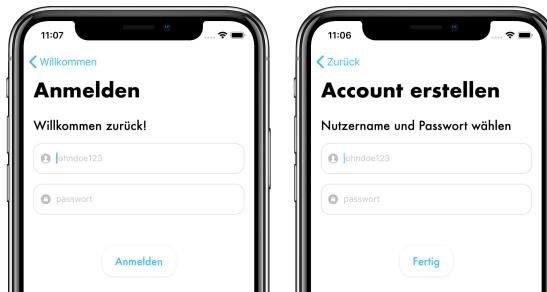


Abbildung 7.6: LoginViewController, initialisiert mit SignInIntroViewModel (l) oder CreateAccountIntroViewModel (r)

Nach Eingabe von Nutzernamen und Passwort folgt beim Tippen auf den Anmelden- oder Registrieren-Knopf ein HTTP-Request. Für die Anmeldung wird mit dem **UsersWorker** im Backend überprüft, ob die Daten korrekt sind. Ist die Antwort vom Backend positiv, wechselt der **LoginViewController** den **rootViewController** auf den **ChatsViewController**. Registriert sich der Nutzer, wird mit dem **UsersWorker** erst im Backend ein Account erstellt und dann ein Login durchgeführt. Die App speichert die Anmelde- und Registrierte-Informationen mit dem **PersistenceWorker** und der Nutzer braucht sich bei einem erneuten App-Start nicht wieder anzumelden. Solange der Nutzer angemeldet bleibt, startet die App direkt mit dem **ChatsViewController** (Abb. 7.4). Im folgenden Sequenzdiagramm 7.7 wird der Ablauf einer Anmeldung im Erfolgsfall dargestellt.

7 Implementierung

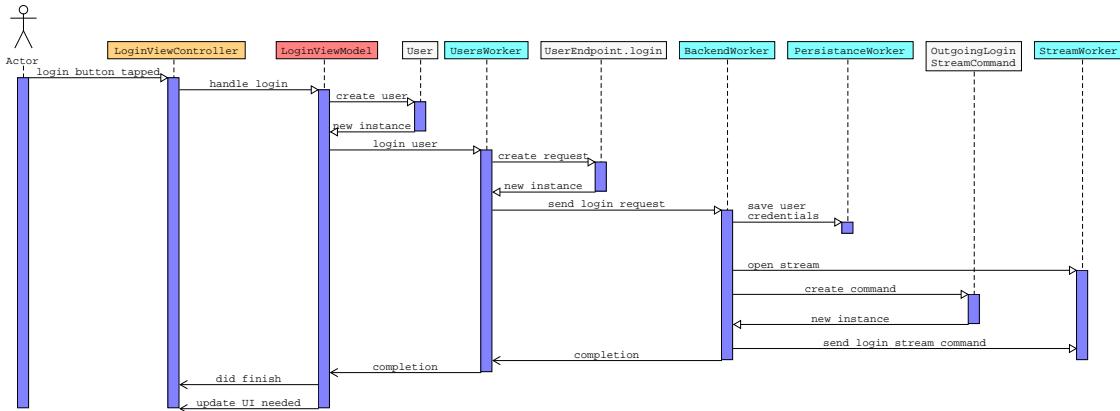


Abbildung 7.7: Sequenzdiagramm für die Anmeldung in M2

7.4 Darstellung von Listenansichten

7.4.1 Überblick

Unter iOS gibt es zwei maßgebliche UI-Elemente zur Darstellung einer Liste:

- UITableView zur Darstellung einer herkömmlichen Liste ([Apple 2019](#)).
- UICollectionView zur Darstellung einer Liste im Kachel-Layout ([Apple 2019](#)).

In M2 wird für die Implementierung aller Listenansichten die UITableView verwendet. Die UITableView bekommt zur Darstellung ihrer enthaltenen Objekte eine UITableViewDataSource ([Apple 2019](#)). Bei der UITableViewDataSource handelt es sich um ein Protokoll, welches in der Regel von dem ViewController implementiert wird, dessen Teil die UITableView ist. Im folgenden Klassendiagramm (Abb. 7.8) wird ein beispielhafter Entwurf eines ViewControllers mit einer UITableView und einer Implementierung der UITableViewDataSource visualisiert.

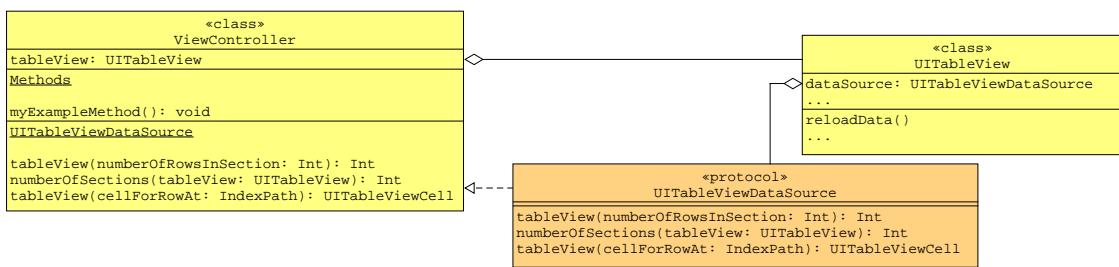


Abbildung 7.8: Beispielhaftes Klassendiagramm für die Implementierung einer UITableView inklusive UITableViewDataSource

Jeder einzelne Listeneintrag wird durch eine Instanz einer `UITableViewCell` repräsentiert. Um die Gestaltung dieser Zelle eigens vornehmen zu können, wird von dieser geerbt. In der Funktion `tableView(cellForRowAt: IndexPath)` der `UITableViewDataSource` kann dann eine Instanz für die jeweilige Zeile zurückgegeben werden.

An dieser Stelle kommt die Implementierung der `ListViewModels` und `CellViewModels` zum Einsatz (Abschnitt 5.3.5). Folgend wird dies mit einem Codebeispiel in Kombination mit der `UITableViewDataSource` demonstriert.

```
1 func tableView(_ tableView: UITableView,
2                 cellForRowAt indexPath: IndexPath) -> UITableViewCell {
3     let cell = tableView.dequeueReusableCell(withIdentifier: "MyCell",
4                                         for: indexPath)
5                                         as! MyTableViewCell
6     let cellViewModel = listViewModel.data[indexPath.row]
7     cell.viewModel = cellViewModel
8     return cell
9 }
```

Quellcode 7.1: Codebeispiel `UITableViewDataSource`

Im obigen Beispiel 7.1 wird in Codezeile 3 die Zelle `MyTableViewCell` initialisiert. Diese muss ein `CellViewModel` beinhalten können. Das `CellViewModel` wird in Zeile 6 aus dem `ListViewModel` des ViewControllers an Stelle der aufgerufenen Reihe `indexPath.row` geholt und in Zeile 7 in `MyTableViewCell` gesetzt. Dieses Beispiel zeigt eine Implementierung, welche sich in allen Listenansichten von M2 so wiederfindet.

7.4.2 Animierte Aktualisierung des Listeninhaltes

Apple

Beim Aufruf der Funktion `reloadData()` (Abb. 7.8) bezieht die `UITableView` ihre Informationen aus der Implementierung der `UITableViewDataSource` neu. Dies geschieht jedoch ohne Animation. Um eine Änderung² einer oder mehrerer Listeneinträge zu animieren, erfordert es die Nutzung einer komplizierten Schnittstelle, welche bei falscher Implementierung schnell unerwünschte Abstürze verursachen kann. Des Weiteren erzeugt die Art und Weise, wie die Schnittstelle von Apple programmiert ist, unleserlich verschachtelten Code, in erster Linie, da der Programmierer selbst eine Unterscheidung zwischen alten und neuen Datensätzen vornehmen muss. Dieses Problem ist Apple selbst inzwischen bekannt, weswegen in der Vorabversion des im Herbst 2019 erscheinenden iOS 13 eine neue Schnittstelle veröffentlicht wird ([Apple 2019](#)).

²Einfügen, Verändern oder Löschen

DifferenceKit

Da die neue Schnittstelle `UITableViewDiffableDataSource` allerdings noch nicht offiziell veröffentlicht ist, wird in M2 die externe Bibliothek `DifferenceKit` (Abschnitt 6.3.5) genutzt, um Listenansichten animiert und mit übersichtlichem Code zu aktualisieren. `DifferenceKit` stellt ein Protokoll zur Verfügung, welches im Model für den Listeneintrag einer Liste implementiert wird. Dieses dient zur eindeutigen Identifikation eines jeden Eintrags, damit diese verglichen werden können. Als `extension` auf die `UITableView` stellt `DifferenceKit` dann eine eigene Funktion zur Aktualisierung der Daten zur Verfügung, welche als Parameter ein `Changeset` erwartet. Dieses `Changeset` kann einfach mit einer Liste der alten Models und einer Liste der neuen Models initialisiert werden.

Zur vereinfachten Nutzung des Aktualisierungsablaufs in M2 über `DifferenceKit` wird eine erbende Variante der `Observable`-Implementierung entwickelt und als Provider genutzt, welcher bei einer Änderung der Daten nicht nur die neuen Models, sondern auch die alten bereitstellt und daraus ein `Changeset` generiert. So kann `DifferenceKit` direkt mit der Antwort des Providers genutzt werden.

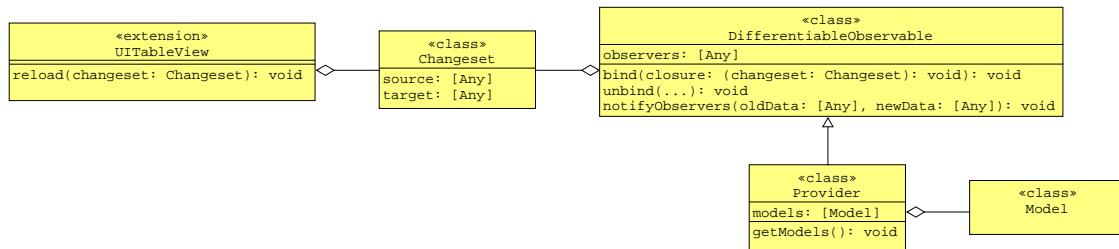


Abbildung 7.9: Klassendiagramm für die Implementierung von DifferenceKit in M2

7.5 HTTP-Requests

Zum Versenden von HTTP-Requests an das Backend wird der `BackendWorker` implementiert. Dieser verkapselt die externe Bibliothek `Alamofire` und setzt sie zum Versenden der Requests ein.

Im folgenden Klassendiagramm 7.10 ist der Aufbau des BackendWorkers dargestellt.

7 Implementierung

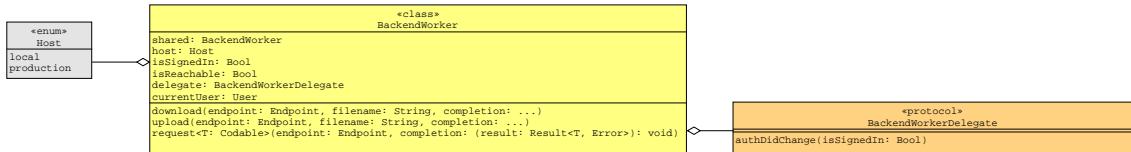


Abbildung 7.10: Klassendiagramm für die Implementierung des BackendWorkers

Alle drei Funktionen des BackendWorkers erwarten als ersten Parameter ein Objekt, welches das Protokoll `Endpoint` implementiert. `Endpoint` wiederum implementiert das Protokoll `URLConvertible` aus Alamofire, um kompatibel zur Bibliothek zu bleiben. In Swift werden Enums als Objekte behandelt, können Funktionen, `Computed Properties`³ und sogar Protokolle implementieren oder von anderen Datentypen erben. Diese Eigenschaft wird sich zu Nutzen gemacht, indem für jede Kategorie von HTTP-Requests in M2 ein Enum erstellt wird, welches das Protokoll `Endpoint` implementiert. Als Beispiel folgt ein Klassendiagramm zum Aufbau des FriendsRequestsEndpoint (Abb. 7.11).

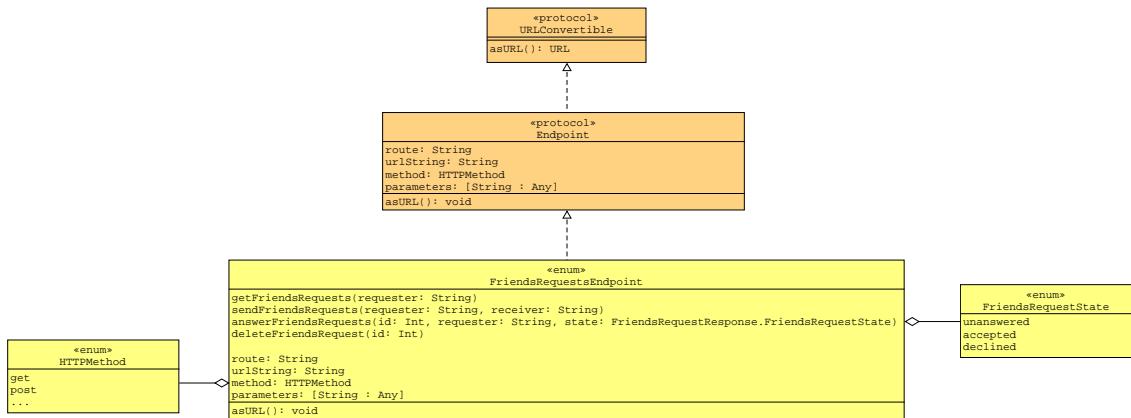


Abbildung 7.11: Klassendiagramm für die Implementierung des FriendsRequestsEndpoints

Wird ein Endpoint in die Funktion `request<T>(...)` (Quellcode 7.2) des BackendWorkers gegeben, werden für die Nutzung mit Alamofire alle benötigten Informationen aus den durch das Protokoll `Endpoint` implementierten Attributen bezogen. Gibt ein HTTP-Request über den HTTP-Status-Code hinaus auch einen JSON-String als Antwort zurück, wird dieser mithilfe des `JSONDecoder` ([Apple 2019](#)) in ein Objekt mit der Implementierung des Protokolls `Codable` konvertiert und im Completion-Block der Funktion zurückgegeben. Der Datentyp des Generics `T` wird beim Aufruf der Funktion in Swift automatisch inferiert. In M2 wurden hierzu diverse Strukturen

³Variablen, welche nur gelesen und nicht beschrieben werden können und bei Abruf ihren Wert mit einer Funktion berechnen

erstellt. Eine vollständige Liste aller Response-Strukturen findet sich in der HTML-Dokumentation im Anhang auf der beigefügten CD (Abschnitt A.1).

Folgend ist der Funktionskopf für das Absenden eines HTTP-Requests aufgeführt.

```
1 func request<T: Codable>(_ endpoint: Endpoint,
2                           completion: ((_ result: Swift.Result<T, Error>) -> Void)?)
```

Quellcode 7.2: Funktionskopf für das Absenden eines HTTP-Requests im BackendWorker

Der BackendWorker nutzt außerdem das Singleton-Pattern, um sicherzustellen, dass nur eine Instanz der Klasse zur Zeit aktiv ist ([Eilebrecht & Starke 2019: 38 f.](#)).

7.6 Stream-Verbindung

7.6.1 Übersicht

In M2 wird die Verbindung zum Stream-Teil des Backends mittels TCP hergestellt. TCP ist ein Protokoll zur zuverlässigen Datenübermittlung eines Streams, da zu jedem versendeten Datenpaket auch eine Empfangsbestätigung erfolgen muss ([Information Sciences Institute 1981](#)). Im Gegensatz dazu steht UDP, welches keine Empfangsbestätigung einfordert, dafür allerdings schneller in der Übertragung der Datenpakete ist ([Postel 1980](#)). Für die Verwaltung aller Stream-bezogenen Aufgaben wird der **StreamWorker** implementiert. Über diesen ist es möglich, Steuerungsbefehle wie Öffnen oder Schließen des Streams zu veranlassen. Außerdem können Daten an den Stream gesendet und über den Stream empfangen werden. Hierzu verkapselt der StreamWorker Apples Klassen **InputStream** ([Apple 2019](#)) und **OutputStream** ([Apple 2019](#)). Der StreamWorker nutzt wie der BackendWorker das Singleton-Pattern, um sicherzustellen, dass nur eine Instanz der Klasse zur Zeit aktiv ist ([Eilebrecht & Starke 2019: 38 f.](#)). Im unteren Klassendiagramm 7.12 ist ein Überblick des StreamWorkers visualisiert. Zur besseren Übersicht wurden nur die relevanten Attribute aufgenommen. Eine vollständige HTML-Dokumentation findet sich im Anhang auf der beigefügten CD (Abschnitt A.1).

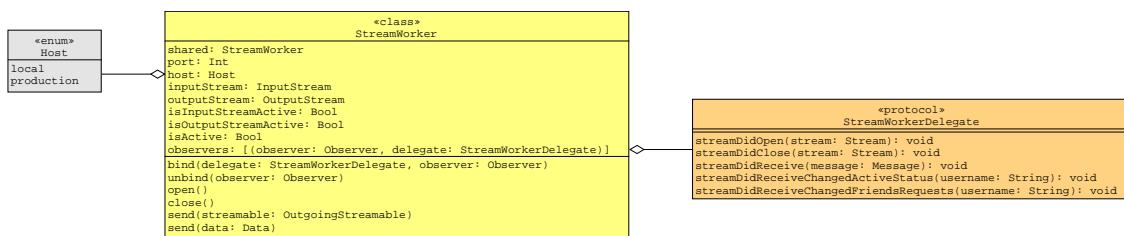


Abbildung 7.12: Klassendiagramm des StreamWorkers

7.6.2 Senden von Daten

Der StreamWorker erwartet zum Versenden von Daten ein Objekt, welches das Protokoll `OutgoingStreamable` implementiert. Im folgenden Klassendiagramm 7.13 ist der Aufbau dieses Protokolls dargestellt.

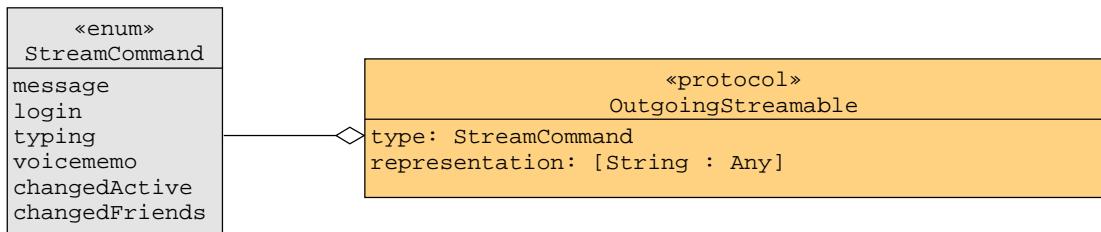


Abbildung 7.13: Klassendiagramm für das Protokoll `OutgoingStreamable`

Jedes `OutgoingStreamable` besitzt einen Command. Des Weiteren muss dieses Objekt als Dictionary darstellbar sein, damit es für die Übertragung zum Stream in einen JSON-String konvertiert werden kann.

Mit der Funktion `send(streamable: OutgoingStreamable)` (Abb. 7.12) kann das Datenpaket abgeschickt werden.

7.6.3 Empfangen von Daten

Beim Eintreffen von Daten am TCP-Stream werden alle Zeichen bis Zeilenende zu einem JSON-String gesammelt und versucht, in ein `IncomingStreamable` umgewandelt zu werden. Im folgenden Klassendiagramm 7.14 ist der Aufbau des `IncomingStreamables` dargestellt.

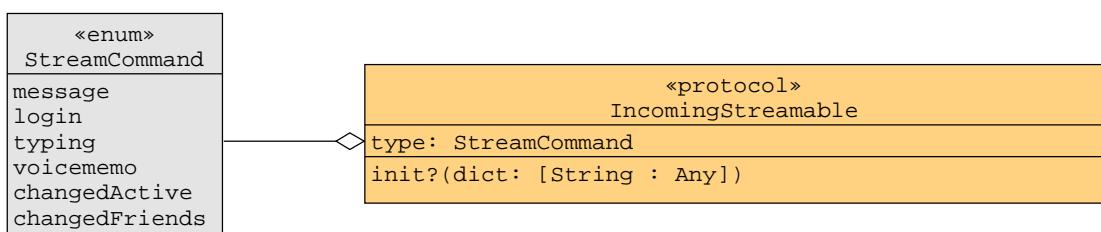


Abbildung 7.14: Klassendiagramm für das Protokoll `IncomingStreamable`

Um andere Objekte über den Erhalt eines eintreffenden Datenpaketes informieren zu können, bietet der StreamWorker ein `StreamWorkerDelegate`. Beliebig viele Objekte können sich mit der Funktion `bind(...)` (Abb. 7.12) als Observer des StreamWorkers registrieren und das `StreamWorkerDelegate` implementieren, um über eintreffende Datenpakete informiert werden zu können.

7.7 Freundschaftsanfragen

7.7.1 Versenden einer Freundschaftsanfrage

Zum Versenden einer Freundschaftsanfrage wird der `AddUserViewController` implementiert. Auch hier existiert ein dazugehöriges `AddUserViewModel`. In der folgenden Abbildung 7.15 wird diese Darstellung gezeigt. Der Nutzer kann einen Nutzernamen in ein UITextField eingeben. Bei jedem eingetippten Buchstaben nutzt das AddUserViewModel den BackendWorker, um eine Prüfungsanfrage zur Existenz des eingetippten Nutzers zum Backend zu senden. Ist dies der Fall, erscheint unter dem UITextField ein grünes UILabel mit dem Text „Nutzer gefunden“, um dies zu bestätigen. Tippt der Nutzer einen Namen ein, der nicht existiert, erscheint kein grünes UILabel.



Abbildung 7.15: AddUserViewController bei gefundenem Nutzer

Folgend wird in dem Sequenzdiagramm 7.16 der Ablauf aller Prozesse bei einem eingetippten Buchstaben in das UITextField offen gelegt.

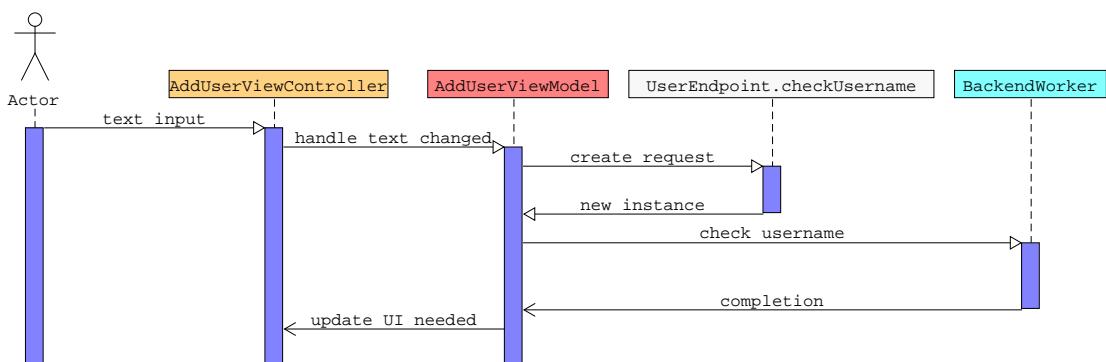


Abbildung 7.16: Sequenzdiagramm für eine Texteingabe im AddUserViewController

Mit einem Tippen auf den Knopf „Anfrage senden“ wird eine Freundschaftsanfrage an den eingetippten Nutzer gesendet. Existiert dieser nicht im Backend, erscheint ein rotes UILabel mit dem Text „Nutzer nicht gefunden“. Existiert der Nutzer im Backend, schließt sich der AddUserViewController und der Nutzer befindet sich wieder in der Hauptansicht. Über den `FriendsRequestsProvider` wird mit dem Schließen

des AddUserViewControllers dann die Freundschaftsanfrage an den Nutzer geschickt. Hierzu wird erst der **BackendWorker** für das Absenden eines HTTP-Requests genutzt, danach der **StreamWorker** um einen **OutgoingChangedFriendsRequestsStreamCommand** an den Stream zu senden. Im folgenden Sequenzdiagramm 7.17 wird der Ablauf aller Prozesse beim erfolgreichen Absenden einer Freundschaftsanfrage dargestellt.

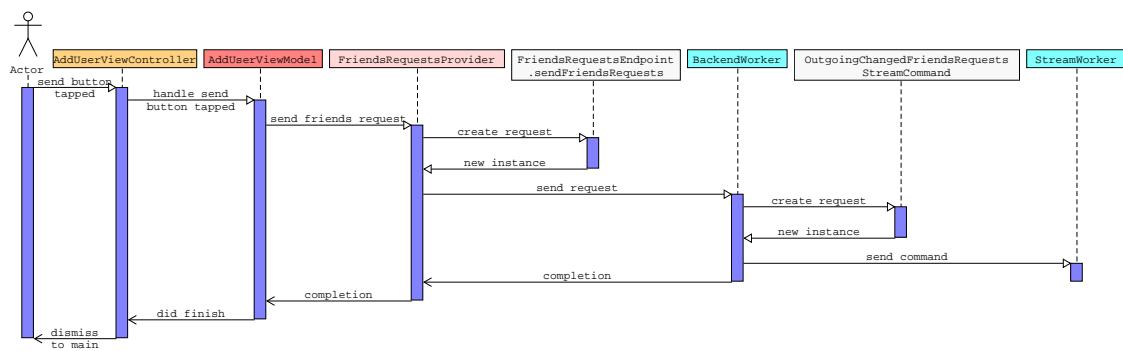


Abbildung 7.17: Sequenzdiagramm für das erfolgreiche Absenden einer Freundschaftsanfrage mit dem AddUserViewController

7.7.2 Empfangen einer Freundschaftsanfrage

Der ChatsListViewController wird als Observer des StreamWorkers registriert. Wurde bei einer eingehenden Freundschaftsanfrage das eintreffende Datenpaket erfolgreich als **IncomingChangedFriendsRequestsStreamCommand** identifiziert, benachrichtigt der StreamWorker all seine Observer und somit auch den ChatsViewController. Besteht die Anfrage, so aktualisiert dieser die Liste der Freundschaftsanfragen und zeigt diese in der UINavigationBar des ChatsViewControllers an (Abb. 7.18).

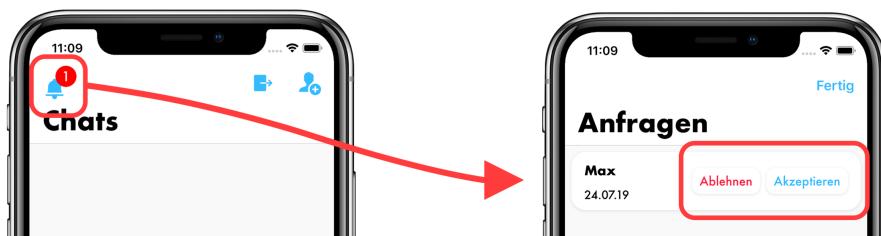


Abbildung 7.18: Badge in der UINavigationBar des ChatsViewControllers (l) und Freundschaftsanfrage in der Freundschaftsanfragenliste (r)

Bevor der Stream im Backend die JSON mit der Benachrichtigung an den Empfänger sendet, überprüft er, ob dieser online ist. Befindet sich der Nutzer nicht in der App, werden keine Daten über den Stream geschickt, sondern es wird eine Push-Benachrichtigung über den Apple Push Notification Service direkt an das Gerät versendet. Die aktuelle Liste der Freundschaftsanfragen wird dann beim nächsten App-Start geholt.

7.7.3 Beantworten einer Freundschaftsanfrage

Die Liste der Freundschaftsanfragen wird im `FriendsRequestsViewController` mit einer `UITableView` dargestellt (Abb. 7.18). Der `FriendsRequestsViewController` beinhaltet ein `FriendsRequestsListViewModel` mit einer Liste von `FriendsRequestCellViewModels`, welches seinen Inhalt vom `FriendsRequestsProvider` bezieht. Der Aufbau ist hier der Gleiche wie im Chats-ViewController mit dem `ChatsListViewModel`. Beantwortet der Nutzer die Freundschaftsanfrage durch Tippen eines `UIButton`s entweder zum Akzeptieren oder zum Ablehnen, behandelt das `FriendsRequestCellViewModel` der jeweiligen Zelle diese Aktion. Das `FriendsRequestCellViewModel` nutzt dann den `FriendsRequestsProvider`, um mit dem Aufruf `answerFriendsRequest(...)` die Freundschaftsanfrage zu beantworten (Quellcode 7.3).

```
1 func answerFriendsRequest(friendsRequestResponse: FriendsRequestResponse,
2 state: FriendsRequestState)
```

Quellcode 7.3: Funktionskopf von `answerFriendsRequest`

7.8 Nachrichtenanfragen

Bevor eine Konversation in M2 gestartet werden kann, müssen beide Nutzer sich im selben Chat befinden. Vom ChatsViewController aus kommt der Nutzer durch Tippen auf eine ChatsTableViewCell zum `MessagesViewController`. Das Tippen auf die Zelle wird durch die Funktion aus Quellcode 7.4 vom `UITableViewDelegate` (Apple 2019) mitgeteilt.

```
1 func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)
```

Quellcode 7.4: Funktionskopf von Funktion aus `UITableViewDelegate`

Der `MessagesViewController` wird mit einem `MessagesListViewModel` initialisiert, welches wiederum eine Liste von `MessageCellViewModels` beinhaltet. Die Daten werden, wie üblich mit einem Provider, dem `MessagesProvider` zur Verfügung gestellt. Der `MessagesViewController` wird auf dem `ChatsViewController` präsentiert.

7 Implementierung

Beim Start des MessagesViewControllers wird über das MessagesListViewModel der Start des Chats signalisiert. Hierzu nutzt es den MessagesProvider, welcher eine Instanz des Chat-Models beinhaltet. Zuerst wird der aktive Chat des aktuellen Nutzers im Backend über einen HTTP-Request geändert, dann wird sicherheitshalber für beide Nutzer überprüft, ob diese online sind. Im folgenden Sequenzdiagramm 7.19 ist der Ablauf dieser Prozesse dargestellt.

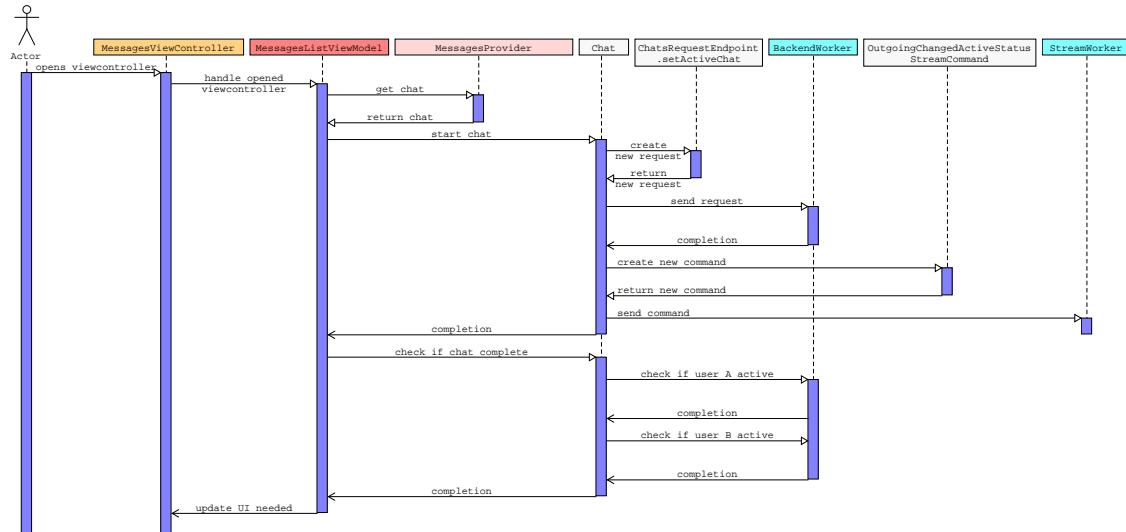


Abbildung 7.19: Sequenzdiagramm zum Ablauf des Chat-Startes

Für eine Veranschaulichung des Aufbaus des Chat-Models folgt Abbildung 7.20.

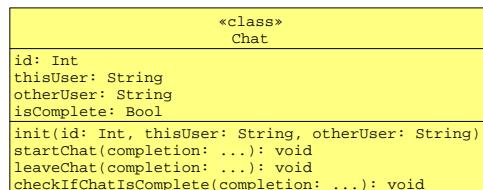


Abbildung 7.20: Klassendiagramm für den Aufbau des Chat-Models

Starten und Verlassen des Chats werden über die Funktionen `startChat(...)` und `leaveChat(...)` vorgenommen. Die Überprüfung, ob beide Teilnehmer online sind, erfolgt über `checkIfChatIsComplete(...)`. Folgend ist der Aufbau der Tabelle `users` im Backend visualisiert. Hier wird für die Chat-basierten Aktionen die Spalte `active_chat` überprüft. Hat die Spalte den Wert `NULL`, ist der Nutzer aktuell in gar keinem Chat. Hat sie einen Wert, beinhaltet sie den Nutzernamen des Nutzers für den der Chat geöffnet wurde. In der folgenden Abbildung 7.21 ist der Aufbau der Tabelle `users` dargestellt.

Database					
Table users					
id	username	password	salt	device_token	active_chat
...	User_A	User_B
...	User_B	NULL

Abbildung 7.21: Aufbau der Tabelle users im Backend

Nachdem ein Nutzer einen Chat gestartet hat, sendet er wie in Abbildung 7.19 beschrieben einen `OutgoingChangedActiveStatusStreamCommand` zu dem anderen Chatteilnehmer. Das Backend überprüft vor Weiterleitung an den Empfänger, ob dieser online, also mit dem TCP-Stream verbunden ist. Ist dies nicht der Fall, wird eine Push-Benachrichtigung mit dem Apple Push Notification Service versendet, um den Empfänger aufzufordern, dem Chat beizutreten.

7.9 Nachrichtenaustausch

Zur Darstellung der ausgetauschten Nachrichten wird der `MessagesViewController` verwendet. Dieser stellt sowohl die Text- als auch die Sprachnachrichten mit einer `UITableView` dar. Auf den grundlegenden Aufbau des `MessagesViewController`s wurde bereits in Abschnitt 7.8 eingegangen. Eine Besonderheit in Bezug auf den Nachrichtenaustausch stellt in diesem Fall die Umsetzung mit der `UITableView` dar. Die `UITableView` ist für eine klassische Listendarstellung von oben nach unten ausgerichtet. In der folgenden Abbildung 7.22 ist die unveränderte, standardmäßige Anordnung von Zellen der `UITableView` dargestellt. Im linken Teil der Abbildung ist ein Zustand mit sechs Zellen, im rechten Teil ein Zustand mit drei weiteren hinzugefügten Zellen. Die Zahlen innerhalb der Zellen der Abbildung stellen die Reihenfolge dar, in der sie hinzugefügt wurden.

7 Implementierung

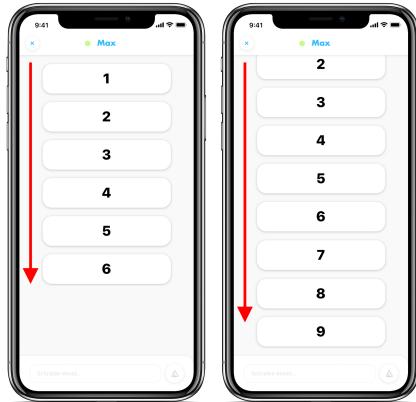


Abbildung 7.22: Standardmäßige Anordnung der UITableView

Bei einem Messenger wird die erste Nachricht klassischerweise allerdings nicht oben angezeigt und weitere Nachrichten von oben nach unten hinzugefügt. Die erste Nachricht startet am Boden und weitere Nachrichten werden darunter eingefügt während der Rest nach oben bewegt wird. Dieses Verhalten ist in der folgenden Abbildung 7.23 dargestellt.



Abbildung 7.23: Bodenorientierte Anordnung der UITableView

Es gibt mehrere Möglichkeiten, dieses Verhalten trotzdem zu erzeugen. Die Art der Anordnung und die Scrollrichtung lassen sich bei der UITableView standardmäßig nicht anpassen. Versucht man beispielsweise, nach jedem hinzugefügtem Listeneintrag direkt wieder programmatisch zum Boden zu scrollen, entsteht eine flackernde Animation, die den Nutzer verwirren kann.

Als Lösung gibt es einen sehr außergewöhnlichen Umweg: Die UITableView wird um 180° gedreht gezeichnet und ihre enthaltenen Zellen ebenfalls. Außerdem wird die

7 Implementierung

Reihenfolge der Nachrichtenliste im Model invertiert.

Eine UITableView ist wie jede Ansicht in iOS eine Subklasse der UIView. Jede UIView besitzt standardmäßig ein Attribut `transform` ([Apple 2019](#)), welches erlaubt mithilfe der nativen Grafik-Bibliothek `CoreGraphics`⁴ die Rotation und Skalierung der dargestellten UIView zu verändern ([Apple 2019](#)). Dieses Attribut besitzt dementsprechend auch eine UITableViewCell.

Im ersten Schritt wird die Reihenfolge der Nachrichtenliste im Model invertiert (Abb. 7.24).

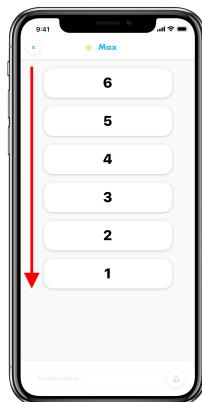


Abbildung 7.24: Standardmäßige Darstellung der UITableView mit invertierter Nachrichtenliste in UITableViewDataSource

Im zweiten Schritt wird die gesamte UITableView um 180° gedreht. Dies ist in der folgenden Abbildung 7.25 visualisiert. Die Darstellungsrichtung der UITableView wird mit dem langen roten Pfeil auf der linken Seite angezeigt, die Darstellungsrichtung der UITableViewCells mit kleinen roten Pfeilen auf der rechten Seite.

⁴Apples Bibliothek zur Kerndarstellung auf geringster Ebene

7 Implementierung

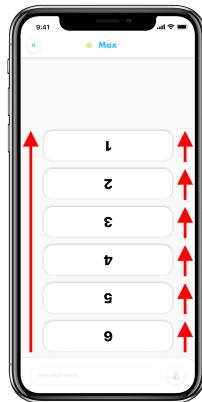


Abbildung 7.25: Um 180° gedrehte UITableView

Die UITableView funktioniert ganz normal wie vorher. Vergleichsweise ist es jetzt so, als hätte man ein Video um 180° gedreht, und nicht seine Darsteller gebeten, sich auf den Kopf zu stellen. Da aber natürlich jetzt auch der Inhalt um 180° gedreht ist, wird als dritter Schritt in der UITableViewDataSource bei der Initialisierung einer jeden Zelle diese ebenfalls um 180° gedreht. Folgend findet sich eine Visualisierung in Abbildung 7.26.

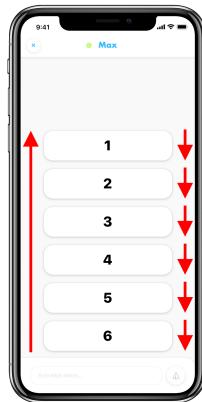


Abbildung 7.26: Um 180° gedrehte UITableView mit um 180° gedrehten UITableView-Cells

Das Ergebnis ist eine korrekt funktionierende UITableView mit flüssigen Aktualisierungsanimationen und der gewünschten Anordnung von unten nach oben. Vergleichsweise haben wir nun also ein um 180° gedrehtes Video, in dem Darsteller gefilmt wurden, die sich ebenfalls auf den Kopf gestellt haben. Die UITableView im MessagesViewController wird beim Laden noch vor der Präsentation in den angegebenen Schritten gedreht.

7.9.1 Textnachrichten in Echtzeit

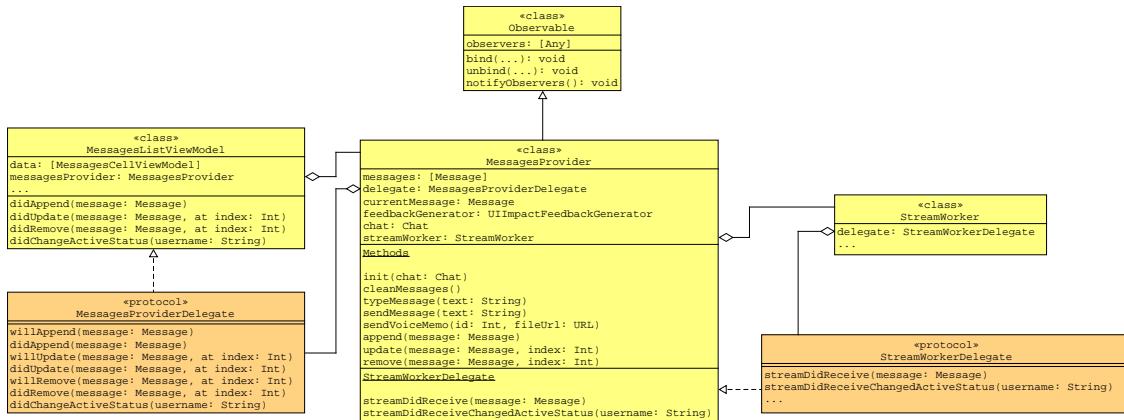


Abbildung 7.27: Klassendiagramm des MessagesProviders und seiner Abhängigkeiten

Eine der wichtigsten Funktionen von M2 ist die buchstabenweise Darstellung von Nachrichten während sie vom anderen Chatteilnehmer eingegeben werden. Fängt ein Nutzer an, eine Nachricht einzutippen, wird jeder eingetippte Buchstabe im MessagesListViewModel behandelt und die Funktion `typeMessage(...)` des `MessagesProviders` aufgerufen. Der Aufbau des `MessagesProviders` ist in der Abbildung 7.27 dargestellt. Handelt es sich um den ersten Buchstaben, wird eine neue Instanz des `TextMessage`-Models erstellt und als `currentMessage` gesetzt. Das Attribut `isEditing` der `TextMessage` erhält den Wert `true`. Werden weitere Buchstaben eingetippt, wird die aktuelle Nachricht über das Attribut `currentMessage` geholt und bearbeitet. Der Stream wird mittels `OutgoingTypingStreamCommand` über das Ereignis benachrichtigt. Dieser Ablauf ist im folgenden Sequenzdiagramm 7.28 dargestellt.

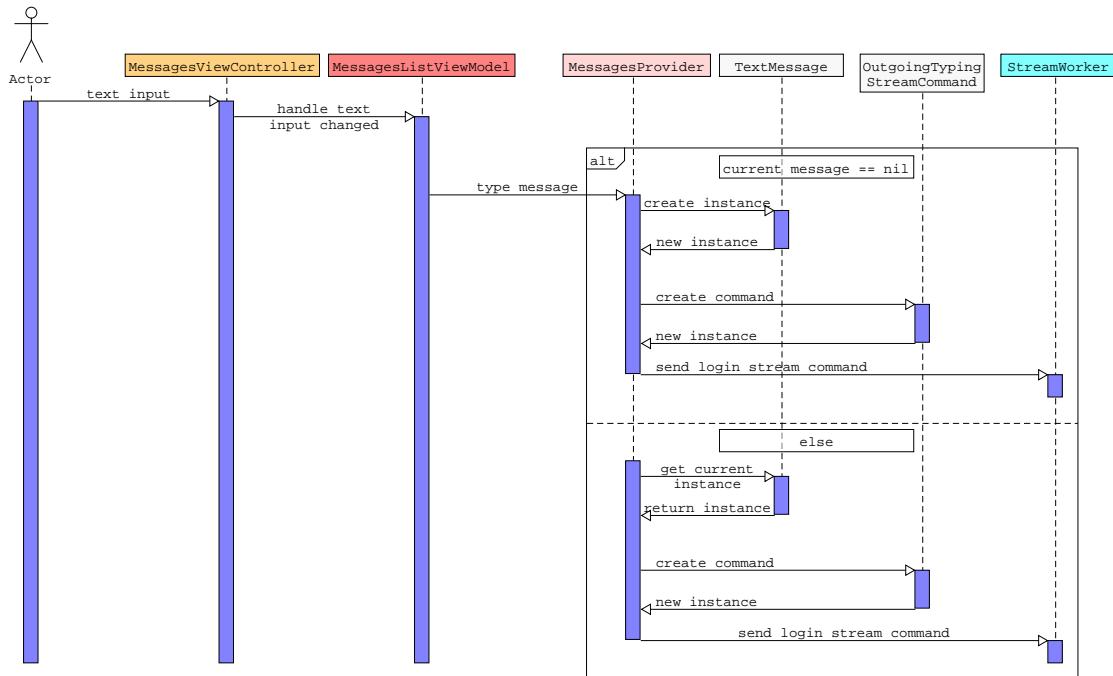


Abbildung 7.28: Sequenzdiagramm eines eingetippten Buchstabens einer Nachricht im `MessagesViewController`

Ist die Nachricht fertig eingetippt, und der Nutzer betätigt den Senden-Knopf, wird das `isEditing`-Attribut des `TextMessage`-Objektes auf `false` gesetzt und per `MessagesProvider` der Liste von Nachrichten angefügt. Der `StreamWorker` sendet danach einen `OutgoingMessageStreamCommand` an den Empfänger. Außerdem wird mit dem `UIImpactFeedbackGenerator` haptisches Feedback erzeugt, um dem Nutzer zu signalisieren, dass die Nachricht erfolgreich gesendet wurde. Die Veränderung der Nachrichtenliste ruft die Funktion `notifyObserver()` auf, denn der `MessagesProvider` ist `Observable` und benachrichtigt so den `MessagesViewController` über neue Nachrichten. Das gilt sowohl für eigens gesendete als auch für empfangene Nachrichten.

Empfängt ein Nutzer eine Nachricht, die gerade getippt oder abgesendet wurde, meldet sich das `StreamWorkerDelegate` im `StreamWorker`. Da der `MessagesProvider` auch ein Observer des `StreamWorkers` ist (Abb. 7.27), behandelt dieser die eintreffende Nachricht. Wird die Nachricht gerade getippt, wird über die Id der Nachricht geprüft, ob diese bereits in der Nachrichtenliste vorhanden ist und nur aktualisiert werden muss, oder ob sie neu hinzugefügt werden muss. Das Attribut `isEditing` steuert die Darstellung der `MessagesTableViewCell` über das `MessagesCellViewModel`.

7.9.2 Sprachnachrichten mit Transkription

Senden und Empfangen von Sprachnachrichten

Das Senden und Empfangen von Sprachnachrichten ist deckungsgleich mit dem Ablauf der Textnachrichten, da die Nachrichtenliste im `MessagesProvider` eine Liste aus `Message`-Objekten ist, welcher der Obertyp von Text- und Sprachnachrichten ist. Beim Versenden wird allerdings im Gegensatz zur Textnachricht nicht der Inhalt per Stream gesendet, sondern per HTTP-Request über den `BackendWorker` hochgeladen. Verschickt wird im Stream dann lediglich die Id der Sprachnachricht auf dem Server, welche dann beim Empfänger zum Download der Audiodatei genutzt wird.

Aufnahme der Audiodateien

Zur Aufnahme einer Sprachnachricht wird der `AudioRecordingWorker` genutzt. Dieser verkapselt Apples Klasse `AVAudioRecorder` aus der Bibliothek `AVFoundation` zur Aufnahme von Audio. In Abbildung 5.13 wurde bereits das Klassendiagramm des `AudioRecordingWorkers` dargestellt.

Wiedergabe der Audiodateien

Eine Wiedergabe der Sprachnachricht kann vom Nutzer durch Betätigen eines Knopfes in der `VoiceMemoReceiverTableViewCell` ausgelöst werden. Das `VoiceMemoCellViewModel` leitet diese Anfrage weiter an den `AudioPlayingWorker`. Dieser wird zur Wiedergabe aller Audiodateien in M2 genutzt, einschließlich der Sprachnachrichten. Der Aufbau des `AudioPlayingWorkers` ist deckungsgleich mit dem des `AudioRecordingWorkers`, jedoch implementiert er den `AVAudioPlayer` aus der Bibliothek `AVFoundation` statt des `AVAudioRecorders`.

Transkription der Audiodateien

Eine Transkription zur Anzeige des textuellen Inhaltes der Sprachnachricht kann vom Nutzer durch Betätigen eines Knopfes in der `VoiceMemoReceiverTableViewCell` ausgelöst werden. Das `VoiceMemoCellViewModel` behandelt diese Anfrage und leitet sie an den `SpeechRecognitionWorker` weiter. Dieser verkapselt die Klasse `SFSpeechRecognizer` aus Apples Bibliothek `Speech` ([Apple 2019](#)). Einer Instanz dieser Klasse kann ein `SFSpeechURLRecognitionRequest` mit dem Dateipfad zur Audiodatei übergeben werden und so ein `RecognitionRequest` angestoßen werden. In der Antwort mit einer Closure wird als Parameter dann ein `SFSpeechRecognitionResult` zurückgegeben, welches u.a. eine Liste aller möglichen Transkriptionen des Inhaltes enthält. Es beinhaltet ebenfalls ein Attribut zur bestmöglichen Transkription, welches als String konvertiert werden kann und dem Nutzer so über das `VoiceMemoCellViewModel` angezeigt wird. Der Ablauf der Erkennung

selbst funktioniert durch die von Apple angebotene API serverseitig und nutzt dieselbe Schnittstelle wie Apples Sprachassistent Siri.

7.9.3 Soundeffekte

In M2 werden mit dem AudioPlayingWorker außerdem Soundeffekte für Ereignisse abgespielt. Die Soundeffekte sind als M4A-Audiodateien im Ressourcen-Ordner der App angelegt und werden durch das Enum `Sound` repräsentiert. Das Enum erbt vom Datentypen String. Ein case repräsentiert sich selbst anhand seiner Namensgebung als String, sofern nicht anders definiert. Die Computed Property `url` baut sich aus der Stringrepräsentierung eines cases und dem Dateipfad zum Ressourcen-Ordner eine URL zusammen. In der Abbildung 7.29 ist der Aufbau des Enums dargestellt. Im Anhang A.6 findet sich eine Liste aller Soundeffekte.

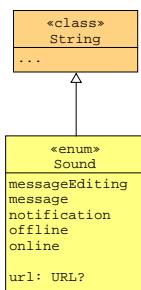


Abbildung 7.29: Enum zur Repräsentierung eines Soundeffektes

7.9.4 Datenpersistenz

Zur Speicherung von Werten wird der `PersistanceWorker` eingesetzt. Dieser ver kapselt Apples Klasse `UserDefaults` ([Apple 2019](#)). `UserDefaults` speichert mit einem einfachen Key-Value-Prinzip primitive Datentypen im Preferences-Ordner der Kernstruktur der App. Dies erlaubt eine schnelle und unkomplizierte Speicherung und Abfrage der Werte. Dieser Ansatz ist allerdings nicht sicher, da bei Zugriff auf das Gerät mit der installierten App mit einigen Umwegen auf die Kernstruktur der App zugegriffen werden kann, und diese Werte ausgelesen und sogar verändert werden können. Mit den `UserDefaults` sollten also nur Werte gespeichert werden, die nicht sicherheitsrelevant sind.

In M2 wurden folgende Werte in den UserDefaults gespeichert:

- Nutzernname
- Nutzerpasswort
- Nutzer-Id
- Device Token für Push-Benachrichtigungen

Die aktuelle Implementierung zur Speicherung der Daten ist daher aus genannten Gründen unter der Nutzung der UserDefaults nicht für eine Produktionsumgebung ausgelegt. Werte wie das Nutzerpasswort können aktuell über einige Umwege herausgefunden werden. In Hinblick auf eine Produktionsumgebung sollte die **Keychain** aus Apples Bibliothek **Security** verwendet werden ([Apple 2019](#)). Diese ist verschlüsselt, über Apples Cloud-Service **iCloud** geräteübergreifend synchronisiert und nicht von außen zugreifbar.

7.10 Technische Schwierigkeiten

In diesem Abschnitt wird auf einige der Schwierigkeiten eingegangen, die sich bei der Implementierung der Funktionalitäten von M2 ergeben haben.

7.10.1 Hoher Datenverkehr

Eine Konversation in M2 erzeugt im Vergleich zu anderen Messengern durch die buchstabenweise Übertragung einer Nachricht sehr hohen Traffic. Um in der aktuellen Implementierung die Fehlerquote bei der Kommunikation zwischen App und Backend zu minimieren, wird bei jedem eingetippten Buchstaben die Nachricht bis zu diesem Buchstaben übertragen. Dazu kommen weiteren Attribute der Textnachricht, wie Datum oder Absender. Angenommen, ein Nutzer tippt mit einer Geschwindigkeit von fünf Buchstaben pro Sekunde, so werden ebenfalls fünf JSON-Strings pro Sekunde an das Backend gesendet. Außerdem erhöht sich die Größe jedes abgeschickten JSON-Strings mit jedem eingetippten Buchstaben. Normalerweise kann die App dies bei einer stabilen Internetverbindung behandeln. Ist die Verbindung allerdings schlecht, was gerade bei mobilen Geräten schnell der Fall sein kann, fängt die ankommende Nachricht beim Empfänger an zu ruckeln. Als Lösungsansatz wurden die Keys der JSON für den **IncomingTypingStreamCommand** gekürzt, damit weniger Zeichen übertragen werden. Um das Problem der wachsenden Nachricht beim Eintippen zu verhindern, müsste allerdings der gesamte Ablauf der Nachrichtenübertragung verändert werden. Folgend ist der Verlauf empfangener JSON-Strings mehrerer eingetippter Buchstaben dargestellt.

```
{
  "cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ", "t": "test"
}, {"cmd": "e", "d": false, "f": "Vincent", "i": "0", "m": "Das i", "t": "test"}, {"cmd": "s", "d": false, "f": "Vincent", "i": "0", "m": "Das is", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist ", "t": "test"}, {"cmd": "e", "d": false, "f": "Vincent", "i": "0", "m": "Das ist e", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist ei", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist ein", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist ein ", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist ein t", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist ein te", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist ein tes", "t": "test"}, {"cmd": "t", "d": false, "f": "Vincent", "i": "0", "m": "Das ist ein test", "t": "test"}]
```

Abbildung 7.30: Nachrichtenverlauf mehrerer eingetippter Buchstaben

7.10.2 Spracherkennung

Die Spracherkennung zur Transkription der Sprachnachrichten kann in einigen Situationen sehr hilfreich sein. Allerdings benötigt die Erkennung durch Apples Server sowohl eine Internetverbindung als auch einen recht großen Zeitraum. Die Wörter werden ungefähr so schnell erkannt, wie sie gesprochen werden. Je länger die Sprachmemo ist, desto länger braucht sie und desto mehr Datenverkehr erzeugt sie. Apple gibt deshalb eine maximale Anfragendauer von einer Minute vor ([Apple 2019](#)). In der `VoiceMemoReceiverTableViewCell` wird daher bei einer Sprachnachricht, die länger als eine Minute ist, der Knopf zur Spracherkennung ausgeblendet.

Ein weiteres Problem stellt die Erkennung bei lauter Umgebung dar. Diese kann fehlerbehaftet sein. Verlässt sich der Nutzer zu sehr auf die Erkennung und hört sich die Audiodatei nicht an, können so auch Missverständnisse in der Kommunikation entstehen.

8 Fazit

8.1 Auswertung

Mit Ausnahme kleinerer Beeinträchtigungen durch die genannten technischen Schwierigkeiten wurde M2 wie konzipiert vollständig umgesetzt. Die App lässt sich als vollwertiger Messenger einsetzen. Das Backend ist auf einem Cloudserver gehostet und erlaubt die Nutzung der App von überall aus. Die App fühlt sich in der Nutzung hochwertig an und profitiert von flüssigen Animationen und einer optisch ansprechenden Benutzeroberfläche. Das Führen einer Konversation in M2 bewirkt im Vergleich zu herkömmlichen Messengern durch die versteckten Nachrichten beim Absender und die Übertragung der eingetippten Buchstaben beim Empfänger eine durchaus spezielle Form von Kommunikation, in der die Nutzer im direkteren Kontakt zueinander stehen. Das Vorbild eines Telefonates in Textform hat sich in der Umsetzung bewährt.

Für die Nutzung in einer Produktionsumgebung müssten allerdings das Backend stark überarbeitet und die Sicherheitsmängel in der App behoben werden. Nachrichten müssten verschlüsselt verschickt und die Kommunikation zwischen App und Backend für die Reduzierung von Datenverkehr optimiert werden. Außerdem fehlen noch einige Mechanismen wie User-Onboarding¹ oder Anpassen der Nutzerdaten nach der Registrierung. Als konzeptionell schwierig hat sich ebenfalls die Nachrichtenanfrage per Push-Benachrichtigung herausgestellt, da der Empfänger der Anfrage oftmals nicht schnell genug reagieren kann und der Absender recht lange im Chat verweilt, bis der Empfänger beitritt. Hier könnte ein Mechanismus eingebaut werden, der die Anfrage auf dem Gerät des Empfängers wie ein ankommendes Telefonat aussehen lässt und ihn bei Annahme in den Chat weiterleitet.

8.2 Ausblick

8.2.1 Weiterentwicklung des bestehenden Konzeptes

Für die Weiterentwicklung von M2 bietet sich zusätzlich zu den in Abschnitt 8.1 genannten Verbesserungsvorschlägen eine breite Auswahl an Möglichkeiten.

¹Interaktive Bedienungsanleitung für neue Nutzer der App

Teilen des Benutzernamens

Das Teilen des Benutzernamens zum Versenden einer Freundschaftsanfrage geschieht aktuell auf Zuruf außerhalb der App. Hier könnte eine Möglichkeit geschaffen werden, dies innerhalb der App zu machen. Auch wenn die Telefonnummer nicht den Benutzernamen darstellen soll, könnte diese zumindest freiwillig mit dem Account verknüpft werden, um andere Nutzer über die Kontakte finden zu können.

Mehr Einstellungsmöglichkeiten

Die Umsetzung setzt aktuell das Konzept auf eine etwas starre Art und Weise durch. Ein interessanter Gedanke ist es, dem Nutzer mehr Einstellungsmöglichkeiten zu bieten. Bspw. die Anzeige der eigenen Nachrichten oder das Angebot verschiedener Chat-Varianten, damit M2 auch wie ein herkömmlicher Messenger ohne Anzeige der eingetippten Buchstaben verwendet werden kann. Es wäre auch denkbar, dass der Nutzer einstellen könnte, wie lange der Chatverlauf noch bestehen bleibt. So würde verhindert, dass der Verlauf beim Schließen des Chats sofort verschwindet.

Diese Möglichkeiten würden allerdings auch das Konzept von M2 in gewisser Weise wieder entkräften, deswegen wurde vorerst auf die Umsetzung solcher Möglichkeiten verzichtet.

Künstliche Intelligenz

Zusätzlich zu der Transkription der Sprachnachricht ist eine vorstellbare Variante die Zusammenfassung des Inhaltes der Sprachnachricht durch eine künstliche Intelligenz. Dies würde das Problem der Überschreitung der Maximallänge von einer Minute beheben, es würde aber auch wiederum nicht dem Grundsatz des Konzeptes entsprechen, da das Ziel ja eine persönlichere Kommunikation sein soll.

Entfernen der Absenden-Funktion für Textnachrichten

Durch die buchstabenweise Übertragung der Textnachrichten stellt sich die Frage, ob das Absenden einer Nachricht noch notwendig ist. Das würde das Konzept ebenfalls auf eine interessante Art verändern.

Barrierefreiheit

Wie bereits in Abschnitt 2.3 erwähnt, würde sich eine Implementierung von ergänzenden Features zur Unterstützung von Barrierefreiheit anbieten. Mit einer Beachtung aller wichtigen Punkte könnte somit M2 zu einer potentiellen Alternative bspw. für Menschen mit einer Schädigung des Hör- oder Sprechvermögens werden.

8.2.2 Technische Weiterentwicklung

Angesichts der bevorstehenden Veröffentlichung von iOS 13 im Herbst 2019 gäbe es einige neue Funktionalitäten und APIs, die für die Weiterentwicklung von M2 interessant sein könnten.

DarkMode

Eine neue Funktion von iOS 13 ist der DarkMode - ein dunkles Erscheinungsbild der App, alternativ zum aktuellen Design ([Apple 2019](#)). M2 könnte ebenfalls ein dunkles Erscheinungsbild erhalten.

Combine

Der Datenbindungsmechanismus für die Umsetzung des MVVM-Entwurfsmusters könnte in M2 durch die mit iOS 13 erscheinende Bibliothek **Combine** auf nativem Wege implementiert werden. Dies würde eine große Menge an benutzerdefiniertem Code einsparen.

Mac Catalyst

Mit der neuen Funktion **Mac Catalyst** kann eine iOS-App ebenfalls für macOS kompatibel gemacht werden. Dies ist eine interessante Möglichkeit, um mehr Plattformen zu bedienen.

Entwicklung einer Android-App

Um mehr Nutzer zu erreichen, sollte ebenfalls eine Version für das Betriebssystem Android entwickelt werden.

8.2.3 RTT

Tatsächlich ist eine offizielle Standardisierung von Textübertragung in Echtzeit aus Gründen der digitalen Barrierefreiheit schon seit langer Zeit im Gespräch. Im Laufe der letzten Jahre wurde ein Standard namens **RTT** entwickelt, welcher inzwischen offiziell festgelegt ist und sogar Netzanbietern eine Umsetzung vorgibt ([Electronic Code of Federal Regulations 2019](#)). Aufgrund der Tatsache, dass M2 über die Internetverbindung funktioniert und kein Protokoll eines Netzanbieters implementiert, hat sich für diese Arbeit eine Implementierung von RTT nicht angeboten. Für eine Weiterentwicklung scheint eine Implementierung allerdings höchst sinnvoll, da RTT im Grunde Teile des Konzeptes hinter M2 standardisiert.

A Material

A.1 CD

- Bachelor-Thesis „Konzeption und Implementierung einer nativen iOS-App zum Nachrichtenaustausch mit innovativen Funktionalitäten“ im PDF-Format
- Quellcode der iOS-App M2
- Quellcode des Backends von M2
- HTML-Dokumentation des Quellcodes der iOS-App M2
- Demonstrationsvideo der Funktionalitäten der iOS-App M2

A.2 Informationen zum Code

Bis auf den Inhalt der verwendeten Bibliotheken wurde der gesamte Code der iOS-App und des Backends vom Verfasser dieser Arbeit selbst erstellt. Es existiert eine vollständige HTML-Dokumentation des Codes der iOS-App, welche mit dem Tool [Jazzy \(Realm 2019\)](#) aus dem Xcode-Projekt generiert wurde.

A.3 Abbildungen

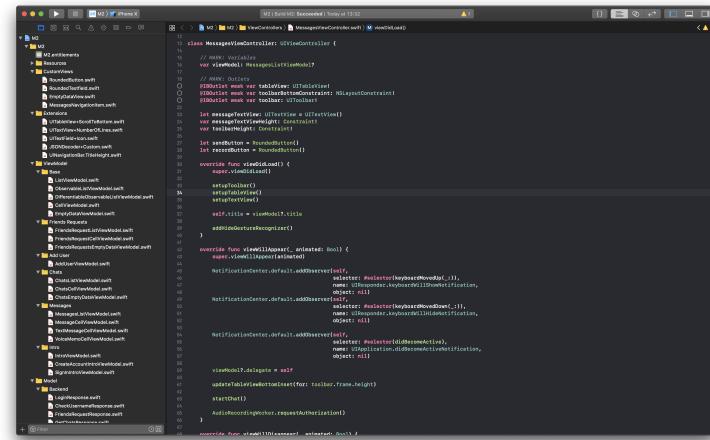


Abbildung A.1: Eine Ansicht des Xcode-Projektes von M2

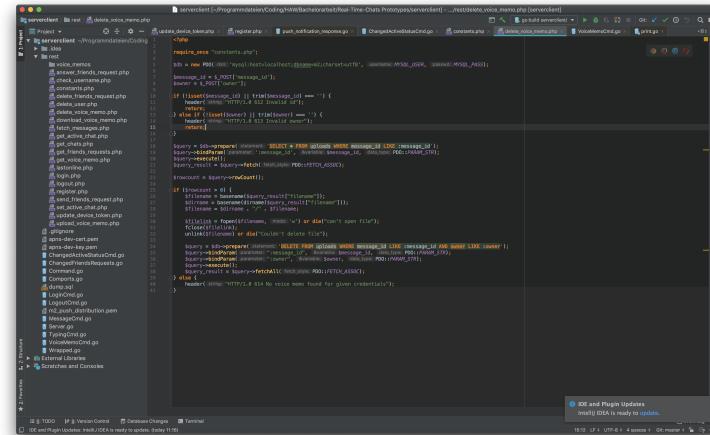


Abbildung A.2: Eine Ansicht des IntelliJ-Projektes vom M2-Backend

A Material

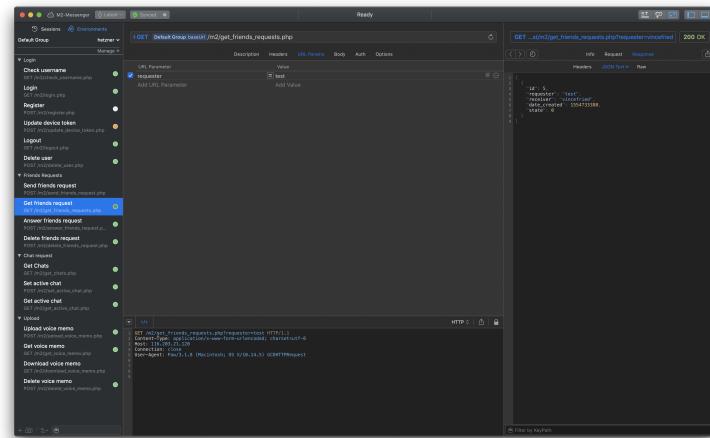


Abbildung A.3: Eine Ansicht der HTTP-Endpunkte von M2 in Paw

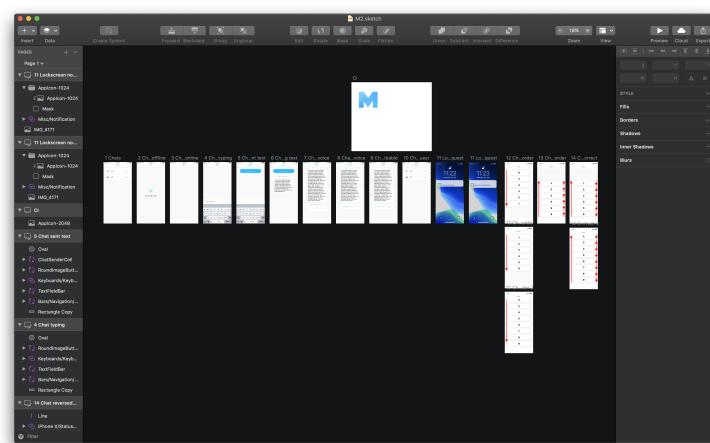


Abbildung A.4: Eine Ansicht der konzipierten Benutzeroberflächen in Sketch

A Material

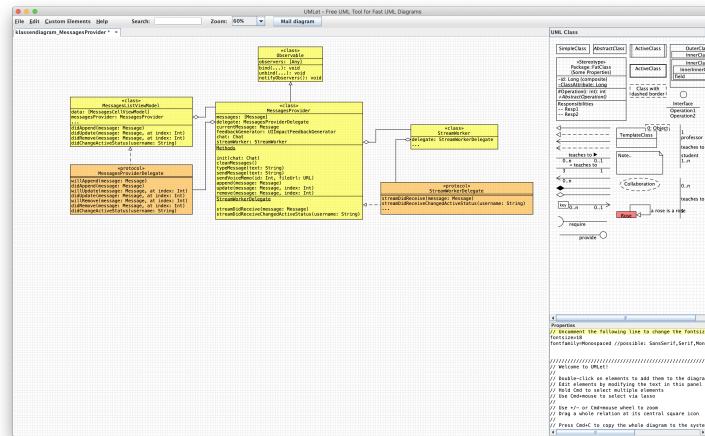


Abbildung A.5: Eine Ansicht eines Klassendiagramms in UMLEt

friends <pre> id int(11) AI PK requester varchar(255) receiver varchar(255) date_created double state tinyint(1) </pre>	users <pre> id varchar(255) PK username varchar(255) AK password varchar(255) salt varchar(255) device_token varchar(255) NULL active_chat varchar(255) NULL </pre>
uploads <pre> message_id varchar(255) PK owner varchar(255) filename varchar(255) </pre>	

Abbildung A.6: Übersicht Tabellen „friends“, „users“ und „uploads“ der SQL-Datenbank im M2-Backend.

PK = Primary Key (Primärschlüssel)

AI = Auto Increment (automatische Inkrementierung der Id)

A.4 Tabellen

Liste der HTTP-Requests für Nutzerbezogene Anwendungsfälle in M2			
Route	Methode	Parameter	Beschreibung
login.php	GET	username password device_token	Meldet einen Nutzer an
register.php	POST	id username password device_token	Registriert einen Nutzer
update_device_token.php	POST	username device_token	Aktualisiert einen Geräte-Token
logout.php	GET	username password	Meldet einen Nutzer ab
delete_user.php	GET	id username password	Löscht einen Nutzer
check_username.php	GET	username	Validiert einen Nutzernamen

Tabelle A.1: Liste der HTTP-Requests für Nutzerbezogene Anwendungsfälle im M2-Backend

Liste der HTTP-Requests für Freundschaften im M2-Backend			
Route	Methode	Parameter	Beschreibung
send_friends_request.php	POST	requester receiver	Sendet eine Freundschaftsanfrage
get_friends_requests.php	GET	requester	Liste der Freundschaftsanfragen
answer_friends_request.php	POST	id state requester	Beantwortet Freundschaftsanfrage
delete_friends_request.php	POST	id	Löscht Freundschaftsanfrage

Tabelle A.2: Liste der HTTP-Requests für Freundschaften im M2-Backend

A Material

Liste der HTTP-Requests für Chatanfragen im M2-Backend			
Route	Methode	Parameter	Beschreibung
get_chats.php	POST	requester	Liste der Chats
set_active_chat.php	POST	requester receiver	Setzt aktiven Chat
get_active_chat.php	GET	username	Fragt aktiven Chat von Nutzer ab

Tabelle A.3: Liste der HTTP-Requests für Chatanfragen im M2-Backend

Liste der HTTP-Requests für Upload & Download der Sprachnachrichten im M2-Backend			
Route	Methode	Parameter	Beschreibung
upload_voice_memo.php	POST	owner message_id	Upload einer Sprachnachricht
download_voice_memo.php	POST	message_id	Download einer Sprachnachricht
get_voice_memo.php	GET	message_id	Informationen zu Sprachnachricht
delete_voice_memo.php	GET	message_id	Löschen einer Sprachnachricht

Tabelle A.4: Liste der HTTP-Requests für Upload & Download der Sprachnachrichten im M2-Backend

Liste der HTTP-Status-Codes im M2-Backend	
Code	Beschreibung
200	Erfolg
601	Ungültiger Nutzernname
602	Ungültiges Passwort
603	Nutzername existiert nicht
604	Passwort existiert nicht
605	Ein unerwarteter Fehler ist aufgetreten
606	Angefragter Nutzer existiert nicht
607	Fehler beim Senden einer Freundschaftsanfrage
608	Anfragender Nutzer existiert nicht
609	Anfrage existiert nicht
610	Upload fehlgeschlagen
611	Datei existiert bereits
612	Ungültige Id übermittelt
613	Ungültiger Nutzer übermittelt
614	Sprachnachricht existiert nicht

Tabelle A.5: Liste der HTTP-Status-Codes im M2-Backend

Liste der verwendeten Soundeffekte in M2	
Dateiname	Verwendung
message_editing.m4a	Eingetippter Buchstabe einer Nachricht
message.m4a	Absenden und Empfangen einer Nachricht
notification.m4a	Erhalten einer Benachrichtigung innerhalb der App
offline.m4a	Verlassen des Chats
online.m4a	Beitreten des Chats

Tabelle A.6: Liste der verwendeten Soundeffekte in M2

Liste der Stream-Befehle im M2-Backend			
Name	Parameter	Parameterbeschreibung	Beschreibung
LoginCmd	cmd username	Command-Id: „login“ Nutzername	Beitreten des Streams
LogoutCmd	cmd username	Command-Id: „logout“ Nutzername	Verlassen des Streams
TypingCmd	cmd i f t d m	Command-Id: „t“ id: Id der Nachricht fromUser: Absendernutzername toUser: Empfängernutzername didCancel: Ob abgebrochen message: Nachricht bis Buchstabe	Eintippen eines Buchstabens
MessageCmd	cmd from_user to_user message messageID sentDate	Command-Id: „message“ Absendernutzername Empfängernutzername Fertige Nachricht Id der Nachricht Sendedatum	Versenden einer Textnachricht
VoiceMemoCmd	cmd from_user to_user messageID sentDate	Command-Id: „voicememo“ Absendernutzername Empfängernutzername Id der Nachricht Sendedatum	Versenden einer Sprachnachricht
ChangedActive-StatusCmd	cmd requester receiver isOnline	Command-Id: „changedActive“ Nutzer dessen Status geändert Nutzer der informiert wird Neuer Onlinestatus	Änderung des Onlinestatus
ChangedFriends-RequestsCmd	cmd requester receiver changedType	Command-Id: „changedFriends“ Nutzer der Anfrage verändert Nutzer von Änderung betroffen Grund der Änderung	Änderung der Freundschaftsanfragen

Tabelle A.7: Liste der Stream-Befehle im M2-Backend

Liste der verwendeten Bibliotheken in der iOS-App von M2		
Name	Verwendung	Quelle
UIKit	Implementierung der UI-Elemente	https://developer.apple.com/documentation/uikit
Foundation	Nutzung der grundlegenden Datentypen	https://developer.apple.com/documentation/foundation
AVFoundation	Nutzung der grundlegenden Datentypen	https://developer.apple.com/documentation/avfoundation
Speech	Spracherkennung für den Inhalt der Sprachnachrichten	https://developer.apple.com/documentation/speech
Alamofire	Alle netzwerkbasierten Aufrufe der HTTP-Endpunkte vom Backend	https://github.com/Alamofire/Alamofire
TinyConstraints	Vereinfachte Steuerung von Constraints	https://github.com/roberthein/TinyConstraints
XCGLogger	Verbessertes Logging	https://github.com/DaveWoodCom/XCGLogger
AppCenter	Verteilung der App außerhalb des App Stores	https://github.com/microsoft/appcenter
DifferenceKit	Verbesserte animierte Aktualisierung von Listeinträgen	https://github.com/ra1028/DifferenceKit
SwiftDate	Sammlung von Hilfsmethoden zur vereinfachten Arbeit mit datumsbasierten Datentypen	https://github.com/malcommac/SwiftDate
EmptyData-Set-Swift	Implementierung von Platzhaltern in Listenansichten	https://github.com/Xiaoye220/EmptyDataSet-Swift
NGSBadge-BarButton	Darstellung eines Badges in einer Navigationsleiste	https://github.com/PauliusVindzigelskis/NGSBadgeBarButton

Tabelle A.8: Liste der verwendeten Bibliotheken in der iOS-App von M2

Abbildungsverzeichnis

3.1	Nutzung von Social-Media-Angeboten (in Prozent), modifiziert	12
3.2	Empfangsansicht, wenn kein Nutzer angemeldet	17
3.3	Anmelde- und Registrierungsansicht	18
3.4	Leere Chatliste	19
3.5	Knopf zur Abmeldung	20
3.6	Hinweis vor Abmeldung	20
3.7	Öffnen der „Nutzer hinzufügen“-Ansicht	21
3.8	Bestätigung eines gefundenen Nutzernamens	21
3.9	Benachrichtigung einer Freundschaftsanfrage als Push-Benachrichtigung und innerhalb der App	22
3.10	Beantworten einer Freundschaftsanfrage	22
3.11	Push-Benachrichtigung beim Absender nach Bestätigung einer Freundschaftsanfrage	23
3.12	Chatlistenansicht bei bestehender Freundschaft Nutzer A und B . . .	23
3.13	Rückgängigmachen einer bestehenden Freundschaft	23
3.14	Öffnen der Nachrichtenansicht	24
3.15	Nachrichtenansicht, wenn gegenüberliegender Nutzer nicht verfügbar .	24
3.16	Push-Benachrichtigung bei gegenüberliegendem Nutzer nach Erhalt einer Nachrichtenanfrage	25
3.17	Nachrichtenansicht, wenn beide Nutzer verfügbar sind	25
3.18	Nachrichtenansicht für Nutzer A und B	26
3.19	Eintippen einer Nachricht von Nutzer A an B	26
3.20	Abgeschickte Textnachricht von Nutzer A an B	27
3.21	Abgeschickte Sprachnachricht von Nutzer A an B	28
3.22	Anhören einer Sprachnachricht	28
3.23	Transkription einer Sprachnachricht	29
3.24	Verlassen einer Konversation	29
3.25	Verlassen einer Konversation	29
4.1	Übersicht genutzter Schriftarten	31
4.2	Übersicht genutzter Farben	32
4.3	Übersicht der eigens gestalteten UI-Elemente (Teil 1)	32
4.4	Übersicht der eigens gestalteten UI-Elemente (Teil 2)	33
4.5	Apple Health und Apple App Store unter iOS 12	33
4.6	Entwicklungsphasen des Logos	34

Abbildungsverzeichnis

5.1	Visualisierung einer oftmals verwendeten Ausführung des Entwurfsmusters MVC	35
5.2	Visualisierung des Observer-Patterns	38
5.3	Ausgelöster Ablauf durch Nutzereingabe in BearbeitungsView mit Observer-Pattern	39
5.4	Visualisierung von MVC mit einem ViewController	40
5.5	Visualisierung des MVVM Entwurfsmusters	41
5.6	Visualisierung eines ViewModels für die BearbeitungsView	42
5.7	Sequenzdiagramm für die Nutzung des MVVM-Entwurfsmusters in M2	43
5.8	Klassendiagramm für die Model-Klasse einer Nachricht	44
5.9	Anmeldeansicht in M2	45
5.10	Klassendiagramm für einen Entwurf von LoginViewController und LoginViewModel	46
5.11	Chatlistenansicht in M2	46
5.12	Klassendiagramm für einen Entwurf von ChatsViewController, ChatsListViewModel und ChatsCellViewModel	47
5.13	Klassendiagramm für einen Entwurf vom AudioRecordingWorker	48
5.14	Klassendiagramm für einen Entwurf von ChatsViewController, ChatsListViewModel und ChatsCellViewModel inklusive ChatsProvider	48
6.1	Visualisierung einer beispielhaften HTTP-Anfrage der App zum Backend	55
6.2	Beispielhafte Tabelle für Möbel im Raumplaner	59
7.1	Visualisierung des gesamten Ablaufs einer HTTP-Anfrage von der App zum Backend	60
7.2	Hierarchie der Darstellung eines ViewControllers unter iOS	61
7.3	Hierarchie von mehreren, auf dem rootViewController präsentierten ViewControllern	62
7.4	Hierarchie aller ViewController in M2	62
7.5	Sequenzdiagramm für Anmeldung oder Registrierung in M2	63
7.6	LoginViewController, initialisiert mit SignInIntroViewModel oder CreateAccountIntroViewModel	63
7.7	Sequenzdiagramm für die Anmeldung in M2	64
7.8	Beispielhaftes Klassendiagramm für die Implementierung einer UITableView inklusive UITableViewDataSource	64
7.9	Klassendiagramm für die Implementierung von DifferenceKit in M2	66
7.10	Klassendiagramm für die Implementierung des BackendWorkers	67
7.11	Klassendiagramm für die Implementierung des FriendsRequestsEndpoints	67
7.12	Klassendiagramm des StreamWorkers	68
7.13	Klassendiagramm für das Protokoll OutgoingStreamable	69
7.14	Klassendiagramm für das Protokoll IncomingStreamable	69
7.15	AddUserViewController bei gefundenem Nutzer	70

Abbildungsverzeichnis

7.16 Sequenzdiagramm für eine Texteingabe im AddUserController	70
7.17 Sequenzdiagramm für das erfolgreiche Absenden einer Freundschaftsanfrage mit dem AddUserController	71
7.18 Badge in der UINavigationBar des ChatsViewControllers und Freundschaftsanfrage in der Freundschaftsanfragenliste	71
7.19 Sequenzdiagramm zum Ablauf des Chat-Startes	73
7.20 Klassendiagramm für den Aufbau des Chat-Models	73
7.21 Aufbau der Tabelle users im Backend	74
7.22 Standardmäßige Anordnung der UITableView	75
7.23 Bodenorientierte Anordnung der UITableView	75
7.24 Standardmäßige Darstellung der UITableView mit invertierter Nachrichtenliste in UITableViewDataSource	76
7.25 Um 180° gedrehte UITableView	77
7.26 Um 180° gedrehte UITableView mit um 180° gedrehten UITableViewCells	77
7.27 Klassendiagramm des MessagesProviders und seiner Abhängigkeiten .	78
7.28 Sequenzdiagramm eines eingetippten Buchstabens einer Nachricht im MessagesViewController	79
7.29 Enum zur Repräsentierung eines Soundeffektes	81
7.30 Nachrichtenverlauf mehrerer eingetippter Buchstaben	83
A.1 Eine Ansicht des Xcode-Projektes von M2	88
A.2 Eine Ansicht des IntelliJ-Projektes vom M2-Backend	88
A.3 Eine Ansicht der HTTP-Endpunkte von M2 in Paw	89
A.4 Eine Ansicht der konzipierten Benutzeroberflächen in Sketch	89
A.5 Eine Ansicht eines Klassendiagramms in UMLet	90
A.6 Übersicht Tabellen „friends“, „users“ und „uploads“ der SQL-Datenbank im M2-Backend	90

Tabellenverzeichnis

3.1	Gängige Features von Messengern auf mobilen Plattformen	13
3.2	Vergleich gängiger Features von Messengern auf mobilen Plattformen	14
3.3	Markt und Wettbewerb	16
5.1	Liste der ViewController in M2	45
5.2	Liste der Worker-Klassen in M2	47
A.1	Liste der HTTP-Requests für Nutzerbezogene Anwendungsfälle im M2-Backend	91
A.2	Liste der HTTP-Requests für Freundschaften im M2-Backend	91
A.3	Liste der HTTP-Requests für Chatanfragen im M2-Backend	92
A.4	Liste der HTTP-Requests für Upload & Download der Sprachnachrichten im M2-Backend	92
A.5	Liste der HTTP-Status-Codes im M2-Backend	93
A.6	Liste der verwendeten Soundeffekte in M2	93
A.7	Liste der Stream-Befehle im M2-Backend	94
A.8	Liste der verwendeten Bibliotheken in der iOS-App von M2	95

Quellcodeverzeichnis

6.1	Beispielhaftes JSON-Objekt Stuhl	57
6.2	Liste von JSON-Objekten	58
6.3	Beispielhafter SQL-Befehl zur Abfrage der gesamten Tabelle	58
6.4	SQL-Befehl zur Abfrage einer einzelnen Spalte mit einer Bedingung .	59
7.1	Codebeispiel UITableViewDataSource	65
7.2	Funktionskopf für das Absenden eines HTTP-Requests im Backend-Worker	68
7.3	Funktionskopf von answerFriendsRequest	72
7.4	Funktionskopf von Funktion aus UITableViewDelegate	72

Literaturverzeichnis

Alamofire: *Alamofire*, <https://github.com/Alamofire/Alamofire>, 2019, letzter Zugriff: 08.08.2019

Andersson, Jimmy M: *iOS Development and the Wrong Kind of MVC*, <https://medium.com/@JimmyMAndersson/ios-development-and-the-wrong-kind-of-mvc-4e3e2decb82e>, 2018, letzter Zugriff: 06.08.2019

Apple: *About Swift*, <https://swift.org/about/>, 2019, letzter Zugriff: 09.08.2019

Apple: *Accessibility on iOS*, <https://developer.apple.com/accessibility/ios/>, 2019, letzter Zugriff: 04.08.2019

Apple: *App Store*, <https://www.apple.com/de/ios/app-store/>, 2019, letzter Zugriff: 19.08.2019

Apple: *Apple erfindet mit dem iPhone das Mobiltelefon neu*, <https://www.apple.com/de/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/>, 2007, letzter Zugriff: 29.07.2019

Apple: *AVFoundation Documentation*, <https://developer.apple.com/documentation/avfoundation>, 2019, letzter Zugriff: 08.08.2019

Apple: *Bring Your iPad App to Mac*, <https://developer.apple.com/ipad-apps-for-mac/>, 2019, letzter Zugriff: 08.08.2019

Apple: *Cocoa (Touch)*, <https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html>, 2018, letzter Zugriff: 08.08.2019

Apple: *Foundation Documentation*, <https://developer.apple.com/documentation/foundation>, 2019, letzter Zugriff: 08.08.2019

Apple: *Foundation Documentation: InputStream*, <https://developer.apple.com/documentation/foundation/inputstream>, 2019, letzter Zugriff: 17.08.2019

Apple: *Foundation Documentation: JSONDecoder*, <https://developer.apple.com/documentation/foundation/jsondecoder>, 2019, letzter Zugriff: 18.08.2019

Apple: *Foundation Documentation: UserDefaults*, <https://developer.apple.com/documentation/foundation/userdefaults>, 2019, letzter Zugriff: 18.08.2019

Literaturverzeichnis

Apple: *Gesundheit*, <https://www.apple.com/de/ios/health/>, 2019, letzter Zugriff: 19.08.2019

Apple: *iOS 12*, <https://www.apple.com/de/ios/ios-12/>, 2019, letzter Zugriff: 06.08.2019

Apple: *iOS 13*, https://www.apple.com/de/ios/ios-13-preview/?&mtd=20925qby39952&aosid=p238&mnid=sz0Qixexq-dc_mtid_20925qby39952_pcrid_359930221153_cid=wwa-de-kwgo-iphone-slid--ipurl-ios+13-e-productid-, 2019, letzter Zugriff: 08.08.2019

Apple: *Human Interface Guidelines*, <https://developer.apple.com/design/human-interface-guidelines/ios>, 2019, letzter Zugriff: 29.07.2019

Apple: *NSURLSession*, <https://developer.apple.com/documentation/foundation/nsurlsession>, 2019, letzter Zugriff: 08.08.2019

Apple: *Foundation Documentation: OutputStream*, <https://developer.apple.com/documentation/foundation/outputstream>, 2019, letzter Zugriff: 17.08.2019

Apple: *Security Documentation: Keychains*, https://developer.apple.com/documentation/security/keychain_services/keychains, 2019, letzter Zugriff: 18.08.2019

Apple: *Speech Documentation*, <https://developer.apple.com/documentation/speech>, 2019, letzter Zugriff: 08.08.2019

Apple: *Speech Documentation: SFSpeechRecognizer*, <https://developer.apple.com/documentation/speech/sfspeechrecognizer>, 2019, letzter Zugriff: 18.08.2019

Apple: *UIKit Documentation*, <https://developer.apple.com/documentation/uikit>, 2019, letzter Zugriff: 08.08.2019

Apple: *UIKit Documentation: CGAffineTransform*, <https://developer.apple.com/documentation/coregraphics/cgaffinetransform>, 2019, letzter Zugriff: 17.08.2019

Apple: *UIKit Documentation: Combine*, <https://developer.apple.com/documentation/combine>, 2019, letzter Zugriff: 08.08.2019

Apple: *UIKit Documentation: Core Graphics*, <https://developer.apple.com/documentation/coregraphics>, 2019, letzter Zugriff: 17.08.2019

Apple: *UIKit Documentation: UICollectionView*, <https://developer.apple.com/documentation/uikit/uicollectionview>, 2019, letzter Zugriff: 15.08.2019

Literaturverzeichnis

- Apple: *UIKit Documentation: UIViewController*, <https://developer.apple.com/documentation/uikit/uiviewcontroller>, 2019, letzter Zugriff: 06.08.2019
- Apple: *UIKit Documentation: UITableView*, <https://developer.apple.com/documentation/uikit/uitableview>, 2019, letzter Zugriff: 15.08.2019
- Apple: *UIKit Documentation: UITableViewDataSource*, <https://developer.apple.com/documentation/uikit/uitableviewdatasource>, 2019, letzter Zugriff: 15.08.2019
- Apple: *UIKit Documentation: UITableViewDelegate*, <https://developer.apple.com/documentation/uikit/uitableviewdelegate>, 2019, letzter Zugriff: 17.08.2019
- Apple: *UIKit Documentation: UITableViewDiffableDataSource*, <https://developer.apple.com/documentation/uikit/uitableviewdiffabledatasource>, 2019, letzter Zugriff: 15.08.2019
- Apple: *The View Controller Hierarchy*, <https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/TheViewControllerHierarchy.html>, 2018, letzter Zugriff: 15.08.2019
- Apple: *WWDC 2017 - Session 204 - iOS: Updating Your App for iOS 11*, <https://developer.apple.com/videos/play/wwdc2017/204/?time=1402>, 2017, letzter Zugriff: 25. 07. 2019
- Apple: *Xcode*, <https://developer.apple.com/xcode/>, 2019, letzter Zugriff: 08.08.2019
- Bitkom Research: *Smartphone-Markt: Konjunktur und Trends*, <https://www.bitkom.org/sites/default/files/file/import/Bitkom-Pressekonferenz-Smartphone-Markt-22-02-2018-Praesentation-final.pdf>, 2018, letzter Zugriff: 24. 07. 2019
- Carthage: *Carthage*, <https://github.com/Carthage/Carthage>, 2019, letzter Zugriff: 08.08.2019
- Chamberlin, Don: *Encyclopedia of Database Systems*, https://doi.org/10.1007/978-1-4899-7993-3_1091-2, 2017, letzter Zugriff: 12.08.2019
- CocoaPods: *CocoaPods*, <https://cocoapods.org>, 2019, letzter Zugriff: 08.08.2019
- Eilebrecht, Karl & Starke, Gernot: *Patterns kompakt*, <http://dx.doi.org/10.1007/978-3-662-57937-4>, 2019, letzter Zugriff: 14.08.2019
- Electronic Code of Federal Regulations: *REAL-TIME TEXT*, <https://www.ecfr.gov/cgi-bin/text-idx?SID=ada7cebd65433a9e05c35426a2bc76b8&mc=true&node=pt47.3.67&rgn=div5>, 2019, letzter Zugriff: 18.08.2019

Literaturverzeichnis

Faktenkontor: *Social-Media-Atlas 2017/2018: Wo sich welche Altersgruppe im Web 2.0 herumtreibt*, <https://www.faktenkontor.de/corporate-social-media-blog-faktzweinull/soziale-medien-fuer-gross-und-klein/>, 2018, letzter Zugriff: 22. 07. 2019

Google: *Firebase pricing plans*, <https://firebase.google.com/pricing>, 2019, letzter Zugriff: 08.08.2019

Fowler, Martin: *Presentation Model*, <https://martinfowler.com/eaaDev/PresentationModel.html>, 2004, letzter Zugriff: 07.08.2019

Google: *Android*, https://www.android.com/intl/de_de/, 2019, letzter Zugriff: 06.08.2019

Grimm, Rüdiger & Delfmann, Patrick: *Digitale Kommunikation*, 2. Aufl., De Gruyter 2017

Grossman, John: *Introduction to Model/View/ViewModel pattern for building WPF apps*, <https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewmodel-pattern-for-building-wpf-apps/>, 2005, letzter Zugriff: 07.08.2019

Icons8: *Free design resources and software*, <https://icons8.com>, 2019, letzter Zugriff: 19.08.2019

Information Sciences Institute: *Request For Comments (RFC) 793: Transmission Control Protocol*, <https://tools.ietf.org/html/rfc793>, 1981, letzter Zugriff: 17.08.2019

Isaac, Mike: *Zuckerberg Plans to Integrate WhatsApp, Instagram and Facebook Messenger*, <https://www.nytimes.com/2019/01/25/technology/facebook-instagram-whatsapp-messenger.html>, 2019, letzter Zugriff: 26. 07. 2019

JetBrains: *IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains*, <https://www.jetbrains.com/idea/>, 2019, letzter Zugriff: 10.08.2019

Katoch, Manish: *Swift + MVVM + Two Way Binding = Win!*, <https://codeburst.io/swift-mvvm-two-way-binding-win-b447edc55ff5>, 2017, letzter Zugriff: 08.08.2019

Kessler, Florence, Runkehl, Jens (Hrsg.), Schlobinski, Peter (Hrsg.), Siever, Torsten (Hrsg.): *Instant Messaging. Eine neue inter- personale Kommunikationsform*, <https://www.repo.uni-hannover.de/bitstream/handle/123456789/2973/networkx-52.pdf?sequence=1&isAllowed=y>, 2008, letzter Zugriff: 29. 07. 2019

Literaturverzeichnis

Kühnel, Andreas: *Visual C# 2012*, http://openbook.rheinwerk-verlag.de/visual_csharp_2012/1997_28_005.html, 2012, letzter Zugriff: 14.08.2019

Lackes, Prof. Dr. Richard & Siepermann, Dr. Markus: *Grundlagen der Wirtschaftsinformatik*, <https://wirtschaftslexikon.gabler.de/definition/bottom-prinzip-27383>, 2019, letzter Zugriff: 05.08.2019

Lahres, Bernhard & Rayman, Gregor: *Objektorientierte Programmierung*, http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_08_002.htm, 2009, letzter Zugriff: 05.08.2019

Lattner, Chris: *Chris Lattner's Homepage*, <http://nondot.org/sabre/>, 2019, letzter Zugriff: 09.08.2019

Microsoft: *Most Valuable Professional*, <http://mvp.microsoft.com>, 2019, letzter Zugriff: 07.08.2019

Müller, Prof. Dr. Günter: *Methoden und Techniken der Organisationsgestaltung*, <https://wirtschaftslexikon.gabler.de/definition/top-down-prinzip-49846#references>, 2019, letzter Zugriff: 05.08.2019

Paw: *Paw - The most advanced API tool for Mac*, <https://paw.cloud>, 2019, letzter Zugriff: 10.08.2019

Postel, J.: *Request For Comments (RFC) 768: User Datagram Protocol*, <https://tools.ietf.org/html/rfc768>, 1980, letzter Zugriff: 17.08.2019

Realm: *Jazzy*, <https://github.com/realm/jazzy>, 2019, letzter Zugriff: 19.08.2019

Reuters: *Facebook to buy WhatsApp for \$19 billion*, <https://www.reuters.com/article/us-whatsapp-facebook/facebook-to-buy-whatsapp-for-19-billion-idUSBREA1I26B20140219>, 2014, letzter Zugriff: 26.07.2019

RxSwift: *RxSwift: ReactiveX for Swift*, <https://github.com/ReactiveX/RxSwift>, 2019, letzter Zugriff: 19.08.2019

Sillmann, Thomas: *Das Swift-Handbuch*, Carl Hanser Verlag 2019

Sketch: *The best products start with Sketch*, <https://www.sketch.com>, 2019, letzter Zugriff: 21.08.2019

Ullenboom, Christian: *Java ist auch eine Insel*, http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_10_002.htm, 2011, letzter Zugriff: 06.08.2019

UMLet: *Free UML Tool for Fast UML Diagrams*, <https://www.umlet.com>, 2019, letzter Zugriff: 21.08.2019

Literaturverzeichnis

Wikipedia: *Firebase*, <https://de.wikipedia.org/wiki/Firebase>, 2019, letzter Zugriff: 08.08.2019

Wikipedia: *JavaScript Object Notation*, https://de.wikipedia.org/wiki/JavaScript_Object_Notation, 2019, letzter Zugriff: 10.08.2019

Wikipedia: *Model View Controller*, https://de.wikipedia.org/wiki/Model_View_Controller, 2019, letzter Zugriff: 05.08.2019

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum

Vincent Friedrich