HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

RedLine. Software

Neukonzeption einer benutzerfreundlichen Smartphone-App zur Unterstützung der Betreuungsarbeit in Heimen

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2020

Autoren	Timo Wetzel, Martin J. Sauter, David Zwick
Betreuer	Prof. Dr. Olaf Zimmermann
Projektpartner	RedLine Software GmbH, St. Gallen
Experte	Dr. Hans-Peter Hoidn
Gegenleser	Dr. Thomas Bocek

Abstract

Die RedLine Software GmbH bietet eine Softwarelösung, um alle administrativen Aufgaben in sozialpädagogischen Institutionen zu erleichtern. Im Fokus von RedLine stehen der Benutzer und seine Arbeit mit den Klienten, die Zusammenarbeit sowie die Prozesse (medizinisch und administrativ) im Kontext der Institution, sowie die Berücksichtigung des gesamten Umfelds. Die RedLine Lösung steht als Software-as-a-Service (SaaS) zur Verfügung und wird über einen Browser aufgerufen. Das Ziel dieser Arbeit ist, in Zusammenarbeit mit Institutionen, eine Smartphone- sowie Tablet-App der zweiten Generation zu konzipieren und zu realisieren, um RedLine sinnvoll mobil einsetzen zu können.

Zu Beginn der Arbeit existierte bereits eine mobile Version von RedLine; diese hatte jedoch einen limitierten Funktionsumfang und einige Schwächen bezüglich Bedienbarkeit. Als erster Schritt befragten wir sozialpädagogische Institutionen, welche Funktionen für sie am interessantesten sind. Im zweiten Schritt modernisierten wir die bestehende App und erweiterten sie mit neuen Funktionen. Da Sicherheit und Datenschutz sowohl von RedLine als auch von den Institutionen sehr wichtig eingestuft werden, mussten wir von Anfang an unser Augenmerk darauf legen, die resultierenden Qualitätsanforderungen zu erfüllen. In einem dritten Schritt verbesserten wir die Qualität und Wartbarkeit der App durch automatisierte Testverfahren. Neben JUnit Unit-Tests schrieben wir Android Espresso Tests, welche das User Interface sowie auch die Funktionalität der App testeten. Um sicherzustellen, dass die App auf verschiedenen Android-Geräten mit unterschiedlichen Versionen funktioniert, verwendeten wir Browserstack, wobei die Tests jeweils auf fünf physischen Geräten durchgeführt wurden. Zum Ende der Arbeit holten wir ein Feedback von den RedLine Mitarbeitern zur neuen App ein, welches sehr positiv ausfiel. Leider konnte kein Feedback von den Institutionen eingeholt werden, da der Zugang auf Grund der Corona-Pandemie eingeschränkt war.

RedLine steht nun eine neue Android-App zur Verfügung. Die neue App bietet eine echte Alternative zur Browserversion ausserhalb des Büros. Durch die Umfragen und Interviews wurden Informationen zur Verwendung der App und der gesamten SaaS gesammelt. Nach der Auswertung und Priorisierung aller Interview- und Umfrageresultate wurden alle Funktionen, welche realisierbar waren, erfolgreich implementiert. Durch die Implementation eines clientseitigen Caches wurde die Performance verbessert; die Verwendung der neuesten Android-Verschlüsselungsalgorithmen erlaubt es offline zu arbeiten. Wenn in einem nächsten Schritt die iOS-Version der App und die serverseitige API von RedLine realisiert werden, kann die App veröffentlicht werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ausgangslage

RedLine bietet eine Software für Sozialpädagogische Institutionen, um die Administration zu erleichtern und zu optimieren. Im Fokus von RedLine stehen die Benutzer und seine Arbeit mit den Klienten, die Prozesse und Zusammenarbeit im Kontext der Institution, sowie die Berücksichtigung des gesamten Umfelds. Den Benutzern von RedLine sollen administrative Aufgaben erleichtert werden, damit diese so effizient wie möglich von statten gehen können. Bisher bietet RedLine dies mit einer WebApp, welche die Benutzer über einen beliebigen Browser aufrufen können. Die Software wird als SaaS (Software as a Service) angeboten. Auf RedLine hat man Einsicht in viele wichtige Informationen wie Klientendaten, Bereichsdaten oder Tagesjournale, welche Betreuer in ihrer täglichen Arbeit benötigen. Es existiert bereits eine Smartphone-App für Android und iOS, diese bietet jedoch nicht den selben Funktionsumfang wie die RedLine Software. Mit dieser App können Klientendaten abgerufen, das Tages- sowie Verlaufsjournal angezeigt und Notizen erstellt werden. Der grösste Vorteil der App ist bislang die Möglichkeit Fotos aufzunehmen. Die bisherige App ist zwar funktionsfähig, jedoch mit begrenztem Funktionsumfang.

1.2 Problemstellung

Das Ziel dieser Arbeit ist, eine Smartphone-App zu erstellen, welche einen grossen Funktionsumfang sowie eine gute Performance und Sicherheit liefert. Die App soll den Betreuern in den verschiedenen Institutionen die Arbeit erleichtern, dies tut sie nur, wenn sie genügend sowie auch die richtigen Funktionen hat, einfach zu bedienen ist und stabil arbeitet. Die App soll auf einen akzeptablen technologischen Stand gebracht werden, welcher Bereiche wie Performance, Sicherheit und Wart- sowie Erweiterbarkeit umfasst. Die neuen Funktionen der App sollen zuerst mittels Umfragen und Interviews mit Benutzern der alten App ermittelt werden. Anschliessend sollen die Resultate zusammengefasst und analysiert werden, um herauszufinden, welche Funktionen von den Benutzern gewünscht sind. Anhand dieser Daten soll anschliessend entschieden werden, welche Funktionen effektiv in die App integriert werden. Diese Arbeit umfasst nur die Android App, die iOS Version wird gegebenenfalls zu einem späteren Zeitpunkt ergänzt, jedoch nicht im Rahmen dieser Arbeit.

1.3 RedLine App

Die App soll Betreuern in Heimen und ähnlichen Einrichtungen helfen, die Arbeit einfacher und effizienter durchzuführen. In solchen Einrichtungen befindet sich meistens ein Computer auf der Station. In den Zimmern bei den Klienten besteht somit kein Zugriff auf die Daten. Durch die App wird diese Lücke geschlossen und die Betreuer können an allen Standorten, bei den Klienten selber oder auf dem Weg zu ihnen, auf die wichtigen Daten zugreifen und Daten ergänzen oder ändern. Dies ist beispielsweise bei der Medikamentenvergabe von Vorteil, wo keine Fehler passieren dürfen. Mit der App kann ein Foto von jedem Medikament erstellt werden, welches bei der Vergabe angezeigt wird, um die Verwechslung von Medikamenten zu verhindern. Des Weiteren wird die Menge der Medikamente direkt angezeigt und somit werden weiteren Fehlern vorgebeugt.

Da die Klientendaten vor Ort verfügbar sind, kann auch das Befinden des Klienten direkt festgehalten werden. Allfällige Notizen können direkt mit der App getätigt werden, bevor sie vergessen oder verloren gehen.
Ein wichtiger Punkt der Arbeit ist Datenschutz und Sicherheit. Jegliche Kommunikation zwischen den Geräten und der Datenbank findet verschlüsselt statt. Dies garantiert einen abhörsicheren und unverfälschten Nachrichtenaustausch. Aus Datenschutzgründen werden keine Daten auf dem Gerät gespeichert. Alle institutionsbezogenen Daten wie Klienten, Medikamente oder Journale werden von der Datenbank geholt und nicht zwischengespeichert. Dies ist insofern notwendig, da die App in vielen Fällen auf den privaten Geräten der Betreuer installiert ist und verwendet wird. Um Zugriff durch Unbefugte auf die App zu verhindern, wird eine Zweifaktorauthentifizierung angewendet. Bei einem Verlust des Passwortes ist somit kein unbefugter Zugriff auf die Daten durch Dritte möglich.

Kapitel 2

Anforderungen

2.1 Umfragen mit Institutionen

Ein Teil der Aufgabenstellung war das Erheben der Anforderungen durch den Kontakt mit Institutionen, welche die bisherige App im Alltag einsetzen oder Interesse daran haben. Zusätzlich wollten wir die überarbeitete App ausgewählten Institutionen präsentieren und ein Feedback dazu erhalten.

2.1.1 Verwendungsorte der App

Der häufigste Verwendungsort der App ist ganz klar bei Besuchen der Betreuer in den Wohnungen der Klienten. Viele Institutionen haben auf der Station einen Computer mit Zugriff auf RedLine Web, in den Wohnungen oder Zimmern der Klienten jedoch nicht. Somit gibt es dort keinen Zugriff auf das RedLine System und diese Lücke soll die App schliessen. Der zweithäufigste Verwendungsort ist bei Spital- oder Arztbesuchen mit den Klienten. Dabei wäre das Ziel, dass die Betreuer dem Arzt Symptome oder auffälliges Verhalten aus der App heraus vorlesen kann, damit dieser direkt auf alle medizinischen und gesundheitlichen Informationen des Klienten Zugriff hat und eine bessere Diagnose stellen kann. Der dritthäufigst genannte Anwendungsort ist innerhalb der Station selber. Wie anfangs bereits erwähnt, haben die meisten Institutionen nur einen Computer auf der Station, in den Zimmern der Klienten ist kein Zugang zu RedLine vorhanden. Durch die Verwendungsorte lassen sich einige Anforderungen an die App herauslesen, sowohl funktionale als auch nicht-funktionale.

2.1.2 Anforderungsanalyse

Um die Anforderungen an die RedLine App zu bestimmen, wurden Umfragen mit den bisherigen Benutzern der App durchgeführt. Ziel dieser Umfragen war es, herauszufinden welche Funktionen der bisherigen App verwendet werden, welche neuen Funktionen gewünscht sind und was für generelle Verbesserungen anfallen. Gewisse Anforderungen wurden auch von der Firma RedLine selber gestellt, die meisten jedoch stammen von den Benutzern der App.
Die Umfragen wurden in drei verschiedenen Formen durchgeführt: Per Telefongespräch, über einen Fragebogen per Mail oder in einem persönlichen Gespräch. Ursprünglich war geplant, die meisten Umfragen im persönlichen Gespräch mit einigen bisherigen Benutzern der App durchzuführen. Dies wurde jedoch durch unvorhersehbare Umstände zum grössten Teil verhindert. Durch die Corona-Pandemie war es uns nicht möglich, alle Institutionen zu besuchen, da deren Klienten oft Teil der Risikogruppe waren. Somit mussten einige Umfragen über ein Telefongespräch durchgeführt werden, was jedoch keinen negativen Einfluss auf die Qualität der Antworten hatte. Gewisse Institutionen hatten ein direktes Interview von Anfang an abgelehnt, erklärten sich jedoch dazu bereit, einen Fragebogen auszufüllen. Der Fragebogen sowie unsere Leitfragen der Interviews sind im Anhang A Abschnitt I. Die Resultate der einzelnen Institutionen sind aus Datenschutzgründen nicht in diesem Bericht enthalten. Jedoch können wir über die zusammengefasste Auswertung aller Institutionen berichten, aus welchen die funktionalen sowie die nicht-funktionalen Anforderungen abgeleitet wurden.

Funktionen

In Tabelle 2.1 finden sich die wichtigsten Funktionen, welche die alte App bietet und die neue App bieten soll. Einige dieser Funktionen waren bereits in der ersten Version der App vorhanden, andere wurden von den Benutzern gewünscht.

Bestehende Funktionen	**Gewünschte Funktionen**
Stammdaten der Klienten anschauen	Zusätzlich Notfallkontakte anschauen und direkt anrufen können
Journale anschauen, erstellen und bearbeiten	Präsenzkontrolle führen
Fotos erstellen und zu Klienten, Medikamenten oder Journalen hinzufügen	Regelmässige und Reserve Medikation eintragen
	Medikationsgeschichte/-verlauf anschauen

Tabelle 2.1: Die wichtigsten Funktionen der App

Sicherheit

Die Fragen zur Sicherheit des Apps lieferten sehr eindeutige Resultate: Die Sicherheit der Daten ist sehr wichtig, jedoch muss die App auch praktikabel sein. Eine Zweifaktorauthentifizierung ist somit erwünscht, sie darf jedoch nicht umständlich sein und einen langen Login-Prozess zur Folge haben. Ein weiterer Punkt ist die Angst, dass Unbefugte die Daten über die Schulter des Benutzers lesen, beispielsweise im Zug. Die meisten Institutionen wollen jedoch die Verwendung der App in der Freizeit unterbinden. Ebenso wurde ein automatischer Logout aus der App gewünscht, wenn man das Gerät sperrt oder die App wechselt. Eine Zweifaktorauthentifizierung ist ebenfalls von RedLine selbst erwünscht, da dies den Zugriff durch Unbefugte erschwert.

Generelles

Die Umfragen haben auch einige generelle Informationen zur App hervorgebracht, welche während dieser Arbeit, aber auch bei der späteren Pflege und Weiterentwicklung, von grossem Wert sind. Somit war Anfangs geplant, die App nicht für Tablets zu optimieren. Die Umfragen ergaben jedoch, dass viele Institutionen die App auf Tablets verwenden, da diese Infrastruktur bereits vorhanden ist und damit die Betreuer nicht ihre privaten Geräte verwenden müssen. Somit wurde die nicht-funktionale Anforderung für die Tablet Optimierung hinzugefügt. Ein weiterer Punkt war der Zugriff auf die Daten. Alle Betreuer benötigen Zugriff auf alle Klienten, nicht nur auf die, welche ihnen zugeteilt sind. Sie verwenden den Verlaufsbericht, um zu erfahren, was ihr Vorgänger gemacht hat und bei Arzt- oder Spitalbesuchen. Einzelne verwenden aktuell die normale RedLine Software auf dem Smartphone-Browser. Diese Version ist jedoch nicht für Mobilgeräte optimiert. Die meisten wünschen sich, dass die App dies jedoch ist und sich gut mit einem Touchscreen bedienen lässt. Eine der wichtigsten Funktionen der mobilen Version von RedLine ist die Möglichkeit, Bilder aufzunehmen und hochzuladen. Dies wird für die Klientenbegleitung, für Krankheitssymptome oder Verletzungen sowie für Journaleinträge gewünscht, jedoch nicht für Medikamente.

Abgeleitete Funktionen

Einige Funktionen wurden nicht direkt von den Institutionen und Betreuern gewünscht, werden jedoch von diesen Wünschen impliziert. Beispielsweise wurde ein Offlinemodus von einigen gewünscht, von anderen jedoch nicht. Dies bedeutet, dass dieser konfigurierbar sein muss, was auf einen Einstellungsbildschirm hinweist. Des Weiteren wurde die Möglichkeit gewünscht, Journaleinträge bearbeiten zu können, um kleinere Fehler zu beheben. Dies kann jedoch zu Konflikten führen, wenn derselbe Eintrag währenddessen von jemandem anderen geändert wurde oder wenn die Änderung im Offlinemodus stattfindet. Dies führte zur Notwendigkeit eines Konfliktbehebungsbildschirm.

2.2 Funktionale Anforderungen

Die funktionalen Anforderungen, welche grösstenteils aus den Umfrageresultaten stammen, wurden in User Stories beschrieben und in einer Story Map priorisiert sowie gegliedert. Die insgesamt 23 User Stories wurden in drei Kategorien aufgeteilt, welche ihre Wichtigkeit anhand der Umfrageresultate wiederspiegelt.

- Must Have: Funktionen, welche zwingend benötigt werden, diese müssen implementiert werden.
- Important: Funktionen, welche wichtig sind und wenn möglich auch implementiert werden
- Nice to Have: Funktionen, welche erwähnt wurden, jedoch für den Erfolg der App nicht zwingend sind

Die User Stories finden sich im Anhang A Abschnitt II. Die Story Map dient der Übersicht über alle Funktionen, welche die App im fertigen Zustand bieten soll. Auch hier wurde dieselbe Priorisierung verwendet, wie bei den User Stories. In der horizontalen impliziert die Story Map einen Ablauf einer User Story, begonnen mit dem Login und anschliessend der gewünschten Funktion, beispielsweise dem Aufrufen eines Klienten und dessen Gesundheitsdaten. In der obersten Spalte der Storymap sind die Kategorien aufgelistet, mit denen gearbeitet wird: Klienten, Gesundheit, Medikamente oder Journale, aber auch App-spezifische Funktionen wie der Login oder die Einstellungen. In der zweiten Zeile finden sich dann die Tätigkeiten oder Abläufe zu diesen Kategorien, beispielsweise, dass zuerst ein Klient über einen Filter ausgewählt wird damit anschliessend seine Details angezeigt werden können. Die spezifischen Funktionen folgen vertikal sortiert nach ihrer Wichtigkeit. Die Einträge in der Story Map sind mit Tags versehen. Diese Tags kennzeichnen, wie mit diesen Daten umgegangen werden darf, beziehungsweise welche Funktionen auf diesen Daten anwendbar sind. Die persönlichen Informationen dürfen beispielsweise nur gelesen werden und dürfen offline verfügbar sein. Die Medikationsinformationen dürfen zusätzlich noch bearbeitet und gelöscht werden.

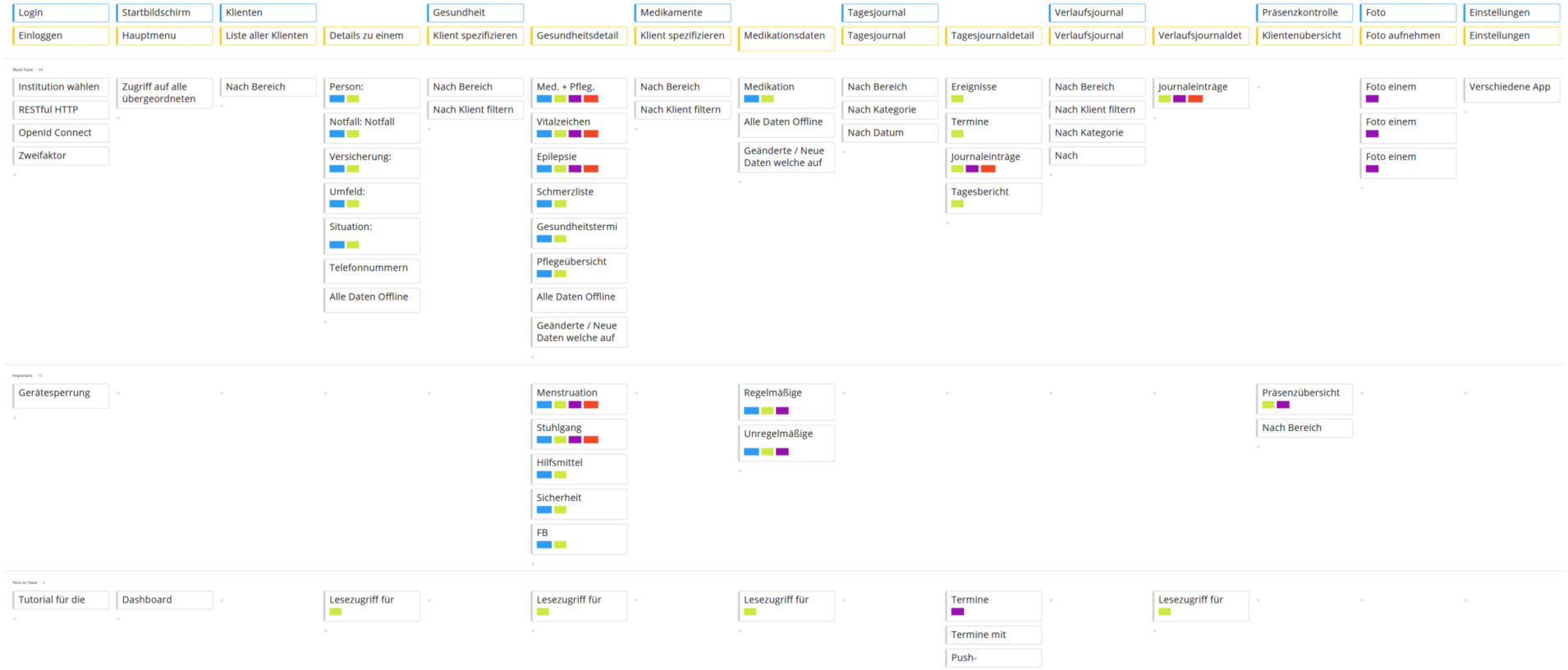


Abbildung 2.1: StoryMap mit Funktionen, abgeleitet aus den User Stories
Legende: Blau - Offline verfügbar / Grün - Lesen / Violett - Schreiben / Rot - Löschen

2.3 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen wurden nach dem SMART Prinzip [1] definiert, wobei der Fokus auf den beiden ersten Kategorien liegt, spezifisch und messbar. Die Anforderungen wurden in sechs Kategorien unterteilt: Performance, Usability, Security, Maintainability, Reliability sowie Portability. Diese sechs Kategorien wurden gewählt, weil sie die gängigsten sind und alle wichtigen Kategorien einer App abdecken. Die nicht-funktionalen Anforderungen finden sich im Anhang A Abschnitt III.

Entscheide

Dieses Kapitel umfasst Entscheide, welche zur Änderung von nicht-funktionalen Anforderungen geführt haben, sowie die Gründe für diese Entscheide.

Android Versionserhöhung

Diese Änderung betrifft die nicht-funktionale Anforderung P1 aus der Kategorie Performance, siehe Tabelle A.1. In dieser wurde festgelegt, dass die App auf Geräten mit der Android Version 5.0 und höher unterstützt werden. Diese Anforderung wurde während des Projektverlaufes auf Android Version 6.0 (API 23) erhöht, um die Verschlüsselung der Offlinedaten, sowie die Geräte-Authentifizierung zu ermöglichen. Ab API 23 (Android 6.0) wird die Crypto-Library von AndroidX unterstützt, was die Verschlüsselung erheblich vereinfacht und verbessert. Ebenfalls unterstützt diese API Version die Geräte Authentifizierung für Apps, womit der Pin-Code oder der Fingerabdruck abgefragt werden kann. Ein Nachteil der Erhöhung der Version ist die Geräteabdeckung. Diese verringert sich von ca. 94% auf ca. 85%, also 9% Verlust in der Geräteabdeckung (Stand 18.04.2020).

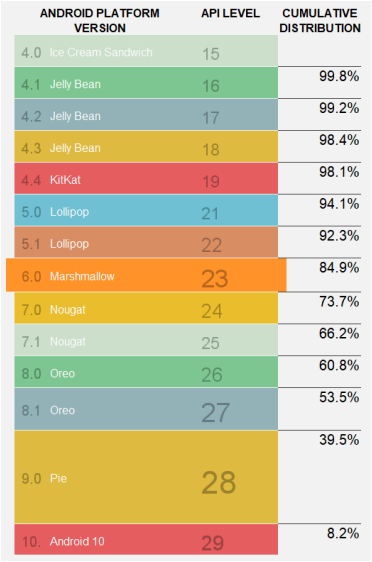


Abbildung 2.2: Auszug von Android Studio (Stand 18.04.2020)

Tablet Optimierung

Diese Änderung betrifft die nicht-funktionale Anforderung P2 aus der Kategorie Performance, siehe Tabelle A.1. Ursprünglich war geplant, die App nur für Smartphones zu entwickeln und keine User Interface Optimierungen für Tablets vorzunehmen. Mit dieser Änderung soll die App zusätzlich auch für Tablets optimiert werden. Dies wur-

de nach der Auswertung der Umfrageergebnisse hinzugefügt, da viele Institutionen bereits Tablets haben und sie nicht möchten, dass private Geräte verwendet werden.

Offlinemodus

Diese Änderung betrifft die nicht-funktionale Anforderung S3 aus der Kategorie Security, siehe Tabelle A.3. Die App darf keine klientenspezifischen oder sensiblen Daten in den nichtflüchtigen Speicher ablegen. Wenn die Daten verschlüsselt sind, dürfen sie abgelegt werden, ansonsten wäre kein Offlinemodus möglich.

Kapitel 3

Analyse Bestehende App

Im folgenden Kapitel wird der Stand der App vor der Bachelorarbeit beschrieben und analysiert. Die Analyse beschränkt sich dabei auf Funktionen der App sowie deren Umsetzung in der Softwarearchitektur.

3.1 Funktionen

Im Folgenden werden die Funktionen der App beschrieben und das User Interface dazu mit Bildern dargestellt. Bei den Informationen, welche auf den Bildern zu sehen sind, handelt es sich um Beispiele und nicht um echte Daten.

3.1.1 Login

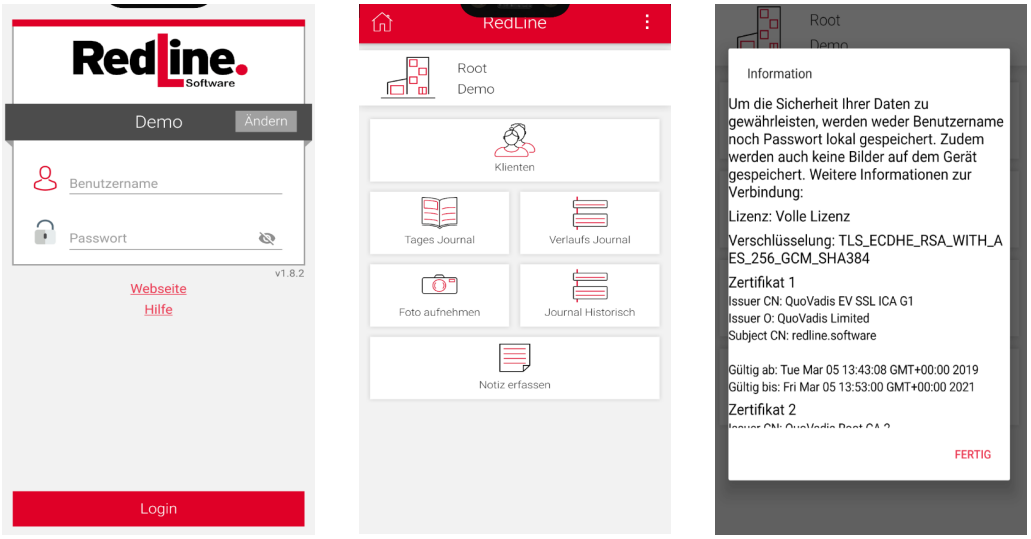
Auf dem Login Screen kann der Benutzer die Institution angeben, in welche er sich gerne einloggen möchte und anschliessend mithilfe seines Benutzernamen und Passwortes einloggen. Nach einigen Sekunden wird ihm mitgeteilt, ob der Login funktioniert hat oder nicht. Es gibt zusätzlich noch Hyperlinks zur RedLine-Webseite und zur Hilfe. Wenn die App Adminrechte hat, was auf einem "gerooteten" Gerät der Fall ist und somit ein Sicherheitsrisiko darstellt, wird eine Warnung angezeigt.

3.1.2 Homebildschirm

Auf dem Homebildschirm kann der Benutzer verschiedene Funktionen auswählen. Er kann des Weiteren über das Optionenmenü am oberen rechten Rand Informationen über die Verbindung zum Server anzeigen, darin wird der TLS Cypher sowie der Zertifikat-Pfad angezeigt.

Verbesserungsmöglichkeiten

Die Verbindungsinformationen sind nicht sehr übersichtlich gestaltet. Mehr Abstände und eine besser Textdarstellung würden die Lesbarkeit erhöhen.



(a) Loginbildschirm (b) Homebildschirm (c) Verbindungsinformationen

Abbildung 3.1: Die ersten drei Bildschirme

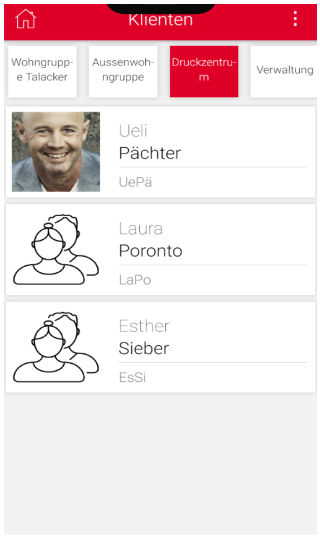
3.1.3 Klienteninformationen

Unter "Klienten" können die Informationen über einen bestimmten Klienten abgerufen werden. Dazu muss zuerst aus einer Liste von Bereichen und einer Liste von Klienten ein bestimmter Klient ausgewählt werden. Die Klienten werden mit vollem Namen und einem Profilbild dargestellt.

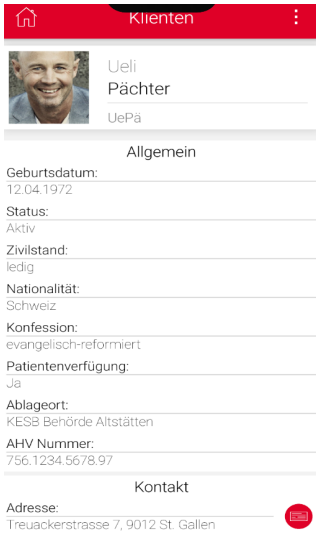
Nach dem Auswählen des Klienten können einige Informationen dieses Klienten untereinander in einer Liste angezeigt werden. Informationen, die mehr als eine Zeile benötigen, können durch anklicken als Popup angezeigt werden, damit alle Informationen ersichtlich sind.

Verbesserungsmöglichkeiten

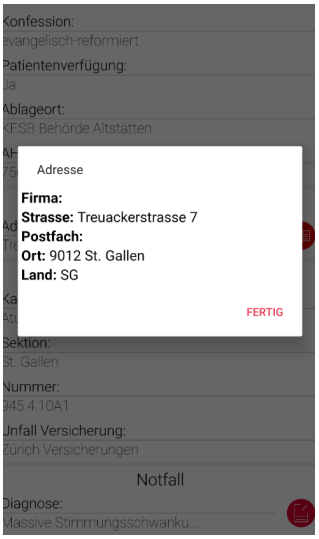
Die Klienteninformationen sind nicht vollständig vorhanden. Auf der RedLine Webseite gibt es noch viel mehr Informationen zum Klienten, welche in der App nicht sichtbar sind.



(a) Auswahl der Klienten



(b) Klienteninformationen



(c) Popup mit Adresse eines Klienten

Abbildung 3.2: Klienten Ansicht

3.1.4 Journal

Das Journal ist eine der Schlüsselfunktionen von RedLine. Es ermöglicht einem Betreuer sich einen Übersicht über den Stand einer Station zu verschaffen, Vorkommnisse zu dokumentieren und geplante Aktionen nachzuschauen. Es wird in der Webversion von RedLine in drei Teile unterteilt. Das Tagesjournal aggregiert Journaleinträge nach Bereichen, es soll den Betreuern helfen, eine Übersicht über einen ganzen Bereich zu erhalten. Das Verlaufsjournal in historischer Ansicht aggregiert Journaleinträge für einen bestimmten Klienten über eine Zeitspanne. Es soll einem Betreuer oder Aussenstehenden eine Übersicht über die vergangenen Tage eines Klienten verschaffen. Das Verlaufsjournal in kompakter Ansicht ist diesem sehr ähnlich, es fasst jedoch wiederholende Termine in einem zusammen. So soll eine kompakte Übersicht über die laufenden Termine eines Klienten darstellen.

Verbesserungsmöglichkeiten

Die Journale sind derzeit zu voll und zu bunt. Teilweise überlappen Elemente und Informationen, sodass diese nicht richtig gelesen werden können.

Tagesjournal

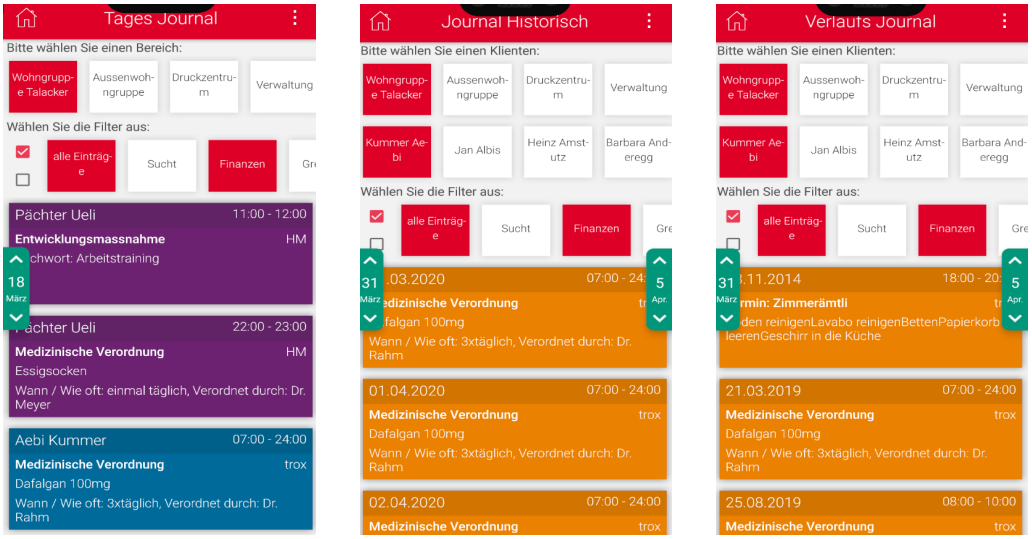
In der App muss der Benutzer einen Bereich und einen Filter setzen, um das Tagesjournal von diesem Tag anzuschauen. Er kann über eine Datumsauswahl an der Seite das Tagesjournal eines anderen Tages anzeigen. Der Filter ist mit Kategorien gefüllt, welche von der Institution vorgefertigt werden. Die Journaleinträge werden mit einer Farbcodierung versehen, welche die verschiedenen Klienten abgrenzen sollen. Journaleinträge können Bilder beinhalten, welche bei einem Klick auf den Eintrag angezeigt werden.

Verlaufsjournal Historisch

Der Aufbau des Verlaufsjournals ist dem Tagesjournal ähnlich. Es gibt jedoch einen zweiten Datumsregler, mit dem es möglich wird, einen Datumsbereich auszuwählen. Zudem gibt es noch eine weitere Auswahl, um den Klienten zu spezifizieren. Anstatt dem Namen des Klienten wird im Titel eines Eintrags das Datum angezeigt.

Verlaufsjournal Kompakt

Die kompakte Ansicht des Verlaufsjournals ist identisch zur historischen. Der einzige Unterschied liegt darin, dass weniger Einträge zu sehen sind, da gleiche Einträge zusammengelegt wurden.



(a) Tagesjournal (b) Verlaufsjournal historisch (c) Verlaufsjournal kompakt

Abbildung 3.3: Journale

3.1.5 Foto Funktion

Die Fotofunktion lässt den Benutzer ein Foto mit der Smartphone Kamera machen, um dieses dann für RedLine zu verwenden. Der Benutzer hat dabei drei Möglichkeiten: Ein Profilfoto eines Klienten, ein Foto eines Medikaments und einen Journaleintrag mit Foto zu erstellen. Da das Foto als sensible Datei eingestuft wird, muss die Kamera direkt aus der App gestartet werden und nicht wie oft üblich, über eine externe Kamera-App.

Verbesserungsmöglichkeiten

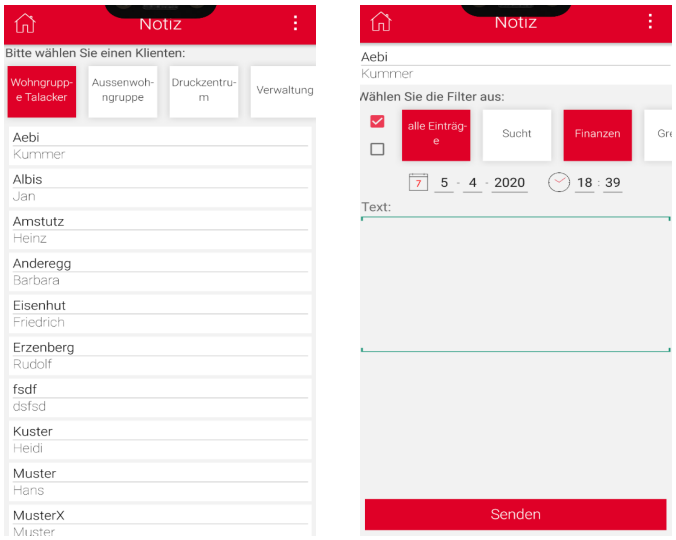
Die Fotofunktion wäre intuitiver, wenn man von anderen Screens aus direkt auf die Kamera gelangen könnte.

3.1.6 Notiz (Journaleintrag erfassen)

Journaleinträge können über eine Notiz erfasst werden. Dieser Eintrag wird dann im Tagesjournal unter dem Klienten angezeigt. Dazu wird als erstes, ähnlich wie bei den Klienteninformationen, der Klient aus einer Liste ausgewählt. In dieser Liste wird jedoch nur der Name ohne Foto angezeigt. Nach der Auswahl kann der Benutzer die Indikatoren für den Filter sowie Datum und Zeit für den Eintrag festlegen. Um den eigentlichen Eintrag zu schreiben, hat der Benutzer ein Textfeld mit limitierter Grösse zur Verfügung.

Verbesserungsmöglichkeiten

Die "Notiz erfassen"-Funktion könnte in den Journalscreen eingebaut werden, anstatt eine separate Funktion darzustellen. Zudem kann das Textfeld bei längeren Texten Layout-Probleme verursachen.



(a) Auswahl des Klienten (b) Erstellen des Eintrags

Abbildung 3.4: Journal Eintrag erfassen

3.2 Architektur

In diesem Abschnitt wird die Architektur der ursprünglichen Android App aufgezeigt und analysiert.

RESTful Http Client

Die Kommunikation mit dem RedLine Server wurde mit einem selbst implementierten REST Client durchgeführt, welcher OkHttp [2] für die Netzwerkkommunikation benutzt. Die Http Anfragen werden dabei von der Verarbeitung getrennt, der Client und die Verarbeitung werden in einem separaten Thread ausgeführt. Die Verarbeitung und das Buffering geschieht im UserDatabase Objekt. Es regelt und überwacht den REST Client, verarbeitet bei Bedarf JSON Strings zu Objekten und speichert Listen der zuletzt angefragten Daten. Dabei wird die Kommunikation zwischen den Threads über eine Queue zur UserDatabase und über Callbacks zurück zum User Interface realisiert. Ein Objekt namens "Task" wird als Steuer- und Payload-Objekt verwendet.

Ein typischer Ablauf sieht dabei so aus: Ein neuer Screen wird geöffnet, wobei das Startverhalten beschreibt, dass ein Task generiert wird, der alle zugängigen Bereiche herunterladen soll. Der Screen hat eine Referenz auf die UserDatabase und greift darüber auf die Queue zu und schreibt das Task Objekt hinein. Im Netzwerkthread wartet ein Runner darauf, dass die Queue einen Task herausgibt. Dort wird der Task an den REST Client weitergegeben, wo ihn ein switch-case verarbeitet, um die URL zu bauen und mittels des OkHttp Clients eine Abfrage zu starten. Nachdem das Resultat erhalten wurde, wird es vom Task Objekt selbst verarbeitet und zu einem JSON Objekt oder einem Bild geparsed. Die UserDatabase überprüft nun, ob die Antwort gültig ist und startet bei Misserfolg zwei identische Versuche.
Bei Erfolg wird der Task mithilfe seines String Identifiers in einem switch-case von einem Stück Code, der für jeden Getter und Setter leicht anders ist, verarbeitet. Am Ende dieser Verarbeitung wird ein Feld im UserDatabase Objekt mit den neuen Daten, die jetzt als Objekte realisiert sind, gefüllt. Ein Callback an die MainActivity signalisiert dem User Interface, dass es neue Daten gibt. Auch hier wird der Task zur Identifizierung mitgegeben. In der Main Activity wird anschliessend die Updatemethode des aktuell offenen Screens aufgerufen. Dieser unterscheidet wieder mithilfe des Tasks, welche Daten verfügbar sind und holt sich diese dann über die UserDatabase. Dieser Zyklus wird nur beim erstmaligen Öffnen eines Screens oder bei einer Interaktion des Benutzers ausgeführt.

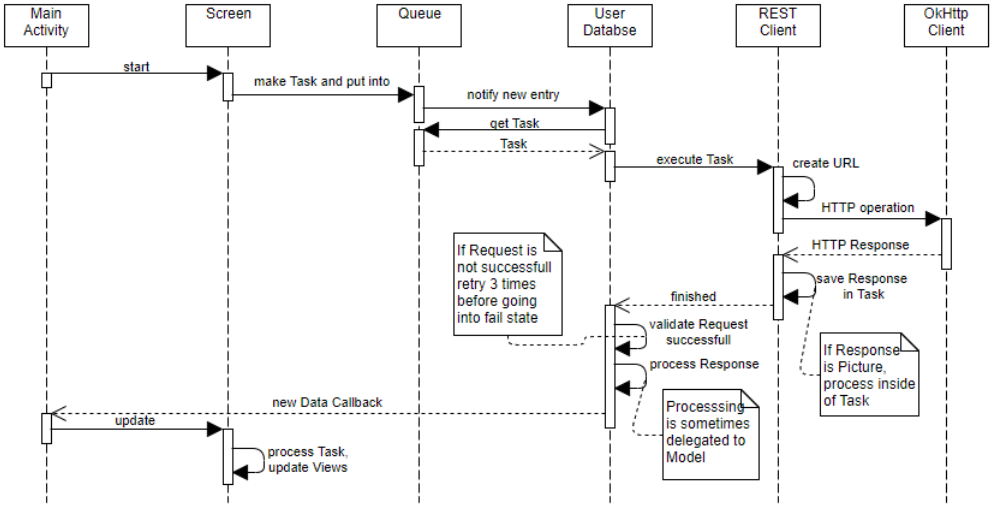


Abbildung 3.5: Sequenzdiagramm einer Netzwerkoperation

Verbesserungsmöglichkeiten

Die Klasse UserDatabase hat zu viele Aufgaben gleichzeitig und wird deshalb überall in der App benötigt. Als Folge davon wird eine Referenz darauf an jeden Screen und zum Teil noch an Subsysteme weitergegeben. Bei einer Weiterentwicklung der API muss zwar lediglich ein weiterer Case hinzugefügt werden, dies kann aber sehr schnell zu einer unübersichtlichen Klasse führen. Zudem hat die UserDatabase keine klare Aufgabe, denn das parsen und überprüfen der Daten wird nicht bei allen Ressourcen in der UserDatabase gemacht. Bei manchen wird das Objekt in der UserDatabase generiert, bei anderen im Task Objekt und manchmal wiederum kann das JSON Objekt dem Konstruktor eines Objekts weitergegeben werden. Des Weiteren agiert die UserDatabase als Buffer für Daten. Letzterer hat keine besonderen Kapselungsmassnahmen und kann direkt ausgelesen oder verändert werden. Er wird beim Erhalt neuer Daten einfach überschrieben, so kann es zu Serialiserungsproblemen kommen.

Das Task Objekt hat als Steuerungselement des Netzwerks eine klare Aufgabe im System, es ist jedoch zu gross. Es soll den kompletten Kontrollfluss einer Netzwerkoperation beinhalten und steuern. Dies kann vor allem beim Verarbeiten von speziellen Operationen, wie zum Beispiel Bildern, komplex und unübersichtlich werden.

Das Mapping der JSON Objekte in die Modellklassen geschieht an verschiedenen Orten. Dies verschlechtert die Wartbarkeit, da die Zuständigkeit an mehreren Orten überprüft werden muss. Es führt zudem dazu, dass die Datenvalidierung komplex und unübersichtlich wird. Somit müssen Daten zum Teil mehrfach validiert werden um sicherzustellen, dass die Daten vollständig sind.

Durch die ganze Architektur zieht sich das Problem der hohen Kopplung und ein tiefe Spezifität. Ein Austausch einer bestimmten Funktion oder eines bestimmten Verhaltens ist sehr zeitaufwändig und fehleranfällig. Dies müsste bei einer Überarbeitung

verbessert werden.

3.3 User Interface Hierarchie

Das User Interface basiert auf einem Objekt, welches mit Views gefüllt und geleert werden kann. Dieses Objekt bleibt während der ganzen Laufzeit bestehen und kann mit einer Liste verglichen werden. Während der Laufzeit delegiert die Main Activity nun die verschiedenen Funktionen an Screens, welche dann Zugriff auf diese Liste haben und sie mit Views befüllen können. Für die Screens ist diese Liste die einzige Möglichkeit, Elemente im User Interface anzuzeigen. Die Screens verarbeiten somit die Daten der UserDatabase und mappen sie auf Views, welche diese dann wiederum in die Liste einfügen. Das Layout, welches für den Benutzer sichtbar ist, hängt von der Reihenfolge ab, mit welcher die Views in die Liste eingefügt werden. Komplexere Layout Elemente wie zum Beispiel der Selektor eines Bereiches werden zuerst in einer Sub-Hierarchie erstellt und dann der Liste hinzugefügt. Bei der Interaktion des Benutzers mit einer View müssen meist Teile der Liste ausgetauscht werden. Dies geschieht über eine Hilfsmethode, welche die unteren "n" Elemente der Liste löscht und mit den neuen Views füllt.

Falls die Funktion nicht in ein Listenlayout passt, kann auf die übergeordnete View zugegriffen werden, über welche dann Views an einer absoluten Position angezeigt werden können.

Verbesserungsmöglichkeiten

Auch wenn die Einschränkung, dass User Interface nur durch einen Entrypoint verändern zu können, den Vorteil hat, erweiterungsfreundlich zu sein, hätte es dafür einige Möglichkeiten gegeben, welche die Screens nicht so stark einschränken. Das Arbeiten mit einer primitiven Liste führt zu viel Managementcode. Beim Ändern des Layout könnte dieser Managementcode vergessen werden, was zu umständlicher Fehlersuche führt. Das Delegieren der verschiedenen Funktionen in eine eigene Klasse ist sinnvoll, dabei wurde jedoch nicht die von Android vorgesehene Fragment API verwendet.

Der Verzicht auf die Trennung der Datenbankoperationen von den User Interface Operationen (wie etwa durch ein ViewModel) führt zu grossen Klassen mit zu vielen Aufgaben. Auch das Mapping der Modelle auf die Data Transfer Objects wurde nicht vorgenommen. Dabei entsteht dasselbe Problem.

3.4 Dependencies

In Tabelle 3.1 werden alle externen Dependencies und Libraries sowie deren Lizenz aufgelistet.

Dependency / Library	Version	Lizenz	Verwendung
Android Support Library	28.0.0	Apache 2 [3]	Standard Android Library, welche die gesamte Grundfunktionalität bereitstellt. Ersatz der Android Support Library.
Squareup okhttp3	3.2.0	Apache 2 [3]	HTTP Client für verschiedene Anwendungen.

Tabelle 3.1: Dependencies und Libraries

Kapitel 4

User Interface Design

Teil des Arbeitsauftrages war es, ein Design für alle Funktionen zu erstellen, welche wir durch unsere Evaluation als nützlich für Betreuer ausgearbeitet haben. Dies beinhaltet auch Funktionen, welche aus Zeitgründen nicht realisiert werden. Das Design sollte sich an der bisherigen App sowie an der RedLine Webseite orientieren. Im folgenden Kapitel werden auf die Design Entscheide und den Prozess eingegangen.

4.1 Vorgaben

Als Vorgabe wurde definiert, dass Betreuer, welche bereits mit dem RedLine Web System vertraut sind, sich auch in der App gut zurechtfinden. Des Weiteren sollte auf das Design der vorherigen App aufgebaut werden, da dies sehr gut angekommen ist und sich viele Benutzer bereits daran gewöhnt haben.

Aus dem Evaluationsprozess mit den Betreuern haben sich folgende Wünsche ergeben:

- Journaleinträge schlichter gestalten
- Namen und Bereiche sollen nicht mehr an unnatürlichen Stellen einen Zeilenumbruch erfahren
- Die Navigation in der App soll schneller und einfacher gestaltet sein

Ausserdem wurden folgende Features aus RedLine für die App ausgewählt:

- Klienteninformationen vollständig anzeigen
- Diverse Gesundheitsdaten anzeigen und schreiben
- Reservemedikamente sollen angezeigt und abgegeben werden
- Die Präsenzliste soll angezeigt und bearbeitet werden

4.2 Designprozess

Der Designprozess beinhaltet das Analysieren der Funktionen, welche realisiert werden sollen und das Verbinden dieser Funktionen mit den User Interface Elementen.

4.2.1 Grundlagen

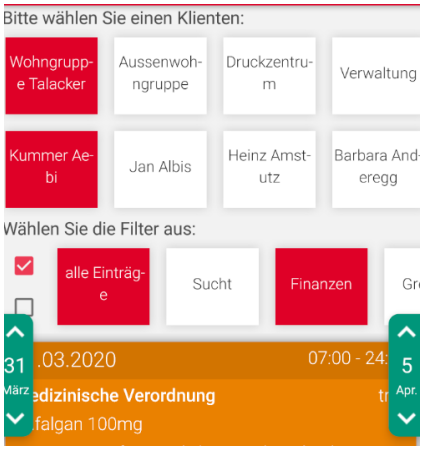
Da die App bereits einen Vorgänger besitzt, müssen viele Fragen bezüglich übergreifendem Design nicht mehr geklärt werden, da auf dem Vorgänger aufgebaut werden kann. Es ist wichtig, dass die Elemente an welche die Benutzer der alten App bereits gewohnt sind, möglichst unverändert bleiben, um den Wechsel auf eine neue Version einfacher zu gestalten. Konkret bedeutet dies, dass die Menüführung, die Iconografie und die Funktionsweise gleich bleiben müssen.

Beim Designen von neuen Funktionen ist es wichtig, dem Benutzer zu zeigen, was er vor sich hat, sodass er sich nicht zuerst durch das Menü oder den Titel durchlesen muss, um zu sehen wo er gerade ist. Um dies zu erreichen, können verschiedene visuelle Signale genutzt werden. Diese visuellen Signale basieren auf Gegenständen oder Zeichen, die der Mensch mit gewissen Situationen in Verbindung bringt. Zum Beispiel verwaschene Grün- und Blautöne bei medizinischen Untersuchungen oder der visuelle Output eines EKGs, dass den Herzschlag misst, für Vitalzeichen.

Als weitere Designguideline wurde Googles Material Design genutzt. Material bietet den Vorteil, einfache Elemente wie Textfelder in einer Library für Android zur Verfügung zu stellen und somit den Entwicklungsaufwand minimiert. Die komplexen User Interface Elemente reagieren ohne grosse Konfiguration so, wie der Benutzer es sich gewohnt ist. Es wurden nicht alle Designguidelines von Material umgesetzt, stattdessen wurden sie als Referenz verwendet.

4.2.2 Menüführung

Die Menüführung wurde auf dem Vorgänger aufgebaut, jedoch wurden alle Elemente modernisiert und so angeordnet, dass es keine überlappenden Elemente mehr gibt. Die moderneren Elemente sind vor allem bei den Filtern erkennbar, die neu mit Chips realisiert wurden. Diese bieten den Vorteil, dass sie weniger Platz benötigen und deshalb zwei in Reihen untereinander dargestellt werden können. Sie passen sich des Weiteren an die Länge des Inhaltes an, weshalb sie keine unnatürlichen Zeilenumbrüche mehr verursachen.



(a) Filter der alten Applikation

(b) Filter der neuen Applikation

Abbildung 4.1: Filter der beiden Applikationen im Vergleich

4.2.3 Listeneinträge

Viele der Informationen werden in Listenform angezeigt. Es ist deshalb sinnvoll, ein Design zu erstellen, welches an vielen Orten wiederverwendet werden kann. Deshalb wurde ein neues Element designt, welches diese Funktion abdeckt. Dieses Designelement ersetzt die Einträge im Tages- und Verlaufsjournal und es wird zudem bei allen Gesundheitsdetails verwendet. Dies soll einer Kritik der ursprünglichen Einträge entgegenwirken, dass sie dank der Farbcodierung zwar angenehm zu lesen sind, jedoch einen zu grossen Blickfang darstellen. Beim neuen Design wurde deshalb darauf geachtet, die Farbcodierung beizubehalten, jedoch an der Menge der Farbe zu sparen. Die Farbcodierung wurde somit auf eine kleinere Fläche reduziert, welche die Einträge jedoch noch klar gruppiert, siehe Abbildung 4.2-I. Da es sich um zeitlich geordnete Daten handelt, wurde ein vertikaler Balken (siehe Abbildung 4.2-II) eingefügt, um den Lesefluss der Listeneinträge zu verbessern und visuell darzustellen. Dieser Balken zieht sich über alle Einträge einer Liste hinweg und erinnert an einen Fahrplan. Eine weitere Kategorisierung der Einträge war bisher schwer, da immer zuerst der Titel gelesen werden musste. Um dies zu vereinfachen wurden Icons eingefügt, welche den Eintrag kategorisieren, siehe Abbildung 4.2-III. Um die Zeitspanne darzustellen, wurden am rechten Rand Plätze für Zeit und Datumsangaben bereitgestellt, welche untereinander angeordnet sind, siehe Abbildung 4.2-IV. Für weitere Zusatzinformationen wie beispielsweise den Autor gibt es am unteren Rand einen Platz, siehe Abbildung 4.2-V. Bei Einträgen mit einem Foto wird dieses direkt im Eintrag angezeigt oder kann per Knopfdruck hinzugefügt werden, wie in Abbildung 4.2-VI ersichtlich ist.

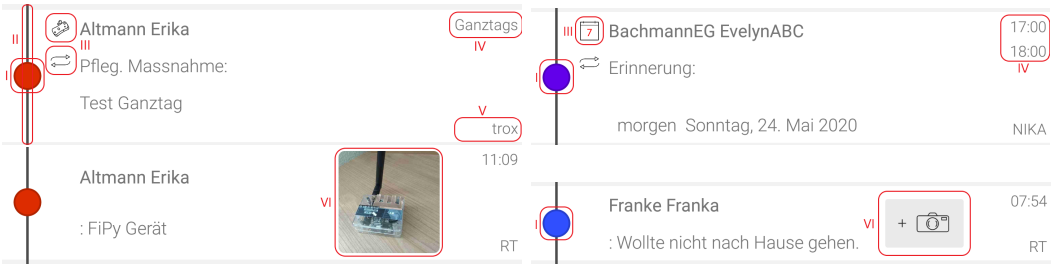


Abbildung 4.2: Listeneinträge am Beispiel des Tagesjournals

4.2.4 Card Einträge

Card Einträge sind Listeneinträge, welche Sublisten beinhalten. Sie sollen als visuelle Barriere dienen, die diese Listen klar voneinander trennen. Sie kommen zum Einsatz wenn nicht die Reihenfolge der Daten im Vordergrund steht, sondern deren Gruppierung. In Abbildung 4.3 ist die Umsetzung sichtbar. Das Layout orientiert sich an einem Etikett, welches gebündelte Informationen anzeigt.

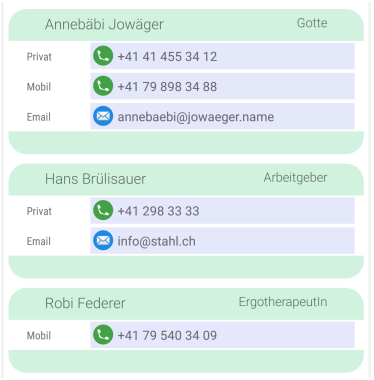


Abbildung 4.3: Card Einträge am Beispiel der Kontakte

4.2.5 Sheet Einträge

Sheet Einträge sollen dem Benutzer das Gefühl geben, er schaue auf ein Blatt Papier auf dem viele Informationen aufgelistet sind. Diese Einträge kommen dann zum Einsatz, wenn die Daten aus vielen einzelnen oder langen Texten bestehen, die mittels Überschriften und Kategorisierungen einfach lesbar sind, siehe Abbildung 4.4. Sie sind inspiriert durch Akten, wie man sie zum Beispiel vom Arzt oder einer Behörde kennt, in welchen Informationen über eine Person untereinander aufgelistet sind.

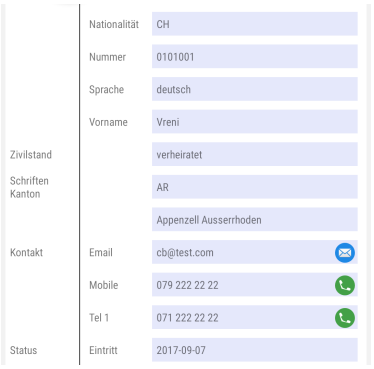


Abbildung 4.4: Sheet Einträge am Beispiel der Klienteninformationen

4.2.6 Iconografie

Wie bereits erwähnt, ist die Iconografie einer der Punkte, die im neuen Design verbessert werden sollte. Icons sind beim Designen von User Interfacen wichtig, da sie Informationen schnell kommunizieren können. Dabei ist es nicht wichtig, dass der Benutzer von Anfang an weiss, wofür ein Icon steht, sondern vielmehr, dass das Icon immer mit dieser Funktion im Zusammenhang genutzt wird. So baut der Benutzer mit der Zeit eine Verbindung zwischen der Information und dem Icon auf, was wiederum dazu führt, dass die App schnell und effizient genutzt werden kann. Aus Lizenzgründen mussten alle Icons, welche die App verwendet, neu erstellt werden. Die Icons folgen einem minimalistischen Design mit höchstens einer zusätzlichen Farbe, um den Aufwand dem Rahmen der Arbeit anzupassen. In Tabelle 4.1 sind die Icons und deren Funktion beschrieben.

	Gesundheits-informationen		Arbeitsumgebung		Person
	To-do-Liste		Wohnbereich		Notfall
	Vitalzeichen		Unterwegs		Umfeld
	Stuhlgang		Pflegepersonal		Medikamente
	Menstruation		Wiederholter Termin		Zeitpunkt
	Epilepsie		Wundpflege		Anwesenheits-kontrolle
	Sicherheit		Pflege		Untersuchung
	Hilfsmittel		Impfung		Verschlüsselung
	Verbindung		Daten-synchronisation		Speichern

Tabelle 4.1: Icons, welche für die App neu designt wurden

4.3 Realisierung der Funktionen

Im Folgenden werden die vollständigen Designs der Funktionen erklärt, welche neu sind oder grundlegend überarbeitet wurden.

4.3.1 Klienteninformationen

Die Klienteninformationen konnten bereits in der alten App detailliert angeschaut werden. Die Informationen waren jedoch nicht vollständig und wurden unübersichtlich dargestellt. Deshalb wurde die Ansicht der Klienteninformationen überarbeitet. Als erstes wurde die Kopfzeile mit Foto und Name verkleinert und mit einem überhängenden Design versehen. Dann wurden Quicklinks auf andere Funktionen eingeführt, um die Navigation zu vereinfachen.

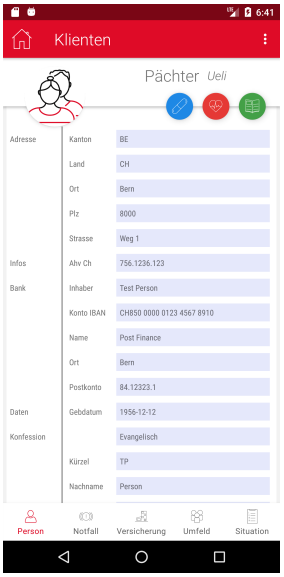


Abbildung 4.5: Klienteninformationen

Nach der Realisierung der Klienteninformationen in RedLine Web wurde beschlossen, welche Informationen für den mobilen Gebrauch nützlich sein könnten. Diese wurden dann in fünf Gruppen unterteilt und in eine Navigationsleiste am unteren Bildschirmrand eingepflegt. Mithilfe dieser Navigationsleiste sollte das Finden von Informationen schneller und einfacher werden.

Die eigentlichen Informationen werden mithilfe von Sheet- und Card-Einträgen in einer Liste aufgeführt.

4.3.2 Medabgabe

Die Medikamenten Einsicht und Abgabe ist ein neues Feature, dass es Betreuern erlauben soll, den Abgabestatus regelmässiger Medikamente zu erfassen sowie auch situationsbedingte Medikamente (Reservemedikamente) abzugeben. Nach der Auswahl des Klienten wird dem Betreuer eine Liste von modifizierten Card Einträgen angezeigt. Diese beinhalten ein Segment, welches das Medikament und die Dosis beschreibt. Am unteren Rand ist dann jeweils zu erkennen, ob und wie viel des Medikaments der Klient bereits erhalten hat. Der Card Eintrag wurde modifiziert, um besser klar zu machen, was der Benutzer gerade anschaut. Die kursive Schrift auf dem gelben Untergrund soll an ein medizinisches Etikett erinnern, dass man zum Beispiel von einer Medikamentenverpackung kennt.

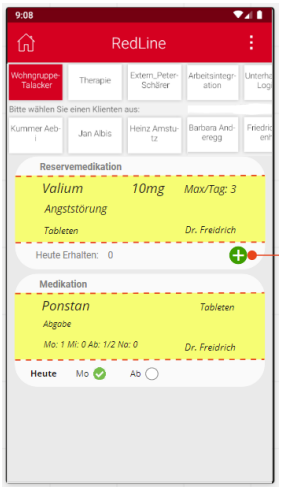


Abbildung 4.6: Medikamentenabgabe

4.3.3 Gesundheitsinformationen

Die Gesundheitsinformationen sind die wohl grösste Erweiterung der neuen App. Sie beinhalten sehr viele Informationen über den aktuellen Zustand des Klienten und bietet die Möglichkeit diese nachzuführen. Diese Informationen können bei einem Arztbesuch als auch im Alltag nützlich sein, um sich eine Übersicht über den Klienten zu verschaffen sowie Aktivitäten an die nächste Schicht zu übermitteln. Die Auswahl an Informationen ist im Filter am oberen Rand zu sehen und zeigt mit den dazugehörigen Icons, welche Informationen erhältlich sind. Es wurden elf solcher Kategorien geplant. Je nachdem welche Informationen angezeigt werden, werden dann Card oder Listen Einträge verwendet. Bei Einträgen, welche bearbeitet werden können, führt eine Interaktion mit dem Eintrag dazu, dass man ihn bearbeiten kann. Alternativ kann über die "+"−Schaltfläche ein neuer Eintrag erstellt werden.

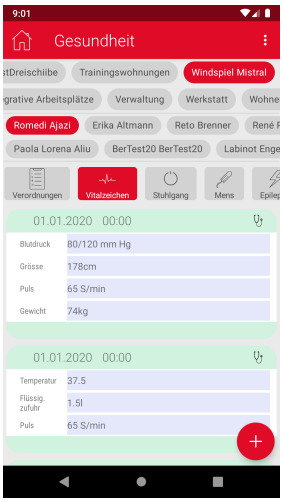


Abbildung 4.7: Gesundheitsinformationen

4.3.4 Präsenzkontrolle

Die Präsenzkontrolle ist ein Feature, dass Betreuern eine schnelle Übersicht geben soll, wo sich Klienten gerade befinden. Das Design ist so aufgebaut, dass der Benutzer auf einer zweiachsigen Fläche scrollen kann, um die Anwesenheit aller Klienten in einem Bereich über mehrere Tage hinweg zu überprüfen. Das Design wurde von RedLine Web mit Modernisierungen und Änderungen übernommen, um in das Theme der App zu passen.

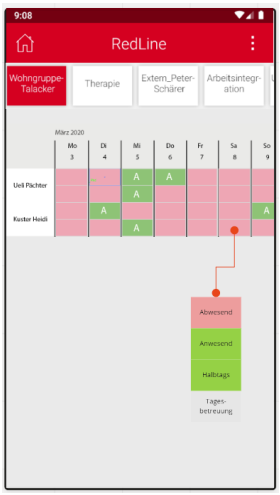


Abbildung 4.8: Präsenzkontrolle

Kapitel 5

Architektur RedLine App V2

5.1 Architektur Überblick

Die gesamte Architektur basiert auf einem drei Schichten Modell, wobei der Data Access Layer in generierte und in implementierte Klassen unterteilt wurde. Bei der Architektur wurde auf Modularität, tiefe Kopplung und hohe Kohärenz geachtet. Dies wurde erreicht, indem jede Klasse eine Aufgabe hat und nur diese Klasse dafür eingesetzt wird.

Am Beispiel von User Story 2 (Anhang A Abschnitt II) wird das Zusammenspiel der Architektur aufgezeigt:
Das Android System öffnet die Login Activity, in welcher eine Instanz des Authenticators das authentifizieren des Benutzers übernimmt. Dieser entscheidet anhand verschiedener Kriterien die Loginvariante. Bei einem RESTful Login nutzt er eine der generierten API Klassen, um einen Authentifizierungsrequest zu starten. Nach Erfolg wird die Main Activity gestartet. Diese öffnet als erstes das Home Fragment, welches zur Navigation der Funktionen genutzt werden kann.

Beim Öffnen der Klientenfunktion wird das Klient Selection Fragment geöffnet. Dieses Fragment stellt beim Cache verschiedene Anfragen für Daten. Der Cache beschafft diese Daten entweder aus den gecachten Daten, aus dem Offlinespeicher oder fordert die Daten neu an. Dazu wird eine DTO Instanz erzeugt, welche geupdated wird. Das DTO nutzt seine generierte API Klasse, um die Daten über eine RESTful Schnittstelle zu erhalten. Da die Anfrage nach neuen Daten non-Blocking realisiert ist, wird im Fragment ein Callback aufgerufen. Das Fragment nutzt die Mapping Instanz, um eine Interface auf die Daten zu erhalten. Dieses übergibt es dem Adapter, welcher wiederum über das Interface auf die Daten zugreift und sie darstellt. Der Benutzer sieht nun die Klienten und kann auswählen, von welchem Klient er gerne genauere Informationen haben würde. Nach der Auswahl eines Klienten wird ein neues Fragment gestartet, welches wiederum alle benötigten Daten beim Cache anfordert.

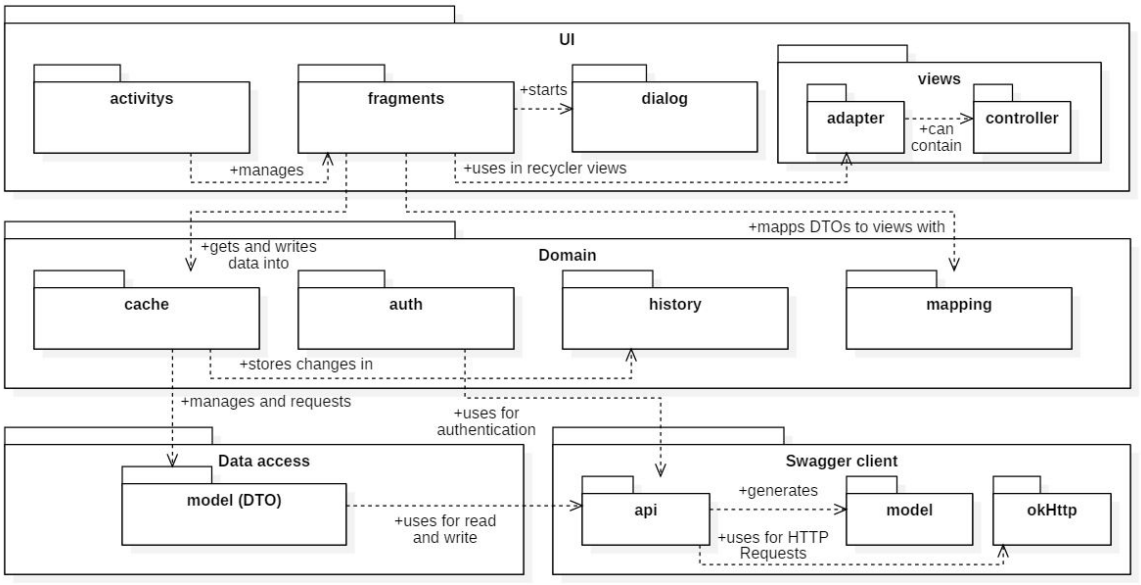


Abbildung 5.1: Architekturüberblick der Pakete und Dependencies

5.2 Authentifizierung

Der Login in die RedLine App erfolgte bisher über eine RESTful HTTP Schnittstelle durch ein POST mit dem Benutzernamen und Passwort. Diese Loginvariante wird auch in Zukunft noch vorhanden sein, jedoch wurde von vielen Institutionen sowie von RedLine selbst eine Zweifaktorauthentifizierung gewünscht. Des Weiteren wollen die Benutzer ihr Passwort nicht bei jeder Verwendung der App eingeben müssen, einfacher wäre eine Bestätigung der Identität mittels Fingerabdruck oder einem Pin-Code.

In Abbildung 5.2 ist die Architektur der Authentisierungs- und Login-Infrasturktur ersichtlich. Der AuthManager hält einen Authenticator, welcher je nach Authentisierungsvariante unterschiedlich ist. Beim Login ist dies entweder ein RESTfulAuthenticator oder ein OpenIDConnectAuthenticator und bei der Geräte-Authentisierung ein DeviceAuthenticator. Diese Architektur ermöglicht das einfache Hinzufügen weiterer Authentifizierungsmöglichkeiten, sofern diese zu einem späteren Zeitpunkt gewünscht sind.

Beim Starten der App wird geprüft, welche Loginvariante von der Institution unterstützt oder verlangt ist und dementsprechend wird das dazugehörige Userinterface angezeigt. Falls die Geräteauthentifizierung im App aktiviert ist und der letzte Login nicht länger als die eingestellte Zeit zurückliegt, wird direkt die auf dem Gerät aktivierte Sperrung aufgerufen. Falls diese Zeit überschritten ist, wird entweder das Benutzername- und Passwortfeld für den RESTful Login angezeigt oder ein Knopf, welcher die Zweifaktorauthentifizierung startet. Letztere wird von der AppAuth [4] Bibliothek durchgeführt, welche die Authentifizierungsseite des eingestellten Zweifaktor-Providers anzeigt.

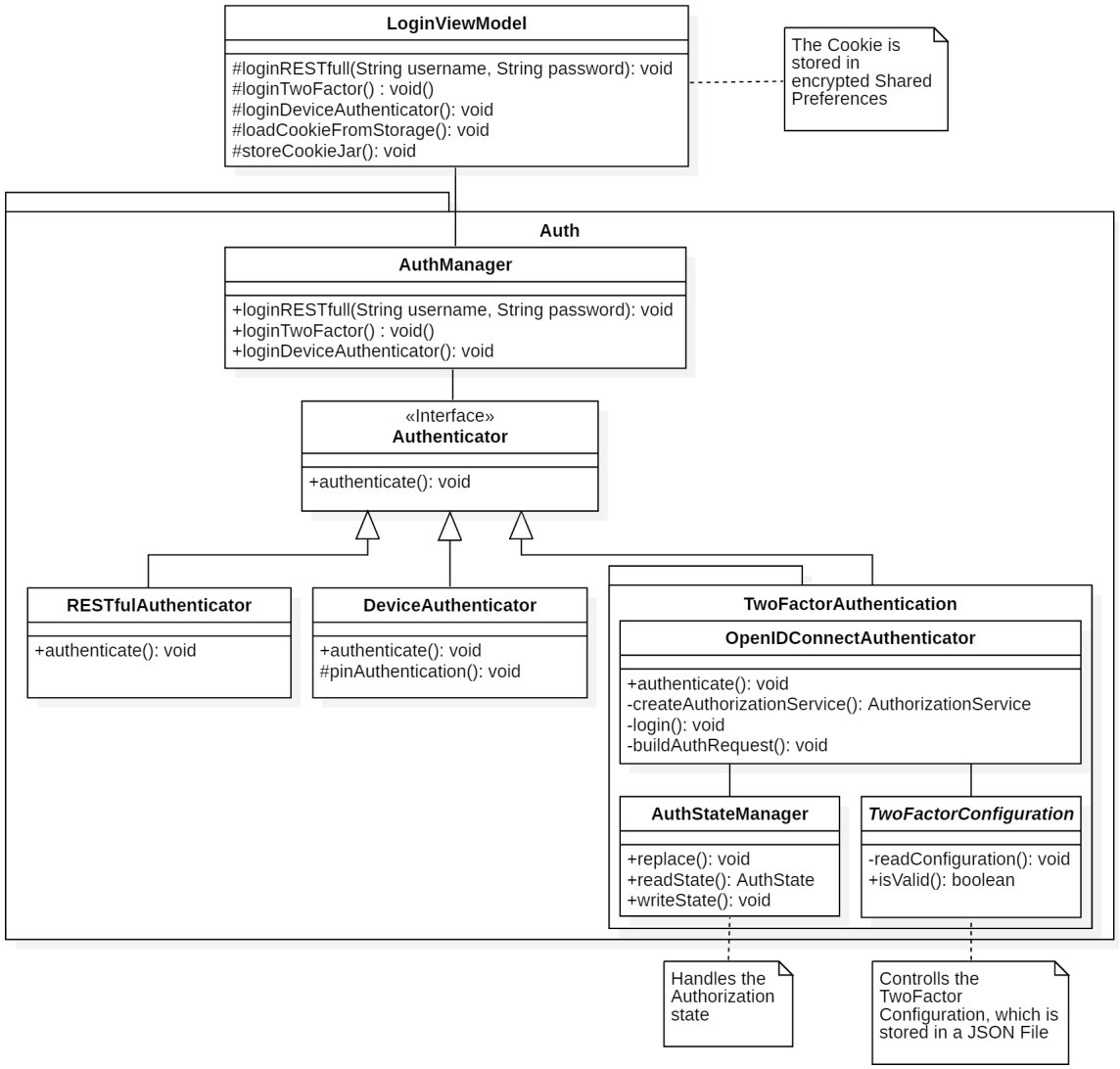


Abbildung 5.2: Authorisierungs- und Login-Architektur

5.2.1 RESTful HTTP Login

Der RESTful HTTP Login gibt ein Cookie zurück, welches eine Gültigkeit von 12 Stunden hat. Innerhalb dieser Zeit müsste sich ein User nicht erneut einloggen. Dies ist jedoch aus Datenschutzgründen bedenklich, da somit der Zugriff einer unbefugten Person ermöglicht wird. Bei der alten Version der App wurde der User bei jedem Verlassen der App ausgeloggt und es wurde ein neues Cookie angefordert. Dies diente zwar der Sicherheit, jedoch nicht der Usability. Deshalb wurde in der neuen Version der App eine Geräteauthentifizierung eingeführt, siehe Unterabschnitt 5.2.2. Der RESTful HTTP Login bleibt erhalten, wurde jedoch in der neuen Version der App aktualisiert und auf den Stand der Technik gebracht.

Cookie

Der Login Request sendet bei erfolgreichem Login ein Cookie zurück, welches das Gerät authentifiziert und wie bereits beschrieben, eine Gültigkeit von 12 Stunden hat. Dieses Cookie wird verschlüsselt auf dem Gerät gespeichert, damit man die App auch ohne erneutes Login verwenden kann. Der Geräte-Authentifizierer lädt dieses Cookie nach erfolgreicher Authentifizierung in den Netzwerkclient, womit dieser wieder Zugriff auf die RedLine Server bekommt. Das Cookie wird als encrypted Shared Preference [5] gespeichert, was Teil des AndroidX Crypto Security Package ist. Somit wird dieselbe Verschlüsselung wie beim OfflineCache (Abschnitt 5.4.3) verwendet.

5.2.2 Geräteauthentifizierung

Die App implementiert eine Geräteauthentifizierung um die Identität des Benutzers sicherzustellen, sofern die Lebensdauer des Cookies nicht überschritten wurde. Somit muss der Betreuer sich nicht jedes Mal neu einloggen, wenn er sein Gerät gesperrt oder die App gewechselt hat. Die Geräteauthentifizierung umfasst die im Android aktivierte Gerätesperrung oder Bildschirmsperre, welche ein Fingerabdruck, Gesichtserkennung, Pin, Passwort oder Muster sein kann. Diese Sperre muss vom Benutzer des Gerätes aktiviert sein, um die Geräteauthentifizierung zu ermöglichen. Ist dies nicht der Fall, erfordert die App bei jedem Zugriff einen RESTfull Login. Die Geräteauthentifizierung ist so implementiert, dass sie automatisch erkennt, welche Identifizierungsmöglichkeit vom Gerät unterstützt wird und auch aktiviert ist. Zuerst wird ein Login mit den biometrischen Daten (Fingerabdruck, Gesichtserkennung) versucht. Der Benutzer hat jedoch die Möglichkeit auf eine andere Authentifizierung wie zum Beispiel den Pin zurückzugreifen. Wenn keine biometrischen Daten hinterlegt sind, wird automatisch auf den Pin, das Muster oder das Passwort zugegriffen. Nach einer gewissen Anzahl Fehlversuchen, welche dem Geräte Standard entspricht, bricht die Geräteauthentifizierung mit einer Benachrichtigung ab und der Benutzer muss sich wieder mit Benutzername und Passwort bei RedLine anmelden. Für eine Geräte-Authentisierung wird Android API 23 benötigt, auf dem Gerät muss also mindestens Android M, bzw. Version 6 "Marshmallow", installiert sein. Während der Entwicklung wurde die minimal unterstützte Android Version erhöht, somit können nun alle Geräte, auf denen die App installiert werden kann, auch die Geräteauthentifizierung verwenden.

BiometricPrompt

BiometricPrompt ist die von Android bereitgestellte und empfohlene Klasse, um eine Authentifizierung mit biometrischen Daten durchzuführen. Abbildung 5.3 zeigt die Abfrage eines Fingerabdruck unter Android 7. Sie wurde in API Level 28 eingeführt und liegt im Paket *android.hardware.biometrics*. BiometricPrompt bietet Methoden zur Abfrage, ob biometrische Daten zur Authentifizierung hinterlegt sind und ob diese verwendet werden können.

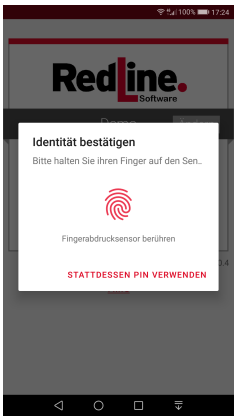


Abbildung 5.3: Geräteauthentifizierung mit Fingerabdruck

KeyguardManager

Mit dem KeyguardManager kann überprüft werden, ob ein Gerät eine Sperrung hat oder nicht. Dies ist wichtig um zu entscheiden, ob die Geräteauthentifizierung verwendet werden kann oder nicht. Der KeyguardManager erkennt automatisch, ob das Gerät mit einem Pin, einem Muster oder einem Passwort gesperrt ist. Dies wird jedoch nur verwendet, wenn

keine biometrische Gerätesperrung vorhanden ist. Wenn ein Gerät per Fingerabdruck oder Gesichtserkennung gesperrt ist, kommt eine Authentifizierung über den KeyguardManager nur zum Einsatz, wenn der Benutzer dies durch ein Klick auf den Knopf "Stattdessen PIN verwenden" wünscht, siehe Abbildung 5.3.

Folgender Codeausschnitt zeigt, wie man mit dem Keyguard Manager eine Pin-Abfrage durchführt.

```
1 keyguardManager = (KeyguardManager) activity.getSystemService(
    KEYGUARD_SERVICE);
2
3 Intent intent = keyguardManager.createConfirmDeviceCredentialIntent(
    "Proof Identity", "PIN:");
4 startActivityForResult(intent, REQUEST_CODE);
```

5.2.3 Zweifaktorauthentifizierung

Zweifaktorauthentifizierung schützt sensible Daten vor unbefugtem Zugriff. Dies ist ein sehr wichtiger Aspekt der Domäne dieser Arbeit, da äusserst sensible Klientendaten über die App erreicht werden können. Es dient ebenfalls der Sicherheit, da über die App Daten verändert werden können. Die Zweifaktorauthentifizierung wurde mit dem OpenID Connect Protokoll [6] implementiert, einem einfachen Identity Layer, welcher auf dem OAuth 2.0 Protokoll [7] aufbaut. OpenID Connect bietet eine Clientauthorisation basierend auf der Authorisierung bei einem Authorisierungsserver, welcher von einem Identity Provider zur Verfügung gestellt wird. Als Identity Provider kann ein kommerzieller Anbieter wie Google oder Okta [8] verwendet werden oder man betreibt selber einen Authorisierungsserver. Das OpenID Connect Protokoll wurde auf Vorschlag und Wunsch von RedLine verwendet und eignet sich sehr gut für diese Aufgabe. Die Android Unterstützung ist ebenfalls gewährleistet.

OpenID Connect Provider: Okta

Okta ist ein Identity Provider, welcher Benutzerauthentifizierung anbietet. In Okta kann man Benutzer sowie Anwendungen definieren und sie einander zuteilen. In der Anwendung kann man Regeln erstellen, wie sich die Benutzer anzumelden und zu authentisieren haben. Eine Variante ist die Zweifaktorauthenzifizierung mit dem OpenID Connect Protokoll. Okta bietet das App Octa Verify [9] für die Zweifaktorauthentifizierung an. Nach einmaligem Login auf der App generiert es sechsstellige Zahlencodes, welche als zweiter Faktor in der Anwendung verwendet werden können. Neben der Verifizierung über diese App kann Okta auch über SMS, Google Authenticator, E-Mail und weiteren Drittanbietern authentifizieren. Somit ist es nicht zwingend nötig, eine zweite App zu installieren.

AppAuth Library

AppAuth for Android [4] ist ein Client SDK für Androidanwendungen, um mit OAuth 2.0 und OpenID Connect Providern zu kommunizieren. Die Bibliothek wird von der OpenID Foundation [10] gepflegt und steht unter der Apache License 2.0 [3]. AppAuth mapped Requests und Responses auf Spezifikationen und bietet Methoden, für einen praktischeren Umgang. Die AppAuth Library übernimmt die gesamte Kommunikation mit dem OpenID Connect Provider und steuert auch den Controlflow der Authentisierung. Die Konfiguration des OpenID Connect Providers wird über eine JSON Datei ermöglicht, in welcher man die Konfiguration des OpenID Connect Providers hinterlegt. Im Folgenden sind die Werte zu finden, welche für Okta benötigt werden:

- **client_id**: ID der Okta-Anwendung, welche die User zugeteilt bekommt.
- **redirect_uri**: URL des Okta Callback, Beispiel: $com.okta.dev-755703 : /callback$

- **authorization_scope**: Legt fest, für was alles authentifiziert wird, beispielweise für E-Mail, Openid oder ein Profil
- **discovery_uri**: URL zur OpenID Connect Konfiguration des Providers bzw. der Anwendung z.B.: $https://dev-755703.okta.com/.well-known/openid-configuration$

Authentifizierungsablauf

Der Authentifizierungsablauf umfasst vier Schritte. Im ersten Schritt wird die Spezifikation des OpenID Connect Provider über den sogenannten Discovery URL abgerufen. Im zweiten Schritt verbindet sich die App mit dem Server des Providers und eine Login Seite erscheint. Der Server prüft den Login und fährt anschliessend mit dem konfigurierten Verfahren weiter. Der dritte Schritt umfasst in diesem Kontext die Zweifaktorauthentifizierung, deshalb wird der Benutzer in der App nun aufgefordert, seinen zweiten Faktor einzugeben. Dies ist z.B. ein Zahlencode von der Okta Verify App. Nachdem dieser Code erfolgreich übertragen wurde, ist die Authentifizierung abgeschlossen. Im vierten Schritt sendet die App den Token an den RedLine Server, um sich dort einzuloggen.

Stand der Zweifaktorauthentifizierung

Die Zweifaktorauthentifizierung wurde implementiert und funktioniert. Sie ist aktuell auf Okta unter Verwendung der Okta Verify App konfiguriert. Jedoch ist die Zweifaktorauthentifizierung aktuell noch nicht verfügbar, da RedLine zuerst die Token-Authentifizierung serverseitig implementieren oder selber einen Identity Server aufsetzen muss. Der Authentifizierungsablauf ist jedoch integriert und funktioniert, allerdings kann man sich darüber nicht in die App einloggen, weshalb aktuell nur der normale RESTful Http Login zur Verfügung steht.

5.3 Datenbank Anbindung

Der Zugriff auf die Datenbank von RedLine läuft wie bisher über die OkHttp [2] Library. Neu wird aber die Schnittstelle mit Hilfe vom OpenAPI Generator generiert. Der Generator generiert einen Client, welcher verwendet werden kann, um Anfragen an den RedLine-Server zu senden. Zusätzlich werden alle Modell Klassen generiert.

5.3.1 OpenAPI Generator

Eine Architekturanforderung, welche wir uns von Anfang an gestellt haben, war, dass die ganze API Anbindung einfach erweiterbar ist. Die Kommunikation zwischen App und Datenbank von RedLine basiert auf einer RESTful HTTP Schnittstelle, welche die Daten im JSON Format sendet. Die API, die bisher im Einsatz war, bietet bereits viele Funktionen wie Journalabfragen, Klientenabfragen und auch Medikamentenabfragen. Für die Neuimplementierung während der Bachelorarbeit wurden neue Funktionen benötigt, welche die RedLine SaaS bereits bietet, jedoch noch nicht in der RESTful Http Schnittstelle vorhanden sind. Die API basiert auf einer OpenAPI Definition, welche in einem YAML File jeden Endpunkt, sowie jede Ressource beschreibt.

Der OpenAPI Generator [11] ist ein Tool, welches dafür ausgelegt ist, möglichst vielseitig einsetzbar zu sein. Er unterstützt über 20 Programmiersprachen und detailliert konfiguriert werden. Der Generator benutzt für jede Sprache Templates, welche auf Mustache basieren. Diese Templates können entweder selber erstellt und bearbeitet werden, oder es können die vorgefertigten Templates für die Sprache seiner Wahl benutzt werden. Die Java Version benutzt standardmässig OkHttp als Web Client und Googles Gson Library um JSON zu verarbeiten. Da diese Libraries bereits in der alten Version verwendet wurden, mussten die Einstellungen, mit Ausnahme der Paketnamen, nicht verändert werden.

Das Generieren des Clients wurde mit dem Terminal Tool durchgeführt:

```
1 java -jar openapi-generator-cli-4.3.0.jar generate
2     -i ./RedLineAPI/redline_api_examples.yaml -o ./RedLineAPI_V2
3     -c config.json -g java --skip-validate-spec
```

Das javabasierte Tool benötigt ein Inputfile, einen Outputpfad sowie zusätzliche Konfigurationen in einem JSON Config-file. Mit der Option -g kann der Generator ausgewählt werden. Hier könnte man bei Bedarf auch eigene Templates angeben. Die Spec-Validierung muss übersprungen werden, da die Spezifikation der API diese nicht besteht, der Grund dafür liegt in den speziellen Tags, welche RedLine zusätzlich für ihre Tools in der Definition hat.

Das Tool generiert nun einen funktionstüchtigen Client, welcher auf die RedLine API zugeschnitten ist. Die gekürzte Struktur einer Beispiel API sieht folgendermassen aus:

```
client
|   ApiClient.java
|   Configuration.java
|   JSON.java
|
+---api
|       PetApi.java
|       StoreApi.java
|       UserApi.java
|
\---model
        Category.java
        Pet.java
        Tag.java
        User.java
```

Der *ApiClient* wird von allen API Klassen verwendet, um Netzwerkooperationen durchzuführen. Er kann über ein Singleton konfiguriert werden, um beispielsweise einen eigenen OkHttp Clienten zu injecten, welcher eigene Interceptor implementiert und Cookies unterstützt. Die API Klassen sind die Einstiegspunkte für den Rest der Software. Sie werden nach Tags organisiert, sodass ähnliche Ressourcen in derselben Klasse gruppiert sind. Die API Klassen sind auch für das Mapping der Endpoints zuständig und haben den Pfad zur Ressource hartcodiert. Jede Abfrage kann synchron, asynchron oder synchron mit Http Informationen gestartet werden. Als Antwort einer Abfrage wird eine Modellklasse zurückgegeben. Diese beinhalten nur Felder zur Speicherung der Daten und können wiederum weitere Modellklassen beinhalten.

5.3.2 Modell Klassen

Die generierten Modell Klassen haben wir in DTO Objekten gewrappt. Dies erlaubt es uns zusätzliche Parameter oder Interfaces den Modell Klassen anzufügen, ohne die generierten Klassen zu verändern. [12]

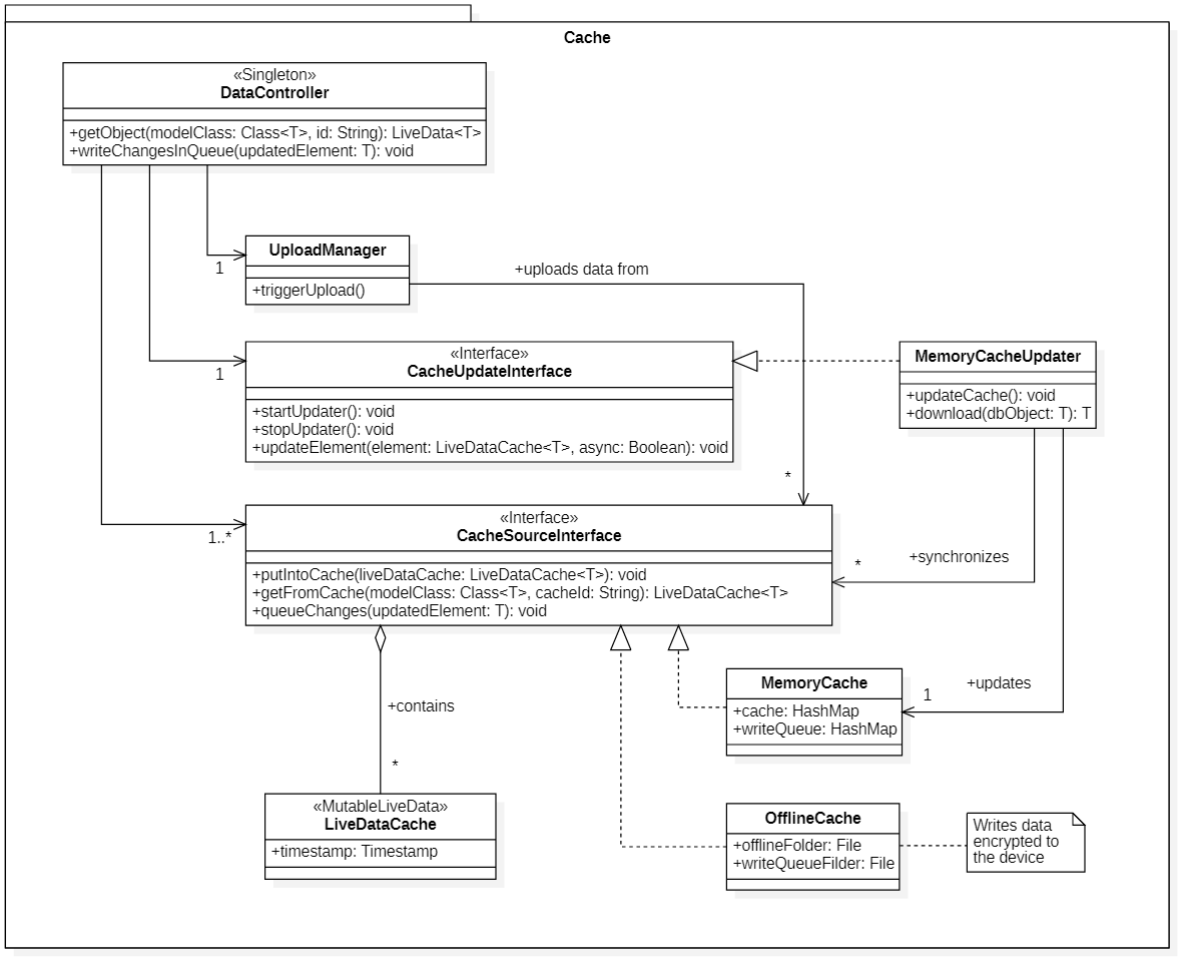


Abbildung 5.4: Cache-Architektur

5.4 Clientside Cache

Wir haben uns aus mehreren Gründen für einen Cache entschieden. Ein Grund war die Anforderung in der Userstorie 5, welche beinhaltet dass Daten offline verfügbar sein müssen. Dafür eignet sich ein Cache gut, da der Rest der App die Daten vom Cache holen kann und keine Fallunterscheidung benötigt, wenn die Daten schon heruntergeladen sind. Ein weiterer Grund ist die Performance. Durch das Cachen der Daten können die Ladezeiten bei schnellem hin- und herwechseln verkürzt werden. Zum Beispiel muss nicht jedes Mal darauf gewartet werden, bis die Klientenliste oder eine Bereichsliste heruntergeladen ist, sondern sie sind sofort verfügbar.

Die Cachearchitektur beinhaltet mehrere Klassen, ersichtlich in Abbildung 5.4.

5.4.1 DataController

Der *DataController* ist ein Singleton, worüber alle Klassen auf den Cache zugreifen können. Er besitzt Referenzen auf alle Komponenten im Cache und verwaltet diese.

Wenn Daten angefordert werden, wird in allen Caches gesucht, ob dieses Objekt schon vorhanden ist. Wenn dies der Fall ist, wird es zurückgegeben und in allen Caches Unterabschnitt 5.4.3 synchronisiert, sowie ein Update angefordert. Wenn das Objekt nicht lokal vorhanden ist, wird stattdessen ein leeres Objekt erzeugt und auf die gleiche Weise weitergearbeitet.

Bei einer Änderung an einem lokalen Objekt oder wenn Daten auf den Server geschrieben werden sollen, wird zuerst das geänderte Objekt in die Schreib-Warteschlangen der Caches geschrieben. Danach werden die lokalen Daten mit den geänderten überschrieben, sodass diese für den Benutzer sichtbar werden. Zuletzt wird ein Upload ausgelöst, wobei versucht wird, die Daten auf den Server zu schreiben.

5.4.2 Upload und Download

Der UploadManager übernimmt die Aufgabe, lokal geänderte Objekte auf den RedLine-Server hochzuladen. Dafür liest er die Schreib-Warteschlangen der Caches und versucht, diese Elemente hochzuladen. Das Einlesen der Warteschlange wird ausgelöst, indem ein Element in die Schreib-Warteschlange geschrieben wird oder eine erfolgreiche Verbindung zum RedLine-Server hergestellt wurde. Letzteres wird benötigt, falls im Offlinemodus gearbeitet wird. So kann bei einer wiederhergestellten Verbindung das Hochladen ausgelöst werden. Falls das Hochladen erfolgreich war, wird das Objekt aus allen Warteschlangen entfernt.

Der Download geschieht durch das CacheUpdateInterface. Das Interface erfüllt drei Hauptaufgaben:

- Bestehende Objekte aktualisieren: Vergleicht die Daten mit dem RedLine-Server und aktualisiert gegebenenfalls die lokalen Daten
- Synchronisieren: Aktualisierte Daten werden in allen anderen Caches synchronisiert
- Objekte entfernen: Entfernt nicht benötigte Objekte nach einer gewissen Zeit aus dem Cache

Diese drei Aufgaben werden regelmässig als Task ausgeführt. Dieser Task wird in einem Hintergrundthread gestartet und beginnt jeweils nach Ende des letzten Tasks. Mit diesem Verhalten wird verhindert, dass ein neuer Task gestartet wird, bevor der letzte abgeschlossen ist. Um Ressourcen zu sparen, wenn die App beispielsweise im Hintergrund ist, lässt sich dieser Update-Task nach Bedarf starten und stoppen.

Zusätzlich wird bei jeder ausstehenden Aktualisierung überprüft, ob lokale Änderungen ausstehend sind. Wenn dies der Fall sein sollte, entsteht ein Konflikt, welcher dem Benutzer mitgeteilt und in der Historie-Komponente (Abschnitt 5.5) verwaltet wird.

5.4.3 Cachearten

Für die App verwenden wir zwei Caches. Ein *MemoryCache*, wo die Daten kurzzeitig gespeichert werden und ein *OfflineCache*, wo Daten zur Offlineverwendung gespeichert werden. [13]

MemoryCache

Der MemoryCache muss Elemente, welche vom Server oder aus dem nicht-flüchtigen Speicher geladen werden, zwischenspeichern. Das heisst, alles was angezeigt wird, landet im Memory Cache. Durch die LiveData von Android können Lifecycleowner die Daten observen und werden automatisch benachrichtigt, wenn sich die Daten im Cache ändern. Zusätzlich kann im Cache abgefragt werden, ob die Daten überhaupt observed werden und ob man sie noch im Memory benötigt. Da mehrere Threads auf den Cache zugreifen ist es wichtig, dass im Cache keine Threading Probleme auftreten.

OfflineCache

Der OfflineCache verwaltet den Zugriff auf das Dateisystem vom Smartphone und kümmert sich um die Verschlüsselung der Daten.

Die Daten welche offline gespeichert werden sollen, werden im App-spezifischen (*context.getFilesDir()*) internen Speicher abgelegt. Die Dateien können jedoch trotzdem von einem Root-Device oder kompromitierten Geräten gelesen werden. Deshalb müssen die Daten verschlüsselt werden. Dies geschieht mit dem Android Security Package [14]. Damit kann ein Verschlüsselungstyp festgelegt werden, wobei *AES256_GCM_SPEC* aktuell die einzige Option ist. Danach wird eine Datei mit dieser Verschlüsselung erstellt und ein Outputstream generiert. Dieser Outputstream kann wie ein normaler Stream behandelt werden und die Daten werden automatisch verschlüsselt. Das Entschlüsseln der Daten funktioniert auf die gleiche Weise, jedoch mit einem Inputstream.

Zusätzlich wurden weitere Massnahmen ergriffen, um die Sicherheit der Daten zu gewährleisten. Die App kann nur auf dem internen Speicher installiert werden, somit werden die Daten nie auf externen Speichermedien gespeichert. Das Backup der App wurde deaktiviert, dies würde die Dateien vom internen Speicher automatisch in die Cloud laden, um sie bei einer Neuinstallation der App wieder zur Verfügung zu stellen. Mit diesen beiden Einstellungen sind die Daten nur auf dem internen Speicher des Gerätes und können nicht von anderen Apps gelesen werden.

5.4.4 LiveDataCache

Die LiveDataCache ist eine `MutableLiveData` mit einem zusätzlichen Attribut für einen `Timestamp`. Damit kann definiert werden, wann die Daten zuletzt aktualisiert worden sind. Die Daten in der LiveData sind die Modell Klassen von der SwaggerAPI.

5.4.5 Cache Design

ClientSide Cache

Der Cache wurde so ausgelegt, dass angezeigte Daten maximal 30 Sekunden alt sind. Dadurch dass der RedLine Server keine Push-Benachrichtigungen bei einer Änderung sendet, müssen die Daten durch Polling aktuell gehalten werden. Zusätzlich werden bei den Daten keine Timestamps mitgesendet, diese sind nur serverintern vorhanden, weshalb lokal verglichen werden muss, ob sich die Daten verändert haben. Der Cache aktualisiert sich nicht weiter wenn die App im Hintergrund ist oder geschlossen wird.

Offlineverhalten

Wenn keine Internetverbindung aufgebaut werden kann, sollen ausgewählte Daten bestehen bleiben, während die restlichen Daten nach Ablauf der Cachezeit nicht mehr angezeigt werden. Um die ausgewählten Daten weiterhin anzeigen zu können, werden sie auf dem Gerätespeicher verschlüsselt gespeichert. So wird garantiert, dass die Daten auch nach dem Beenden der App noch vorhanden sind.

Wir haben folgendes Verhalten für den Cache definiert: Bei einem Login wird der Cache standardmässig geleert. Dies dient dazu, dass bei einem Benutzerwechsel oder Ablauf des Authentifizierungscookie nicht auf schützenswerte Daten zugegriffen werden kann. Wenn sich jedoch der gleiche User wieder einloggt, wird der Cache nicht geleert, dies wird durch eine Serveranfrage nach erfolgreichem Login sichergestellt.

5.5 Historiekomponente

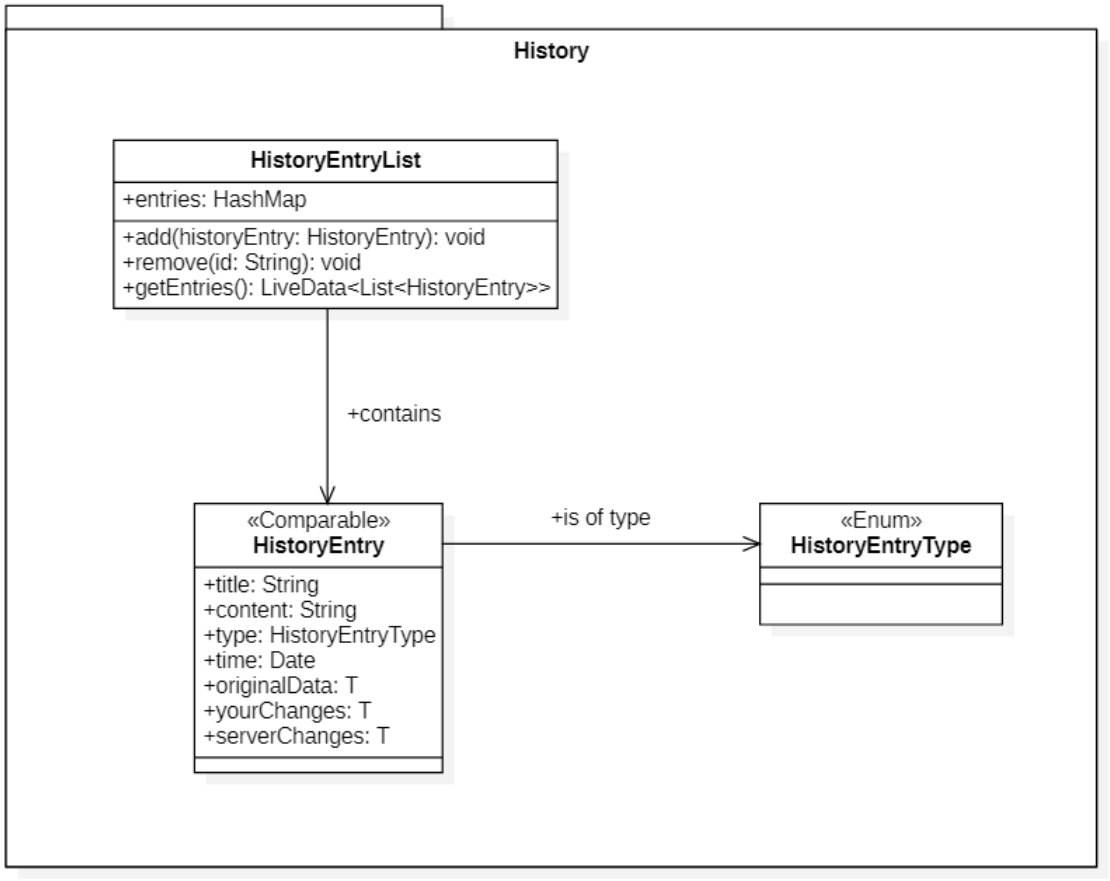


Abbildung 5.5: Historie-Architektur

Die Historiekomponente verwaltet eine Liste von Änderungen, welche auf den Server hochgeladen werden sollen. Dies ist hauptsächlich für den Benutzer gedacht, damit dieser den Upload-Status sieht. Ein HistoryEntry wird erstellt, bevor das Hochladen startet und mit dem Status "Ausstehend" gekennzeichnet. Nach erfolgreichem Hochladen wird der Eintrag auf "Synchronisiert" geändert. Wenn jedoch ein Konflikt mit gleichzeitigen Änderungen auf dem Server entsteht, wird der Eintrag auf "Konflikt" geändert und der Benutzer kann nun beim Eintrag auswählen, welche Version er hochladen möchte. Es stehen ihm drei Optionen zur Verfügung: das ursprüngliche Objekt, seine Änderungen oder die Änderungen vom Server. Nach der Auswahl wird der Eintrag wieder auf "Ausstehend" geändert und ein Upload gestartet.

5.6 Fragment Architektur

Der Grossteil der Applikation wurde in einer Activity realisiert, welche über einen FragmentHolder verfügt, um die verschiedenen Funktionen zu realisieren. Für jede Funktion wird ein Fragment implementiert, welches über eine FragmentFactory zur Verfügung gestellt wird. Die FragmentFactory übernimmt hierbei die Aufgabe, ein Fragment über seine `newInstance` Methode zu instanzieren. Der von Android zur Verfügung gestellte FragmentManager kann diese Fragmente selber managen. Dies hat den Vorteil, dass das Ersetzen von Fragmenten effizient verläuft und vorgegebene Animationen für den Übergang genutzt werden können. Fragmente können zudem auf Hardwareeingaben, wie beispielsweise den "Zurück"-Knopf reagieren. Es wird ein Callback eingesetzt, um dem Fragment die Möglichkeit zu geben, mit der Activity zu kommunizieren.

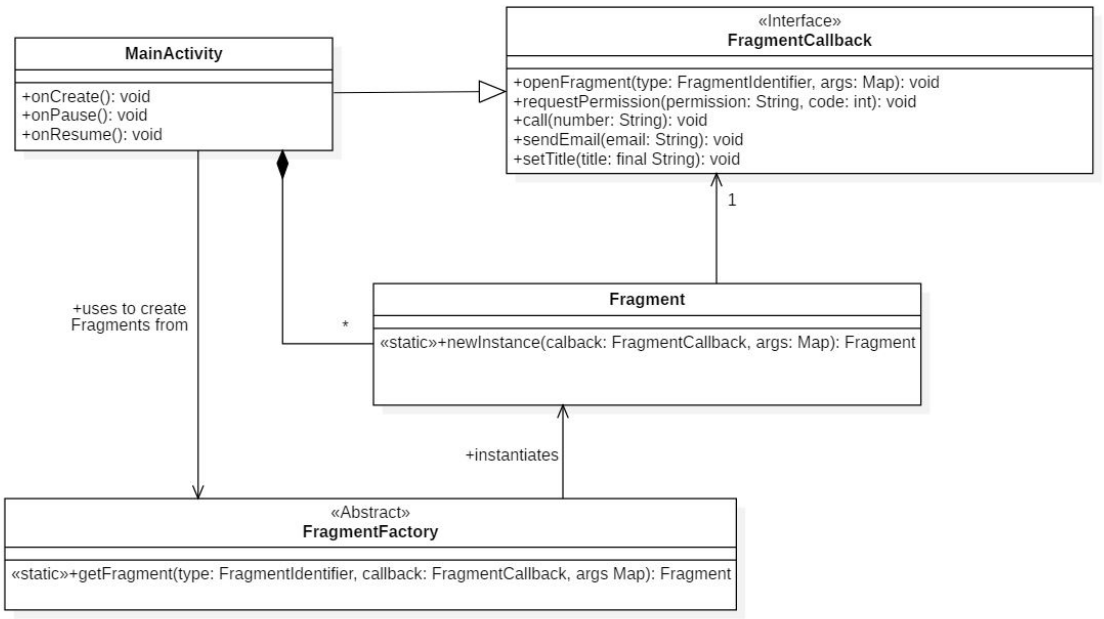


Abbildung 5.6: Fragment Architektur

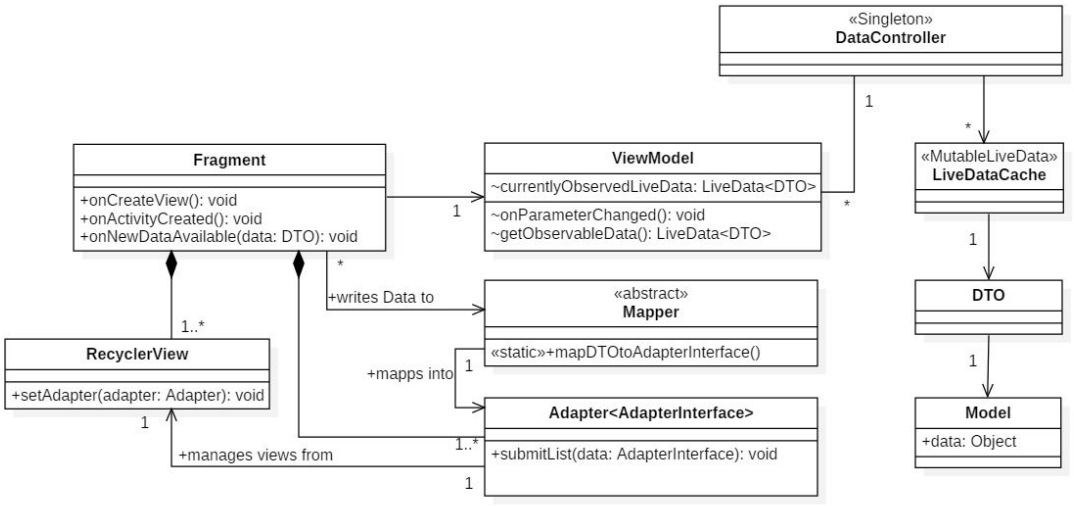


Abbildung 5.7: MVVM Realisierung

5.7 Datenbindung

Die UI Architektur basiert auf dem MVVM-Pattern, dass jedoch in einigen Instanzen erweitert und verändert wurde (siehe Abbildung 5.7), um die Android API zu nutzen. Mithilfe von LiveData wird das Databinding realisiert.

5.7.1 Modelle

Die vom OpenAPI Generator instanzierten Modelle, welche in den Views verwendet werden, sind in einem DTO Objekt gewrapt und über LiveData erhältlich. LiveData vereinfacht das Realisieren von Databinding, indem der Lifecycle der View über das Aktualisieren des UIs entscheidet.

5.7.2 ViewModel

Das ViewModel hat die Aufgabe, Daten der View zur Verfügung zu stellen und Referenzen auf die aktuell offenen LiveDatas zu halten. Die von Android zur Verfügung gestellte ViewModel Klasse kann an eine Activity oder ein Fragment angehängt werden, wodurch es sich an den Lifecycle dieser View hält. Das ViewModel erfährt dabei jedoch nicht denselben Lifecycle wie die View. Während die View im Laufe seiner Lebenszeit mehrere `onPause` und `onResume` erfahren kann, je nachdem ob die View sich im Vorder- oder Hintergrund befindet, bleibt die ViewModel Instanz solange aktiv, bis die View endgültig zerstört wird. Dies macht ein ViewModel in Androidsystemen noch wichtiger als in anderen User Interface Systemen, da das Ändern der Bildschirmgrösse, das Drehen des Bildschirms oder auch das Öffnen eines Overlays, zum Neuzeichnen der Views führt, wobei deren Daten verloren gehen.

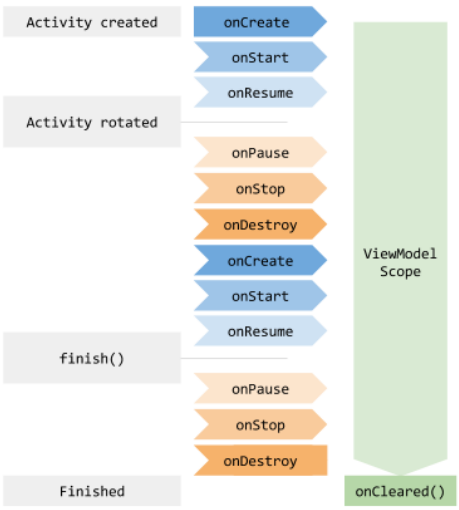


Abbildung 5.8: Lifecycle eines ViewModels

5.7.3 View

Die LiveData, welche das ViewModel bereitstellt, werden von der View observed. Dabei handelt es sich um ein Observerpattern, welches zusätzlich Lifecycleaware ist und die View somit nicht benachrichtigt, wenn sie sich im `onPause` befindet. Bei einem Update der Daten wird ein Callback ausgelöst, in dem die neuen Daten übermittelt werden. Diese Daten werden anschliessend in die vorgesehene View geschrieben. Einzelne Views wie zum Beispiel den Titel des Screens werden direkt aktualisiert und neu gezeichnet. Wenn es sich jedoch um eine Auflistung von Daten handelt, wird ein Adapter verwendet. Dabei werden die Daten einem Adapter übergeben, welcher die Daten in einer RecyclerView darstellt. Letzteres ist in der RedLine App fast überall anzutreffen.

Da das User Interface meist aus wiederverwendbaren Elementen besteht, wurde beschlossen, für diese Elemente Interfaces zu erstellen, welche in einem Adapter auf die View gemappt werden. Dies führt dazu, dass nicht für jedes Modell ein neuer Adapter erstellt werden muss, sondern nur ein Mapping vom Modell auf das AdapterInterface. Ein Beispiel ist die Bereichsauswahl, welche in fast jedem Fragment anzutreffen ist. Dies kann realisiert werden, indem das Modell das Interface implementiert oder indem eine separate Mappinginstanz das Implementieren übernimmt. Da in unserem Projekt die Modellklassen durch eine externe Instanz erstellt wurden, war es nicht möglich, das Interface dort zu implementieren, weshalb ein statischer Mapper diese Aufgabe übernimmt.

5.7.4 Austauschen der Adapter

In einem zustandsbehafteten Screen, in welchem je nach Auswahl des Benutzers eine andere Liste in einer Recyclerview dargestellt wird, muss der zugehörige Adapter ausgetauscht werden können. So muss nur eine Recyclerview erstellt werden, um mehrere Funktionen zu erfüllen. Das Austauschen von Adaptern benötigt weniger Overhead, als zum Beispiel das Einsetzen von Fragmenten um die verschiedenen Listen darzustellen. Das Austauschen der Adapter wird beim Gesundheitsscreen eingesetzt, da der Benutzer normalerweise denselben Klienten und denselben Bereich beibehalten möchte, wenn er die verschiedenen Gesundheitsdaten anschaut. Die Struktur dieser Daten unterscheiden sich jedoch grundlegend voneinander.

Um dieses Verhalten zu realisieren wurde ein Delegator-Pattern eingesetzt. Je nachdem welche Liste der Benutzer ausgewählt hat, wird das Gestalten der Funktion und das Reagieren auf Benutzerinteraktionen an separate Instanzen delegiert. Dies führt zu einer besseren Übersicht und einer einfachen Erweiterbarkeit.

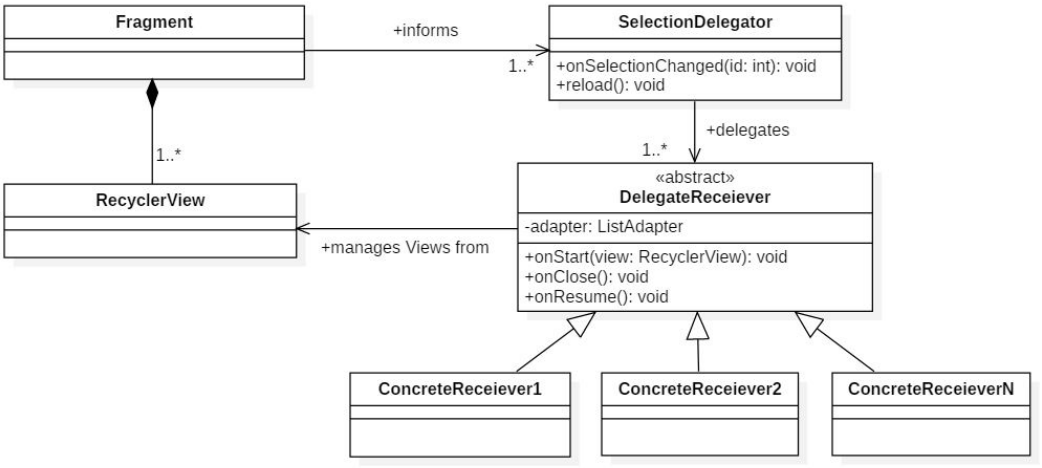


Abbildung 5.9: Delegator Realiserung für Adapter

5.8 Fehlerbehandlung

In diesem Unterkapitel soll es darum gehen, welche Fehlerbehandlungsmethoden wir einsetzen, um eine stabile App zu gewährleisten. Dabei gibt es hauptsächlich zwei Fehler für den User: Dass die Daten nicht verfügbar sind und dass der User sie nicht schreiben kann. Diese Fehler werden im Code weiter unterteilt.

5.8.1 Verbindungsfehler

Der Verbindungsfehler kann immer auftreten und soll allgemein erkannt und dem Benutzer gemeldet werden. Um die spezifische Behandlung der Fehler sollen sich die einzelnen Fragments kümmern.

5.8.2 Read-Fehler

Read-Fehler teilen sich auf in Verbindungsfehler (Unterabschnitt 5.8.1), Serverfehler und Lesefehler vom Gerätespeicher.

Serverfehler

Wenn bei einer 'GET' Anfrage auf den Server ein Fehler auftritt, serverintern oder beim Parsen der Antwort, soll ein Fehler geworfen werden. Wenn dies bei einem Objekt auftritt, welches noch nicht im Cache gespeichert ist, soll der Benutzer auf den vorherigen Screen zurückgeführt werden und es soll eine Fehlermeldung angezeigt werden. Danach kann er die Anfrage nochmals stellen. Wenn das Objekt jedoch schon im Cache vorhanden ist, soll der Fehler vorerst ignoriert werden. Erst falls im nächsten Update Zyklus immer noch ein Fehler gemeldet wird, soll dem Benutzer spätestens nach 30 Sekunden angezeigt werden, dass es Probleme gibt.

Lesefehler

Da die App einen offline Zwischenspeicher anbietet, können beim Lesen dieser Daten Fehler auftreten. Falls so ein Fehler auftritt, wird dieser ignoriert und stattdessen versucht, die Daten vom Server zu holen. Um die Daten nochmals zu lesen, muss der Benutzer diese Anfrage nochmals auslösen.

5.8.3 Write-Fehler

Write-Fehler treten auf, wenn der Benutzer Änderungen vornimmt und diese nicht auf dem Server gespeichert werden können oder sie nachträglich mit dem Server synchronisiert werden.

Änderungen übernehmen

Wenn beim Übernehmen einer Änderung ein Fehler auftritt, ob Verbindungsproblem (Unterabschnitt 5.8.1) oder Serverfehler, soll die Änderungsmaske nicht geschlossen werden und der Benutzer soll eine Benachrichtigung erhalten, dass ein Fehler aufgetreten ist. Zusätzlich soll der Benutzer sehen, wenn sich die Daten auf dem Server ändern, während er Änderungen vornimmt. Somit wird verhindert, dass der Benutzer unwissentlich Änderungen auf dem Server überschreibt.

Offline-Write

Wenn der Benutzer nicht online ist und trotzdem Daten ändern oder erstellen will, soll die Option bestehen, diese Änderung in einer Queue zu speichern, welche im Historyscreen (siehe Abschnitt 5.5) für den Benutzer sichtbar ist. Wenn wieder eine Verbindung hergestellt werden kann, wird diese Queue abgearbeitet. Bei Fehlern soll der Benutzer informiert werden und im Historyscreen den Status der einzelnen Operationen sehen. Durch antippen des Eintrags wird der Benutzer in die entsprechende Eingabemaske geführt und er kann versuchen das Problem zu beheben.

5.8.4 Memory- und Gerätespeicher-Fehler

Memory

Da die App einen Cache unterhält und regelmässig Speicher alloziert, soll auf mangelnder Speicherkapazität reagiert werden. Android informiert die Activities über diesen Zustand, damit diese freiwillig Platz schaffen können, bevor Android keinen Speicher mehr freigibt. Wenn eine solche Benachrichtigung eintrifft, soll der Cache zuerst die Bilder aus dem Cache entfernen und anschliessend alle aktuell nicht referenzierten Daten im Cache freigeben.

Gerätespeicher

Beim Benutzerwechsel in der App wird der lokale Speicher geleert, zusätzlich kann der Benutzer in den Einstellungen den Cache manuell leeren.

5.8.5 Infotainmentsystem

Diese Komponente soll hauptsächlich die Fehleranzeige managen. In der App müssen verschiedene Fehler, Warnungen und Informationen dargestellt werden. Die Behandlung des Fehlers bleibt jedoch beim Aufrufer, das Infotainmentsystem dient lediglich der Anzeige und zum Loggen der Fehler. Das Infotainmensystem bietet Methoden zur Darstellung sowie zum Loggen von Fehler, Warnungen und Informationen. Diese unterscheiden sich hauptsächlich in der farblichen Darstellung auf dem Bildschirm.

Die zweite wichtige Funktion des Infotainmensystems ist das Anzeigen des Verbindungsstatus. Wenn der CacheUpdater merkt, dass ein Request zum Server nicht beantwortet wird, meldet er dies dem Infotainmensystem. Dieses zeigt direkt eine Warnung an, dass die angezeigten Daten möglicherweise veraltet sind, da die Verbindung fehlgeschlagen ist. Diese Warnung bleibt solange bestehen, bis eine Antwort vom Server angekommen ist oder 30 Sekunden vergangen sind. Nach dieser Zeit wechselt die Warnung in die Offlinemodus Anzeige, welche den Benutzer ausdrücklich warnt, dass die Daten nicht aktuell sind. Im Offlinemodus wird diese Warnung direkt nach dem Login angezeigt und bleibt solange bestehen, bis wieder eine Verbindung hergestellt wird.

5.9 API-Erweiterung

Um alle neuen Funktionen einzubauen, welche in den Umfragen und Interviews gewünscht wurden, musste die Kommunikationsschnittstelle zum Server erweitert werden. Informationen wie die Medikamentenabgabe waren in der bisherigen App nicht vorhanden und sind somit auch nicht in die Schnittstelle integriert. Es war nicht möglich, die Schnittstelle während der Arbeit auf der Serverseite anzupassen, weshalb die neuen Funktionen nur schwierig implementiert und getestet werden konnten.

Nach Absprache mit RedLine wurde beschlossen, die API Definition zu erweitern und die Schnittstelle mithilfe von einem Mockserver zu simulieren. Da sich dadurch der Scope der Arbeit vergrösserte, musste mehr Zeit für die API investiert werden, als ursprünglich geplant. Da dies jedoch ein Risiko war, mit dem wir gerechnet hatten, konnten wir frühzeitig reagieren und die Planung anpassen.

Die neuen Funktionen wurden mithilfe einer Analyse der RedLine Web Applikation und dem Kopieren vom Implementationsstil der bisherigen API erreicht. So sollte sichergestellt werden, dass die API möglichst so realisiert werden kann. Die Erweiterungen konnten in Zusammenarbeit mit einem Entwickler von RedLine durchgeführt werden, es war jedoch für ihn nicht möglich, unsere Implementation komplett zu sichten und deren Machbarkeit zu prüfen. Dies führte dazu, dass die API nach initialer Erweiterung inkrementell leichte Veränderungen erfahren hat.

5.10 Dependencies

In Tabelle 5.1 werden alle externen Dependencies und Libraries sowie deren Lizenz aufgelistet.

Dependency / Library	Version	Lizenz	Verwendung
AndroidX	1.1.0	Apache 2 [3]	Standard Android Library, welche die gesamte Grundfunktionalität bereitstellt. Ersatz der Android Support Library
Squareup okhttp3	4.4.1	Apache 2 [3]	HTTP Client für verschiedene Anwendungen
Squareup retrofit2	2.7.2	Apache 2 [3]	Typsicherer REST client für Android und Java sowie gson Konverter
Dagger	2.27	Apache 2 [3]	Statisches kompilezeit Dependency-Injection Framework für Android und Java
ThreeTen Backport	1.4.0	3-Clause BSD [15]	Backport für Java SE6 und 7 Date-Time Klassen
Bumptech glide	4.11.0	Copyright by Google	Bibliothek für das Laden und Cachen von Bildern für Android
Openid AppAuth	0.7.1	Apache 2 [3]	Android SDK für die Kommunikation mit OAuth 2.0 und OpenID Connect Providern
Conscrypt	2.2.1	Apache 2 [3]	Java Security Provider für Teile der Java Cryptography Extension und Java Secure Socket Extension

Tabelle 5.1: Dependencies und Libraries

Tabelle 5.2 listet alle Dependencies und Libraries, welche zusätzlich zu den bestehenden für Tests benötigt wurden. Diese werden nur in die Tests kompiliert, in der fertigen Releaseversion der App sind sie nicht vorhanden.

JUnit5	5.6.1	Eclipse Public License v2.0 [16]	Test Framework um Unit Tests für Java zu erstellen
Espresso	3.2.0	Apache 2 [3]	Framework um Android User Interface Tests zu schreiben. Teil des AndroidX Paket

Tabelle 5.2: Dependencies und Libraries, welche nur für Tests verwendet werden

Kapitel 6

Testing

6.1 Unittests

Die Businesslogik der App wird mit Unittests unter Verwendung des JUnit 5 Frameworks [17] getestet. Somit werden hauptsächlich Klassen aus den Paketen *domain* und *data_access* getestet, sowie einige aus dem *ui*. Die vom OpenAPI Generator automatisch generierten Klassen werden indirekt über die DTO Tests des *data_acces* Layers getestet. Die JUnittests wurden während der Entwicklung der App mit erstellt und gegen Ende der Arbeit, vorallem nach dem Feature Freeze, noch erweitert und optimiert.

Testabdeckung

Die Unittests erreichen eine Testabdeckung von 30% aller Code Zeilen, was 36% aller Klassen entspricht. Dabei ist zu beachten, dass die Unittests nicht alle automatisierten Tests im Projekt sind und dies somit nicht der Testabdeckung des gesamten Projektes entspricht. Da die Instrumented-Tests jedoch keine Testabdeckung messen, ist dies diesbezüglich die einzige Angabe. Weiter zu beachten ist, dass der automatisch generierte Code nicht direkt getestet wird. In Tabelle 6.1 ist die genaue Abdeckung aufgelistet.

Package	Klassen	Methoden	Zeilen
data_access	81% (49/60)	57% (311/541)	53% (794/1473)
domain	13% (10/73)	11% (47/419)	10% (158/1544)
io	50% (150/297)	22% (618/2776)	27% (2421/8690)
ui	5% (9/164)	2% (22/950)	1% (72/5017)

Tabelle 6.1: Testabdeckung der Unittests

6.2 Integrationtests

Die Integrationstests wurden mit dem Espressotest Framework [18] in der Version 3.2.0 erstellt. Android Studio teilt die Tests in zwei Gruppen ein: Sogenannte Instrumented-Tests (androidTest) und normale Tests. Letztere sind Unittests mit JUnit, wie im vorherigen Kapitel beschrieben. Die Instrumented-Tests sind weitaus komplexer als normale Unittests, funktionieren jedoch im Grunde sehr ähnlich. Für Instrumented-Tests gibt es verschiedene Frameworks, für diese Arbeit wurde Espresso ausgewählt, da dieses Framework sehr verbreitet ist und es am besten mit Jenkins und Browserstack kompatibel ist. Zu Beginn der Arbeit wurden Tests mit Appium, einer Alternative zu Espresso, durchgeführt. Die Verknüpfung von Appium und Jenkins/Browserstack war jedoch aufwändiger als mit Espresso, dementsprechend fiel die Entscheidung auf Espresso.

Der Hauptunterschied von Instrumented-Tests und normalen Unittests ist, dass für erstere ein Gerät vorhanden sein muss, auf welchem die App gestartet wird, dies kann ein Emulator oder ein physisches Gerät sein. Vor jedem Test wird das App gestartet und anschliessend getested. Espresso bietet verschiedene Funktionen, um einen Benutzer zu imitieren, beispielsweise durch das Drücken von Knöpfen oder durch verschiedene Gesten. Die Instrumented-Tests bedienen somit die App so wie dies ein echter Benutzer tun würde. Espresso kann dadurch nicht nur prüfen, ob sich die App richtig verhält, sondern auch ob alles dargestellt wird.

Espresso hat jedoch gewisse Nachteile, beispielsweise dass die Tests sehr langsam sind. Die App benötigt bei jedem Input eine kurze Zeit um zu reagieren, Espresso ist somit oft zu schnell. Ein Datenmock verbessert dieses Problem zwar erheblich, da Netzwerklatenzen minimiert werden, allerdings muss in Espresso trotzdem viele künstliche Pausen eingebaut werden, damit der Test funktioniert. Wie bereits erwähnt, werden die Instrumented-Tests auch auf Browserstack ausgeführt, wo sich die Probleme noch gehäuft haben. Mit steigender Anzahl Espressotests stieg auch die Absturzrate von Espresso. Dies ist Möglicherweise auch auf die Leistung der Notebooks und der angeschlossen physischen Geräten zurückzuführen, erschwerte jedoch das Testen erheblich.

Probleme mit Browserstack

Während der Entwicklung sind wir auf Probleme mit RESTMock im Zusammenhang mit Browserstack gestossen. Auf dem lokalen Emulator sowie auf unseren privaten Androidgeräten funktionierte der Mock-Server ohne Probleme, sowohl in den Instrumentedtests als auch im normalen App. Auf den Geräten bei Browserstack, welches ebenfalls physische Geräte sind, startete jedoch der Mock-Server nicht, weshalb alle Tests fehlschlugen. Dieses Problem wurde mit einem improvisierten Mockserver behoben, wie im Anhang unter Anhang B Unterabschnitt IV.II beschrieben. Dies funktioniert gut, jedoch nicht 100% zuverlässig. Durch die Netzwerklatenz schlagen einige Tests manchmal fehl, da alle mit einem Timeout ausgestattet sind. Ein weite-

res Problem, welches bei Browserstack gelegentlich auftritt, ist, dass die Geräte nicht immer funktionieren oder die Tests nicht zuverlässig starten. So kann es möglich sein, dass auf allen bis auf einem Gerät alle Tests positiv durchgeführt wurden und auf dem einen viele fehlschlagen sind. Wenn man die Tests zu einem späteren Zeitpunkt ohne Änderung startet, funktionieren diese auch auf diesem Gerät einwandfrei. Wir stellten zwei mögliche Fehlverhalten fest: Einerseits startet die App gar nicht und der Test bricht nach ungefähr 15 Minuten ab oder gewisse Espressobefehle werden nicht ausgeführt. Im ersten Fall scheint es ein Problem von Browserstack zu sein, da die Espressotests einen Timeout von ungefähr 15 Sekunden haben und Browserstack selber einen von 900 Sekunden, was 15 Minuten entspricht. Bei letzterem Fehlverhalten werden Benutzerinteraktionen nicht durchgeführt, beispielsweise das Klicken auf einen Knopf, wodurch der Test demenstprechend fehlschlägt.

Diese Probleme traten im Vergleich zu der Anzahl Tests auf allen Geräten relativ selten auf, jedoch in jedem gesamten Browserstacktestdurchlauf meist einige Male, sodass der Jenkins Build selten ganz Grün wurde.

Ergebnisse

Das Espressoframework kann keine Testabdeckungsmessung messen, weshalb hier keine Information über die komplette Testabdeckung der gesamten App angegeben werden kann. Die Espressotests umfassen jedoch alle Funktionen, welche die App bietet. Es werden alle Screens, welche die App hat, getestet. Die Tests basieren darauf, was auf dem Screen angezeigt wird und was nach einer Benutzerinteraktion geschieht. Wenn Informationen oder User Interface Elemente nicht angezeigt werden oder nicht sichtbar sind, schlägt der Test fehl. Dies geschieht ebenso, wenn nach dem Klick auf einen Knopf nicht der erwartete Screen oder Dialog geöffnet wird. Abgesehen von der Stabilität liefern die Espressotests somit eine gute Testqualität und aussagekräftige Resultate.

Lessons learned

Die Instrumented-Tests von Android stellen eine sehr interessante Möglichkeit dar, eine Smartphoneapp zu testen. Auch die Möglichkeit, die App auf verschiedenen physischen Geräten zu testen, wie Browserstack sie bietet, ist im Grunde eine sehr gute Möglichkeit, die App zu testen. Durch die Unzuverlässigkeit der Espressotests und der Geräte selber wird die Aussagekraft dieser Tests jedoch stark vermindert. Uns ist bis heute nicht klar, wieso unser ursprünglicher Mockserver auf den Browserstackgeräten nicht funktioniert, auch der Support von Browserstack konnte uns dabei nicht helfen.

Kapitel 7

Fazit

7.1 Zusammenfassung

Die neue RedLine App bietet mehr Funktionen als die alte, wurde optisch überarbeitet und die Tests sowie das Feedback zeigen, dass sie stabiler sowie performanter ist. Durch die Umfragen und Interviews konnte sehr viel Feedback von verschiedenen Institutionen eingeholt werden, was die Entwicklung während der Arbeit, aber auch in Zukunft, unterstützt. Dank neuen Funktionen wie dem Offlinemodus oder der Geräteauthentifizierung steigen die Anwendungsmöglichkeiten sowie die Benutzerfreundlichkeit stark an. Die Klientenansicht sowie die Darstellung der verschiedenen Journale wurden komplett überarbeitet und sind jetzt übersichtlicher und einsteigerfreundlicher. Neu hinzugekommen ist der Medikamentenscreen, in dem die Medikamentenabgabe koordiniert werden kann. Viele alte Features, wie die Kamera oder die Notizen, wurden ebenfalls überarbeitet und bieten eine bessere Benutzererfahrung. Die Kamera verzerrt die Bilder nicht mehr und die Notizen haben einen besseren Textinput, welcher nicht mehr die Zeichenanzahl limitiert. Durch den Cache wurde die Performance der App stark verbessert und dank der Verwendung der aktuellsten, von Android zur Verfügung gestellten Verschlüsselungsalgorithmen, wurde auch die Sicherheit erhöht.

Neben all dem Positiven der Arbeit gab es auch einige negative Aspekte. Wir mussten die unterstützte Androidversion erhöhen und verloren dadurch einige Prozent der verwendeten Geräte. Durch die spezielle Situation während der Arbeit konnte nur begrenzt Feedback zur neuen App eingeholt werden und bedauerlicherweise kein Feedback von den Institutionen. Da die API auf der Serverseite nicht zeitgleich mit der Appentwicklung angepasst werden konnte, musste der Server durch einen Mock ersetzt werden, was seinerseits wieder neue Herausforderungen brachte.

Die Arbeit war somit ein Erfolg und alle Beteiligten sind zufrieden. Neben der neuen RedLine App und den wertvollen Informationen aus den Umfragen und Interviews, war die Arbeit auch sehr lehrreich und interessant. Trotz der speziellen Situation und den dadurch bedingten Remote-Meetings über Skype konnte eine gute Zusammenarbeit von allen Parteien erreicht werden. Dies war eine gute Erfahrung, welche in Zukunft sicherlich hilfreich sein wird.

7.2 Auswertung

Auswertung der funktionalen Anforderungen

Durch die Einteilung aller Umfrageergebnisse in verschiedene Prioritätskategorien konnte der Fokus auf die wichtigen und gewünschten Funktionen gelegt werden. Dies führte zu dem Ergebnis, dass alle Stories, mit Ausnahme von einer, der Must Have Kategorie erfolgreich implementiert wurden. Lediglich die "Schmerzliste", eine Story der Kategorie "Gesundheitsdetails ansehen", wurde nicht implementiert. Diese Story, siehe Abbildung 2.1, wurde nicht implementiert, da sie sehr speziell ist. Dort werden die verschiedensten Schmerzarten in verschiedene Kategorien unterteilt und je nach Altersgruppe (z.B. Kinder und Erwachsene) wurde eine andere Aufteilung und andere Kategorien verwendet. Dies lies sich sehr schwer auf einem Bildschirm eines mobilen Gerätes darstellen, zumal der nutzen dieser Funktion von anfang an minimal war.

Stattdessen wurden noch einige Stories aus der Kategorie "Important" umgesetzt. Diese insgesamt sechs Stories beinhalten einerseits die Gerätesperrung, vier Bildschirme aus den Gesundheitsdetails (Menstruation, Stuhlgang, Hilfsmittel und Sicherheit) und einen Bildschirm aus den Medikationsdaten (unregelmässige Medikation). Diese Stories wurden zusätzlich implementiert, da sie sehr wichtige Funktionen der App darstellen, insbesondere die Gerätesperrung und die unregelmässigen Medikationsinformationen.

Auswertung der nicht-funktionalen Anforderungen

Die konkreten Resultate der NFR Messungen sind in Anhang A Abschnitt IV zu finden. Die Anforderung Reliability R2 wurde nicht erfüllt, da die Anzahl der Fehlversuche in der OkHttp Library aktuell nicht konfigurierbar ist. Drei Anforderungen wurden nur teilweise erfüllt:

- Performance P1: Die minimal unterstütze Android Version wurde von Version 5.0 auf Version 6.0 erhöht.
- Security S3: Die App speichert Daten, diese jedoch verschlüsselt.
- Maintainability M1: Bei Spezifikationsänderungen müssen die Kommunikationsklassen neu generiert werden.

Die Anforderung Usability U2 konnte nicht getestet werden, da die App keinem Betreuer gegeben werden konnte. Alle anderen Anforderungen konnten getestet und erfüllt werden.

Feedback der Tester

Es war geplant, gegen Ende der Arbeit ein Feedback bei den bisherigen Benutzern der App einzuholen. Ziel war es, den Interviewpartnern die neue App vorzustellen, um zu erfahren, wie die Änderungen bei den Benutzern ankommen. Dieses Feedback konnte jedoch dank der laufenden Pandemie nicht eingeholt werden. Als Alternative haben wir beschlossen, die App den Mitarbeitern von RedLine zu senden, sodass diese uns eine Rückmeldung geben können.

Insgesamt haben sechs RedLine Mitarbeiter die Umfrage ausgefüllt, die Resultate finden sich im Anhang unter Anhang A Abschnitt V. Die App kam bei allen sechs Testern überwiegend gut an. Gelobt wurde die Übersichtlichkeit und das neue Design der Journaleinträge sowie der neuen Klientenansicht. Ebenso ist das neue Design nicht so auffällig wie das alte und ist somit nicht mehr ein so grosser Blickfang. Der einzige Negativpunkt des neuen Designs war, dass die Schriftgrösse für einige zu klein ist. Die Performance der App wurde ebenfalls überwiegend positiv bewertet und dies wird sich noch verbessern, sobald wieder auf den live Server zugegriffen wird. Als grösste Stärken der App wurde die Möglichkeit Fotos aufzunehmen, Notizen zu erfassen und die vielen Informationen beziehungsweise die gute Übersicht auf kleinem Raum, genannt.

Diese Umfrage mit den RedLine Mitarbeitern kann keine Rückmeldung von Betreuern in Institutionen ersetzen, ermöglicht jedoch trotzdem einen Einblick von Aussenstehenden.

7.3 Lessons learned

Die wichtigste Lektion, welche sich gezeigt hat, bezieht sich auf die Kommunikation während eines Projektes. Durch Tools wie Skype sind Remotemeetings zwar möglich, für eine effiziente Zusammenarbeit ist jedoch ein physisches Meeting unersetzlich.

Eine weitere Lektion wurde durch die Probleme mit der API gelernt. Das Problem, dass eine API noch nicht fertig spezifiziert ist, die darauf aufbauende Applikation jedoch entwickelt werden muss, ist wahrscheinlicher als erwartet. Es wäre von Vorteil gewesen, damit von Anfang an zu rechnen und eine Lösung zu konzipieren. Die API Definition dann von der Applikationsseite aus zu definieren schafft zwar Abhilfe, die Wahrscheinlichkeit dass die API bei der Realisierung dann jedoch wieder abgeändert wird, ist dafür sehr hoch. Glücklicherweise hatten wir an dieses Problem bereits auf Grund der Wartbarkeit gedacht, weshalb wir für die Implementation der API einen Generator verwendet haben, so hat das Ändern nur einige Minuten in Anspruch genommen.

7.4 Ausblick

Das Projekt hatte von Anfang an das Ziel, die alte App zu ersetzen und in Produktion zu gehen. Der Stand der App nach Beendigung der Bachelorarbeit erfüllt diesen Anspruch noch nicht ganz. Die Integrationsphase konnte nicht abgeschlossen werden, Grund dafür ist die API, welche noch nicht fertig spezifiziert und implementiert ist. Dies wurde in der Risikoanalyse als externes Risiko festgehalten, was bedeutet, dass dies von uns nicht steuerbar war. Wir hatten den Bearbeiter bei RedLine so früh wie möglich über die Änderungen informiert, diese konnten jedoch nicht alle während der Projektdauer umgesetzt werden. Um die App produktionsreif zu machen, müsste die API fertig implementiert und anschliessend allfällige Anpassungen in der App vorgenommen werden. Da dies bei dem Design der Architektur berücksichtigt wurde, würden diese Änderungen innerhalb von höchstens 10 Stunden realisiert werden.

Da die Arbeit jedoch nur eine Androidapp beinhaltet und dies einen grossen Teil des Marktes nicht abdeckt, muss noch eine iOS Version implementiert werden. Diese App könnte sämtliche Designs und Ressourcen von der Android App übernehmen. Auch von der Architektur können Teile wiederverwendet werden, wobei Funktionen der Android API durch äquivalente iOS-Funktionen ersetzt werden müssten.

Ob die neuen Features in Produktion gehen, entscheidet die RedLine Geschäftsleitung über den Sommer 2020. Zuvor müsste jedoch die iOS Version der App auf den selben Stand der Android Version implementiert werden. Da unsere Entwicklergruppe diese Android Version implementiert hat, kam von RedLine das Angebot, die iOS-Version im Rahmen einer Anstellung zu implementieren.

Literaturverzeichnis

[1] Olaf Zimmermann. Smart nfr requirements. Vorlesung, 2019. Vorlesung Herbstsemester 2019, Woche 1, Folie 16.

[2] OkHttp Library. Website. Online erhältlich unter `https://square.github.io/okhttp/`, aufgerufen am 9. April 2020.

[3] Apache License, Version 2.0. Website. Online erhältlich unter `http://www.apache.org/licenses/LICENSE-2.0`, aufgerufen am 8. April 2020.

[4] AppAuth for Android. Website. Online erhältlich unter `https://openid.github.io/AppAuth-Android/`, aufgerufen am 8. April 2020.

[5] Encrypted Shared Preference. Website. Online erhältlich unter `https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences`, aufgerufen am 6. Mai 2020.

[6] OpenID Connect. Website. Online erhältlich unter `https://openid.net/connect/`, aufgerufen am 6. April 2020.

[7] OAuth 2.0. Website. Online erhältlich unter `https://oauth.net/2/`, aufgerufen am 6. April 2020.

[8] Okta — The Identity Standard. Website. Online erhältlich unter `https://www.okta.com/`, aufgerufen am 6. April 2020.

[9] Okta Verify. Website. Online erhältlich unter `https://play.google.com/store/apps/details?id=com.okta.android.auth&hl=en_US`, aufgerufen am 8. April 2020.

[10] OpenID Foundation. Website. Online erhältlich unter `https://openid.net/`, aufgerufen am 8. April 2020.

[11] OpenAPI Generator. Website. Online erhältlich unter `https://openapi-generator.tech/`, aufgerufen am 9. April 2020.

[12] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Amsterdam, 2012.

[13] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Java, 6th Edition*. Wiley Global Education, 2014.

[14] Android Security Library. Website. Online erhältlich unter `https://developer.android.com/topic/security/data`, aufgerufen am 18. April 2020.

[15] The 3-Clause BSD License. Website. Online erhältlich unter `https://opensource.org/licenses/BSD-3-Clause`, aufgerufen am 8. Mai 2020.

[16] Eclipse Public License - V2.0. Website. Online erhältlich unter `https://www.eclipse.org/legal/epl-2.0/`, aufgerufen am 8. Mai 2020.

[17] JUnit 5. Website. Online erhältlich unter `https://junit.org/junit5/`, aufgerufen am 7. Mai 2020.

[18] Espresso Testing Framework. Website. Online erhältlich unter `https://developer.android.com/training/testing/espresso`, aufgerufen am 7. Mai 2020.

[19] Wiko Jerry 3. Website. Online erhältlich unter `https://ch-de.wikomobile.com/m2302-jerry3#bleen`, aufgerufen am 13. Mai 2020.

[20] Cipher Suites OkHttp. Website. Online erhältlich unter `https://github.com/square/okhttp/blob/c70e35802e984e8b9bf3fbf1479deb37c0bdfa43/okhttp/src/main/java/okhttp3/ConnectionSpec.java#L46-L65`, aufgerufen am 15. Mai 2020.

[21] OkHttp configuration for retries, Issue #4239. Website. Online erhältlich unter `https://github.com/square/okhttp/issues/4239`, aufgerufen am 15. Mai 2020.

[22] Reinhard Schiedermeier. *Programmieren mit Java*. Pearson Deutschland GmbH, München, 2010.

[23] Browserstack. Website. Online erhältlich unter `https://www.browserstack.com/`, aufgerufen am 7. März 2020.

[24] Browserstack-Espresso-API. Website. Online erhältlich unter `https://www.browserstack.com/app-automate/rest-api?framework=espresso`, aufgerufen am 7. März 2020.

[25] Webhook Step Plugin. Website. Online erhältlich unter `https://plugins.jenkins.io/webhook-step/`, aufgerufen am 8. Mai 2020.