HOCHSCHULE **TRIER**
Trier University of Applied Sciences
Informatik - Computer Science

Entwicklung mobiler Applikation zur zentralen Verwaltung WG-typischer Aufgaben

Development of an mobile application for central administration to manage flat sharing tasks

Tobias Barwig, Robert Raschel, Simon Ritzel

Bachelor-Projektarbeit

Betreuer: Prof. Dr. Georg Rock

Trier, 18.12.2014

Kurzfassung

Mit der steigenden Anzahl von Studierenden an den Bildungseinrichtungen Deutschlands erfreuen sich die Wohngemeinschaften immer größerer Beliebtheit. Mit den vielen neuen Studenten vergrößern sich gleichzeitig die Wohngemeinschaften und die Probleme die mit solch zusammengewürfelten Personengruppen einhergehen. Viele Studierende kennen das Problem einer chaotischen Zettelwirtschaft in Küche, Bad und Flur.
In dieser Projektarbeit wird die Problematik der Verwaltung von Haushaltsaufgaben in einer Wohngemeinschaft mit meist jungen erwachsenen Studenten behandelt. Ziel ist das oftmals auftretende Chaos (durch Zettelwirtschaft und schlechter Kommunikation) möglichst gering zu halten und die anstehenden Aufgaben und Termine jederzeit klar definiert jedem Mitbewohner zugänglich zu machen. Dieses Ziel soll durch eine mobile Applikation für Smartphones auf Basis des von Google entwickelten Betriebssystems Android erreicht werden.

This project thesis displays the often problematic administration of the various household tasks within a flatsharing student community. Objective is to reduce the mostly chaotic atmosphere (be it due to jumble of bits of paper or bad communication) by defining due tasks and deadlines and publishing them to every flatmate in an easy manner. This should be reached by using a mobile application based in an operation system from Google, called ?Android?. Carrier medium should be a common smartphone. The application out every roommate into position to manage and create different notes and calender appointments. Furthermore it is possible to manage a shopping list where articles can easily be added, deleted or marked. In addition to this shows

Inhaltsverzeichnis

1 Einleitung 1
1.1 Zielsetzung 2
1.2 Ähnliche Apps 3

2 Vorgehensweise 4
2.1 Projektmanagement 4
2.1.1 Use-Cases 5
2.1.2 Mockups 7
2.1.3 Lastenheft 10

3 Anforderungen 13
3.1 Anforderungen für DorMApp 15
3.2 Risiken 15

4 Prototyp 19
4.1 Prototyp Implementierung 19
4.1.1 Datenbankstruktur 20
4.1.2 PHP-Skripte 20
4.1.3 Prototyp-App 23

5 Implementierung von *DorMApp* 31
5.1 Entwicklungsgrundlagen 31
5.2 App 33
5.2.1 Probleme 33
5.2.2 Lösungen 34
5.3 Putzplan 37
5.3.1 Umsetzung 37
5.3.2 Probleme und Lösungen 37
5.4 Einkaufsliste 42
5.4.1 Implementierung 42
5.5 Blackboard 44
5.5.1 Implementierung 44
5.6 App/Datenbank Schnittstelle 46
5.6.1 PHP-Skript 46

5.6.2 PHP-Datenbankschnittstelle 48
5.6.3 Datenbank 49
5.7 Weboberfläche 52
5.7.1 Umsetzung 52

6 Zusammenfassung und Ausblick 55

Glossar 57

Erklärung der Kandidaten 58

Abbildungsverzeichnis

1.1 Marktanteile der Betriebssysteme an der Smartphone-Nutzung in Deutschland von Dezember 2011 bis Juni 2014 2

2.1 Use-Case-Diagramm für das Blackboard 6
2.2 Beschreibung des Use-Cases *Notiz bearbeiten* 7
2.3 Mockup 01: Übersicht Black Board 8
2.4 Mockup 02: Neue Notiz 9
2.5 Mockup 03: Notiz(en) löschen 10
2.6 Inhaltsverzeichnis unseres Lastenhefts 11

3.1 Anforderung F1 - Notiz hinzufügen (Blackboard) 14

4.1 Aufbau der Tabelle tp_test 20
4.2 Aufbau der Tabelle users 20
4.3 Struktur der PHP-Skripte im Dateisystem 21
4.4 Auszug aus login.php 22
4.5 Auszug aus DB_Functions.php 23
4.6 Verzeichnisstruktur des DBPrototyp-Projekts 24
4.7 Start-Oberfläche 24
4.8 Login-Oberfläche 25
4.9 Auszug aus der MainActivity.java 26
4.10 Auszug aus der MainActivity.java 27
4.11 Auszug aus SyncRemoteDatabase.java 28
4.12 Auszug aus UserFunctions.java 29
4.13 Auszug aus JSONParser.java 30
4.14 Auszug aus der DatabaseHandler.java 30

5.1 Verwendung von Secure-Preferences 35
5.2 Auszug aus MessengerService.java 36

1

Einleitung

Haushaltsaufgaben müssen sinnvoll und einfach auf alle Mitbewohner verteilt werden und der aktuelle Stand zu jedem beliebigen Zeitpunkt für jeden einsehbar sein.

Die App bietet jedem Mitbewohner die Möglichkeit Notizen einzusehen und zu erstellen. Außerdem kann eine Einkaufsliste verwaltet werden. In dieser können Artikel hinzugefügt, gelöscht und als gekauft markiert werden. Des Weiteren zeigt ein Putzplan die noch anstehenden bzw. bereits erledigten Aufgaben aus dem WG Haushalt an. Jedes WG-Mitglied kann sich in der App einloggen. Dass ein Smartphone als neues Trägermedium dient, liegt nahe, da nahezu alle der durchweg jungen erwachsenen Personen der Zielgruppe solch ein Gerät besitzen. Außerdem bietet es mit der hohen Konnektivität die perfekte Grundlage alle Mitbewohner jederzeit auf dem gleichen Wissensstand zu halten. Als Betriebssystem kommt die von Google entwickelte Android Plattform zum Einsatz. Durch deren hohen Marktanteil von 68,2% wird hiermit die größte Anzahl an potentiellen Nutzern erreicht (siehe Abb. 1.1).

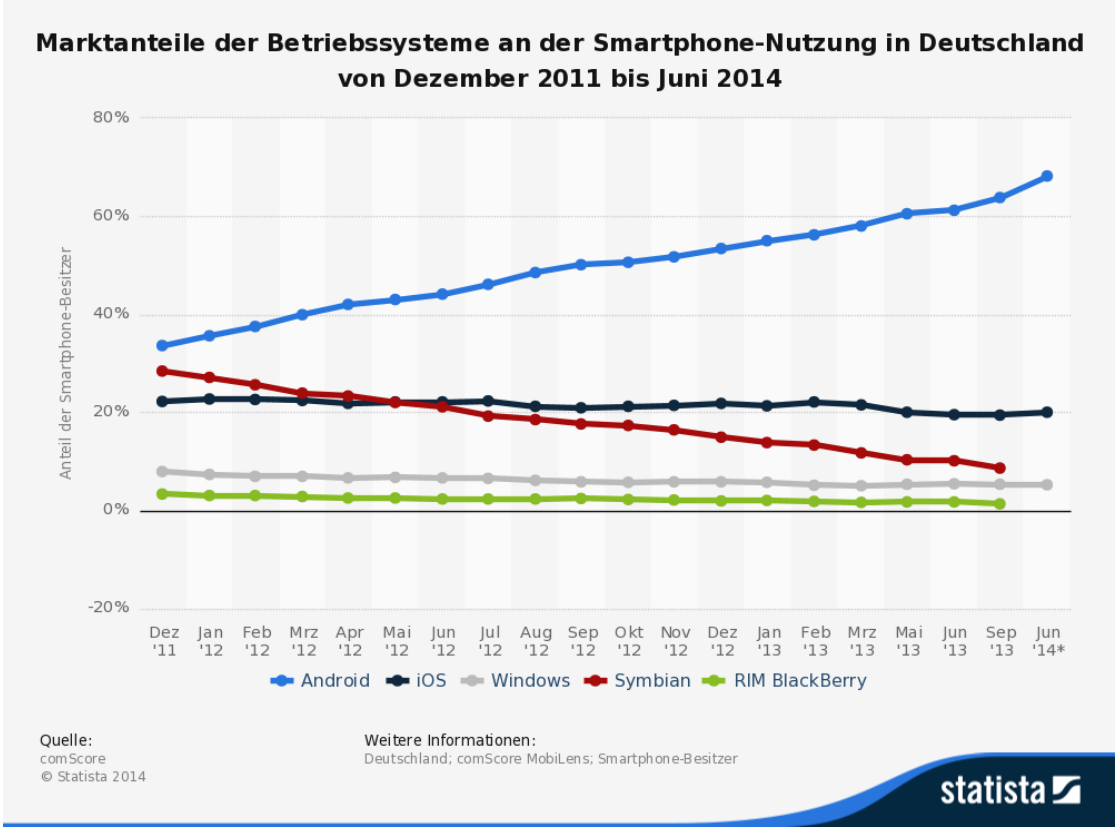

Abb. 1.1. Marktanteile der Betriebssysteme an der Smartphone-Nutzung in Deutschland von Dezember 2011 bis Juni 2014

1.1 Zielsetzung

Einer der WG-Mitbewohner erklärt sich dazu bereit die Aufgaben des WG-Adminstrators zu übernehmen. Dieser WG-Administrator registriert sich als erster und legt dabei eine neue WG für sich an. Zu seinen Aufgaben gehört unter anderem die Verwaltung der Mitbewohner sowie das Pflegen des Putzplans. Neue Mitbewohner müssen vom WG-Administrator per E-Mail in die WG eingeladen werden. Für den Putzplan definiert er Aufgaben, die in einem einstellbaren Rhythmus wiederholt werden. Zu jeder Aufgabe wird ein Mitbewohner ausgewählt. Nun beginnt der Rhythmus zu laufen und die Aufgaben wechseln nach Erledigung automatisch zu der nächsten Person aus der WG. Auf dem Schwarzen Brett können Einträge angezeigt, erstellt und gelöscht werden. In der Einkaufsliste können Artikel hinzugefügt und entfernt werden. Wurde ein Artikel gekauft, kann derjenige der den Artikel gekauft hat, den Artikel als *gekauft* markieren. Alle Änderungen eines Mitbewohners sind für alle anderen Mitglieder der WG nach einer kurzen Synchronisation sofort sichtbar. Alle Informationen einer WG werden serverseitig in einer Datenbank und clientseitig auf dem Smartphone des Benutzers gespeichert. Bei jedem Start der App wird ein Datenabgleich der auf Client- und Serversei-

te gespeicherten Informationen durchgeführt und alle voneinander abweichenden Daten auf einer Übersichtsseite dem Benutzer als *Neu* aufgelistet.

1.2 Ähnliche Apps

Es gibt bereits eine Auswahl an Apps, die sich jeweils an einem kleinen Teilbereich unseres Funktionsumfanges orientieren und dies gut umsetzen. Hierbei sind einzelne Apps für Einkaufslisten wie *Shopping List* [**?**] oder Putzpläne wie *Roomboard - Cleaning Roster* [**?**] zu nennen. Außerdem gibt es eine weitere App die sich stark an unserer Idee mit ähnlichem Funktionumfang orientiert. Die App *Flatastic: Die WG-App* [**?**] bietet neben einer Einkaufsliste, einem Putzplan und einer Pinnwand zusätzlich einen Ausgabenrechner, womit alle für die WG getätigten Einkäufe zusammengerechnet werden. Wir beschränken uns in dieser Ausarbeitung dennoch weiter auf unseren festgelegten Funktionsumfang und können uns nach der Fertigstellung nach wie vor dazu entscheiden weitere Zusatzfunktionen zu implementieren.

2

Vorgehensweise

Um mögliche Probleme bei der Implementierung der App bereits früh zu identifizieren und den Arbeitsumfang der einzelnen Funktionen besser ermitteln zu können, haben wir vor Beginn der Implementierung die Risiken analysiert und uns für ein geeignetes Vorgehensmodell im Projektmanagement entschieden.

2.1 Projektmanagement

Vor der eigentlichen Implementierung eines Softwareprojektes liegt meist mehr schriftliche Arbeit, die meist mit Stift und Papier zu erledigen ist, als man denkt. Um in einem Team effektiv zu arbeiten, ist es nicht nur wichtig auf das gleiche Ziel hinzuarbeiten, sondern ebenso wichtig ist, gemeinsam auf dem gleichen Weg zum Ziel zu arbeiten. Dafür werden wir uns intern, sowie mit Absprache des Kunden verschiedene Meilensteine setzen. Zu jedem abgeschlossenen Meilenstein, ob nach einem großen oder kleinen Fortschritt, treffen wir uns im Team an einem Tisch und besprechen die Ergebnisse jeder Person um sie am Ende zusammenzutragen. Des Weiteren wird es in Absprache mit dem Kunden nach jedem abgeschlossenen großen Meilenstein ein Treffen bei dem Kunden geben. So können wir uns zu jedem Zeitpunkt des Projektes sicher sein, auf die Wünsche und Vorgaben des Kunden richtig einzugehen. Ausserdem ist der Kunde so direkt mit in die Entwicklung integriert.
Zu Beginn müssen wir für unser Projektmanagement das passende Modell finden. Nach reichlicher Überlegung haben wir uns dafür entschieden mit Use Cases, Mockups und Prototypen das Projekt zu beginnen. Dadurch war schnell mit dem *Prototyping* das passende Modell für unser Vorhaben gefunden. Es gibt verschiedene Arten von *Prototyping*, die alle eine andere Herangehensweise bieten. Bei näherem Betrachten stellt sich heraus, dass das *horizontale Prototyping* genau die Art von *Prototyping* bietet, die wir bevorzugen. Dabei wird eine Ebene des Gesamtsystems möglichst genau entworfen oder implementiert und genau möchten wir mit den Mockups und dem Prototypen umsetzen, lediglich die Use Cases falle ein wenig aus dem Rahmen. Weitere Details finden Sie in den folgenden Unterkapiteln.
Da wir allerdings am Ende ein Lastenheft erstellen möchten und wir während der

Projektzeit Meilensteine definieren und zu Meilensteinsitzungen einladen werden, wird unser Management nicht nur alleine auf dem *horizontalen Prototyping* basieren, sondern zusätzlich Einflüsse aus dem *Wasserfallmodell* beinhalten. Dieses Modell zeichnet sich durch die Organisation in Phasen aus. Dabei werden kleine Meilensteine während einer Phase und große Meilensteine am Ende einer Phase definiert und in Meileinsteinsitzungen die Ergebnisse besprochen. Die wichtigsten Dokumente aus dem *Wasserfallmodell* ist unter anderem das Lastenheft.
Damit haben wir nun als Vorgehensmodell für unser Projektmanagement das *Horizontale Prototyping mit Einflüssen aus dem Wasserfallmodell.*

2.1.1 Use-Cases

Ein Use-Case (dt. Anwendungsfall) schildert auf einfacher Art und Weise alle möglichen Szenarien die zu einem bestimmten Ziel führen können. Am Anfang steht ein Akteur (z.B. der Benutzer der App) der ein bestimmtes Ziel im System erreichen möchte (z.B. das erstellen einer Notiz). Der Use-Case beschreibt nun grob das System, die Umwelt in welchem das System arbeitet, die Anforderungen falls gegeben und letztendlich alle nötigen Schritte, welche vom Benutzer getätigt werden müssen, um das Ziel zu erreichen. Dazu können auch automatisierte Ereignisse vom System gehören.
Inwieweit der Use-Case ins Details geht ist nicht definiert. Der Ersteller kann hier selbst entscheiden wie sehr eine Aktion abstrahiert werden kann. Die Entscheidung kann von Use-Case zu Use-Case variieren. Es empfiehlt sich eine Vorlage mit relevanten Punkten zu erstellen, die dann bei jedem Use-Case möglichst einheitlich ausgefüllt werden kann. Um eine Übrsicht über das gesamte System aller Use Cases zu erhalten, wird eine grafische Modellierung empfohlen. Dabei können die textbasierten Use-Cases in ein Use-Case-Diagramm vereint werden. Hierfür ist in der Softwarentwicklung die Modellierungssprache *Unified Modelling Language* (dt. vereinheitlichte Modellierungssprache, kurz: UML) zu empfehlen. Sie wird für Version 2.4.1 unter anderem von der *International Organization for Standardization* (dt. Internationale Organisation für Normung, kurz: ISO, ISO/IEC 19505) standardisiert.
Ein beispielhafter Auszug aus unseren Use Case Diagrammen und den dazugehörigen Use-Cases finden sie unter *Abb.***??** und *Abb. 2.2.*
Abb. **??** zeigt das Use-Case-Diagramm für die Funktion des Black Boards. Jedes elipsenförmig eingekreiste Schlagwort steht für ein Use-Case. Oft ist die Wahl der Use-Cases identisch zu den der Funktionen, d.h. jede Funktion wird als Use-Case abgebildet.
Am Anfang steht der Benutzer der App, hier dargestellt als ein Strichmännchen am linken Rand des Diagramms. Der Benutzer hat die Wahl zwischen der Funktion *Notiz erstellen* und *Notiz bearbeiten.* Hierbei teilt sich das Diagramm in zwei Stränge auf.
Möchte der Benutzer eine Notiz erstellen wird diese am Ende des Vorgangs in der Datenbank (kurz: DB) gespeichert. Um eine Notiz zu speichern, muss der Benut-

zer vorher seine Eingaben getätigt haben. Wir haben uns hierbei dafür entschieden Die Aktionen so zu minimalisieren, dass solche Aktionen wie *Text eingeben* ect. als nicht relevant und gegeben anzusehen sind. Diese Aktionen fallen in den Use-Case *Notiz erstellen*.
Möchte der Benutzer eine bestehende Notiz bearbeiten, so hat er dabei die Möglichkeit, die Notiz zu löschen oder mit seinen Änderungen zu speichern.
Die an jeden Use-Case mit extends angefügte Fehlerbehandlung wird vom System automatisch ausgeführt, sobald ein Fehler in einem der Use-Caes auftritt. Der Fehler kann dabei durch den Benutzer, z.B. durch falsche Eingaben, sowie vom System, z.B. durch eine fehlgeschlagene Datenbankverbidnung, ausgelöst werden.

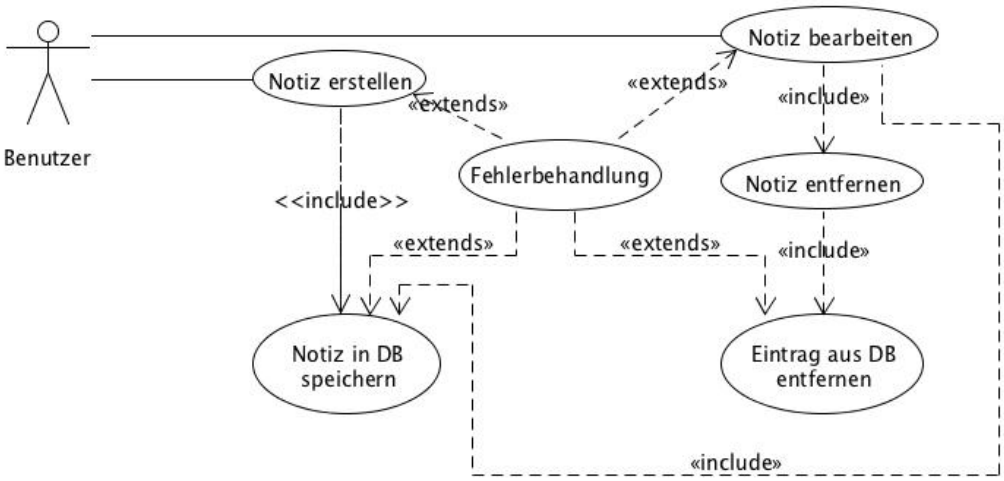

Abb. 2.1. Use-Case-Diagramm für das Blackboard

Betrachtet man das oben eingefügte Use-Case-Diagramm *Abb. 2.1* wird einem vermutlich auffallen, dass das Use-Case *Notiz bearbeiten* komplexer erscheint als die anderen, jedoch genauso abstrakt dargestellt wird. Bei solchen Use-Cases, bei denen vermutlich nicht sofort jedem klar ist, was im Hintergrund passiert, empfiehlt es sich eine Use-Case Beschreibung in Textform zu erstellen. Für den Fall *Notiz bearbeiten* sieht unsere Beschreibung wie folgt aus.
Es wird klar definiert, dass das Ziel des Use Cases eine unsortierte Liste von Notizen sein soll. ein Ziel muss immer klar definiert werden. Ebenso werden *Vor-* und *Nachbedingungen* aufgeführt, falls diese vorhanden sind. Unter *Akteure* wird aufgezählt, wer in diesen Use-Case gelangen kann. In unserem Fall sind es alle Benutzer unserer App. Da man die App nur als registriertes Mitglied einer WG benutzen kann, wird in den Klammern nochmals darauf hingewiesen. Die *Beschreibung* schildert in Stichworten den Ablauf des Use-Cases, d.h. man bescheibt Punkt für Punkt die Ereignisse die vom Benutzer sowie vom System unternommen werden. Mit *Alternativen* beschreibt man Ereignisse, die durch eine Verzweigung im Ablauf vorkommen können. Diese müssen bereits jetzt berücksichtigt werden.

```
1  Use-Case: Notiz bearbeiten
2  Ziel: Unsortierte Liste von Notizen angezeigt
3  Kategorie: optional (nützlich, nicht unbedingt notwendig)
4  Vorbedingung: Benutzer ist eingeloggt (in App)
5  Nachbedingung Erfolg: Mitteilung an Benutzer, dass Eintrag erfolgreich
6  Nachbedingung Fehlschlag: Mitteilung an Benutzer, dass Eintrag fehlgeschlagen
7  Akteure: Benutzer der App (nur registrierte WG Mitglieder)
8  Auslösendes Ereignis: Erstellung eines Eintrags von Benutzer
9  Beschreibung:
10 Notiz erstellen
11 1 Benutzer nimmt Änderungen an Notiz vor.
12 2 Benutzer möchte Notiz speichern.
13 3. Notiz in Datenbank speichern.
14 Alternativen:
15 2a Benutzer entschließ sich die Änderungen zu verwerfen. Benutzer gelangt auf
       vorherigen Bildschirm
16 2b Benutzer entschließt sich die Notiz zu löschen. Der Datenbankeintrag wird
       gelöscht.
```

Abb. 2.2. Beschreibung des Use-Cases *Notiz bearbeiten*

2.1.2 Mockups

Ein Mockup ist eine nicht lauffähige Version einer Ebene des Gesamtsystems. Sie dient hauptsächlich zur ersten Veranschaulichung designtechnischer Visionen. Der Begriff des Mockups wird auch ausserhalb der Softwareentwicklung gerne für maßstabsgetreue plastisch modellierte Nachbildungen verwendet. In der Softwarentwicklung versteht man meistens eine in Bildbearbeitungsprogrammen designte Vorlage der grafischen Benutzeroberfläche, die beispielhaft das finale Produkt repräsentieren soll. Im späteren Verlauf wird oft auf Basis der Mockups ein Screendesign entwickelt, in dem weitaus mehr Details ausgearbeitet werden. Das Screendesign dient zum Ende des Projekts als Vorlage für den Designer, der sich bei der Gestaltung der grafischen Benutzeroberfläche an den Vorlagen des Screendesigns orientieren muss.
Auch hier betrachten wir uns ein Beispiel aus unseren angefertigten Mockups. Wir wählen hierfür bewusst die Mockups für das Black Board, um zusammen mit den unter dem Kapitel *2.1.1 Use-Cases* in *Abb. 2.1* und *Abb. 2.2* beschriebenen Dokumenten einen möglichst vollständigen Einblick in eine Funktion der App zu erhalten.
Für die Mockups wurde das kostenfreie Programm *Pencil* [1] von Entwickler *Evolus* [2] verwendet. Das Programm hilft speziell bei der Gestaltung von Mockups für die Softwareentwicklung. Es bringt bereits viele vorgefertigte Formen und Icons aus verschiedenen Systemen mit. So ist man sofort nach der Installation bereits in der Lage Mockups basierend auf Android Icons zu erstellen. Diese Vorteil hat uns überzeugt und uns deshalb zu *Pencil* greifen lassen.
Unter Berücksichtigung unseres Use-Case-Diagramms Black Board aus *Abb. 2.1* benötigen wir drei Mockups um die relevanten Screens des Black Boards darzustellen. So wäre das ein Mockup für die Übersicht des Systems, eins für *Notiz*

[1] „Pencil Project“ Homepage
[2] „Evolus“ Homepage

erstellen und eins für *Notiz bearbeiten/Notiz löschen.*

Abb. 2.3. Mockup 01: Übersicht Black Board

In Abb. 2.3 ist die Übersicht zu sehen. Dies ist der Startbildschirm für das Black Board. Hier werden alle Notizen in einer Liste angezeigt. Zu jeder Notiz gehört ein Benutzer mit dessen Profilbild, sowie ein Datum mit Uhrzeit und dem Text des Benutzers. Die Navigation befindet sich im Kopfbereich der App und ist für den Benutzer immer zu sehen. Im unteren Bereich des Bildschirm befinden sich die speziellen Optionen des Black Boards. Man wird mit einem Fingertipp auf das Textfeld eine neue Notiz erstellen können und mit einem Fingertipp auf *Löschen* die gewünschten Notizen aus der Datenbank entfernen.

Abb. 2.4. Mockup 02: Neue Notiz

Will der Benutzer eine neue Notiz erstellen, so muss er von der Übersicht auf das Textfeld *neue Notiz* tippen. Dann gelangt er auf das Mockup Abb. 2.4, auf dem sich eine Tastatur geöffnet und der Button *Löschen* verschwunden ist. Der Button zum löschen von Notizen würde hier nur verwirren, weil die Funktion nur in der Übersicht Möglich ist. Der Button *Zurücksetzen* ist hier allerdings sehr wichtig, um dem Benutzer die Möglichkeit zu geben seinen Text mit einem Tipp auf den Button zu entfernen und gleichzeitig auf die Übersicht zurück zu gelangen. Ohne diese Funktion würde der Benutzer den Text über die Tastatur löschen müssen und könnte schnell genervt davon sein.

Abb. 2.5. Mockup 03: Notiz(en) löschen

Möchte der Benutzer eine Notiz löschen, kann er aus der Übersicht mit dem Fingertipp auf den Button *Löschen* diese Funktion aktivieren (Abb. 2.5). Ist er in dieser Funktion, hat er die Möglichkeit mehrere Notizen auszuwählen und mit einem weiteren tippen auf *Löschen* seine Auswahl aus der Datenbank zu löschen.

Man erkennt, dass man sich mit dem Design der grafischen Benutzeroberfläche an gewisse Vorlagen des Betriebssystems *Android* halten möchte. Ausserdem wird mit einem schlichten und wenig verspielten Design eine gute Lesbarkeit gewährleistet. Auf ein späteres verfeinern der Mockups durch ein Screendesign wurde bewusst verzichtet. Durch die bereits in den Mockups vorkommenden Details und farblichen Designaspekte ist eine weitere Verfeinerung in unseren Augen nicht nötig.

2.1.3 Lastenheft

Ein Lastenheft beschreibt alle Anforderungen die der Auftraggeber an ein Projekt stellt. Es wird vom Auftraggeber erstellt und an den Auftragnehmer weitergeleitet.

Der Auftragnehmer erstellt auf der Grundlage des Lastenhefts ein Pflichtenheft, in dem er seine eigenen Lösungsansätze für die Anforderungen aus dem Lastenheft dem Auftraggeber präsentiert. Ist der Auftraggeber mit dem Pflichtenheft zufrieden, entsteht auf Grundlage dieser beiden Dokumente ein Vertrag zwischen beiden Parteien.Bei der finalen Abgabe des fertiges Produktes an den Auftraggeber, wird das Produkt auf Grundlage der Dokumente auf Vollständigkeit geprüft. Kommt es zu Auseinandersetzungen, können beide Parteien auf den Grundlagen der Dokumente gegeneinander argumentieren.
Da wir als Team Auftraggeber sowie Auftragnehmer sind, schreiben wir uns selbst ein Lastenheft. Dies ist uns wichtig, um während der Implementierung der App den Umfang und den Aufwand im Auge zu behalten. Wir können uns mit einem Lastenheft von Punkt zu Punkt arbeiten und bei der finalen Abgabe gut das Produkt mit den Anforderungen vergleichen. Auf ein Pflichtenheft werden wir bewusst verzichten.

1. Einleitung ... 3
1.1 Allgemeines ... 3
1.2 Verteiler und Freigabe ... 3
1.3 Reviewvermerke und Meeting-Protokolle ... 3
2. Konzept und Rahmenbedingungen ... 4
2.1 Ziele des Anbieters ... 4
2.2 Ziele und Nutzen des Anwenders ... 4
2.3 Benutzer / Zielgruppe ... 4
2.4 Systemvoraussetzungen ... 4
2.5 Nutzervoraussetzungen ... 4
2.6 Ressourcen ... 4
3. Anforderungen ... 5
3.1 Blackboard ... 5
3.2 Putzplan ... 7
3.3 Einkaufsliste ... 8
3.4 Kalender ... 10
3.5 Web Login ... 12
3.6 Web Oberfläche ... 14
4. Freigabe / Genehmigung ... 16
5. Anhang / Ressourcen ... 17
5.1 Offizielles Meeting-Protokoll „WG-APP“ Nr.1 14.05.2014 ... 17
5.2 Offizielles Meeting-Protokoll „WG-APP“ Nr.2 29.09.2014 ... 18
5.3 Offizielles Meeting-Protokoll „WG-APP“ Nr.3 08.10.2014 ... 19

Abb. 2.6. Inhaltsverzeichnis unseres Lastenhefts

In Abb. 2.6 ist das Inhaltsverzeichnis unseres Lastenhefts zu sehen und zeigt die Struktur unseres Dokuments. Wir konzentrierten uns dabei auf das größte Kapitel, die *Anforderungen*. Dieses Kapitel beschreibt alle primären Funktionen der App. Wir haben uns dabei für *Blackboard*, *Putzplan*, *Einkaufsliste*, *Kalender*, *Web Login* und *Web Oberfläche* entschieden. Jedes dieser Funktionen wird zuerst mit einem kurzen Text beschrieben, um danach die Anforderungen und am Ende die Risiken

zu beschreiben.

Das gesamte Lastenheft, sowie alle Mockups, Use-Case-Diagramme und Use-Case Beschreibungen können im Anhang dieser Dokumentation nachgelesen werden.

3

Anforderungen

In der Softwareentwicklung erläutern Anforderungen alle Funktionen eines Softwareprojekts, die ein Auftraggeber vom fertigen Produkt erwartet. Spezifiziert werden die Anforderungen im Lastenheft, welches direkt vom Auftraggeber an den Auftragnehmer verteilt wird.
Die Anforderungen werden für gewöhnlich in die zwei Kategorien funktional und nichtfunktional unterteilt.

Funktionale Anforderungen *beschreiben die Merkmale einer Funktion, also was das Produkt tun soll.*
Nichtfunktionale Anforderungen *beschreiben die Eigenschaften des Produktes, also we das Produkt etwas tun soll.*

Als Beispiel nehmen wir wieder unser bekanntes Blackboard, welches wir bereits detailiert in den Kapiteln 2.1.1 Use-Cases sowie 2.1.2 Mockups besprochen haben. Bei unserem Blackboard haben wir die drei funktionalen Anforderungen *Notiz hinzufügen*, *Notiz bearbeiten*, *Notiz löschen* und die eine nichtfunktionale Anforderung *Desgin*. Betrachten wir uns die funktionale Anforderung *Notiz hinzufügen* in Abb. 3.1.

3. Anforderungen

3.1 Blackboard

Das Blackboard soll zum einfachen Informationsaustausch zwischen den Bewohnern dienen. Dazu soll es einer echten Pinnwand nachempfunden werden.

3.1.1 Anforderung F1 - Notiz hinzufügen

Nr. / ID	1	Nichttechnischer Titel	Notiz hinzufügen		
Quelle	Use Case Blackboard	Verweise	NF4	Priorität	hoch

Beschreibung

- Benutzer gibt Notiz ein
- Benutzer drückt auf „Speichern“
- Notiz wird in lokaler Datenbank gespeichert
- DB-Synchronisierung im Hintergrund

Risiken

- Benutzer bricht Vorgang ab
- Speichern nicht möglich
- Synchronisieren nicht möglich

Abb. 3.1. Anforderung F1 - Notiz hinzufügen (Blackboard)

Zu Beginn wird die Eigenschaft jeder Anforderung in ein bis zwei kurzen Sätzen erläutert. Daraufhin folgt eine Tabelle mit charakteristischen Merkmalen der Anforderung. Jeder Anforderung wird eine eindeutige, fortlaufende ID zugeteilt. Außerdem wird ein nichttechnischer Titel vergeben. Als Quelle der Anforderung dient ein anderes Dokument der Projektarbeit, im besten Fall bereits verfügbar und für alle abrufbar. Meistens wird ein Dokument aus den Mockups oder Use-Cases als Quelle herangezogen. Die Verweise geben alle weiterführenden Dokumente an, in denen die Anforderung weiter ausgeführt wird oder welche, die auf die Zusammenarbeit mit dieser Anforderung unverzichtbar sind. Letztendliche kann noch eine Priorität vergeben werden. Wir haben uns, mit Ausnahmen, bei vielen unserer funktionalen Anforderungen für eine hohe Priorität und bei vielen nichtfunktionalen Anforderungen für eine normale Priorität entschieden.
Nach der Tabelle folgt eine stichwortartige Beschreibung des grundsätzlich vorgesehenen Ablaufes. Hierbei wird dargestellt, wie das System sowie der Benutzer agieren muss und welche Konsequenz zu erwarten sind. Zuletzt werden alle bekannten Risiken aufgezählt und stichwortartig beschrieben.

3.1 Anforderungen für DorMApp

Alle funktionalen Anforderungen werden als *Anforderung F#* und alle nichtfunktionalen Anforderungen werden als *Anforderung NF#* gekennzeichnet.
Es folgen nun alle Anforderung für unsere App, sortiert nach den Hauptfunktionen.

- **Blackboard**
  - *Anforderung F1 - Notiz hinzufügen*
  - *Anforderung F2 - Notiz bearbeiten*
  - *Anforderung F3 - Notiz löschen*
  - *Anforderung NF4 - Design*
- **Putzplan**
  - *Anforderung F5 - Aufgabe erledigen*
  - *Anforderung NF6 - Design*
- **Einkaufsliste**
  - *Anforderung F7 - Artikel hinzufügen*
  - *Anforderung F8 - Artikel löschen*
  - *Anforderung F9 - Artikel gekauft*
  - *Anforderung NF10 - Design*
- **Kalender**
  - *Anforderung F11 - Kalendereintrag eintragen*
  - *Anforderung F12 - Kalendereintrag bearbeiten*
  - *Anforderung F13 - Kalendereintrag löschen*
  - *Anforderung NF14 - Design*
- **Web Login**
  - *Anforderung F15 - Web Login*
  - *Anforderung F16 - Eingabekontrolle*
  - *Anforderung F17 - Passwort vergessen*
  - *Anforderung NF18 - Design*
- **Web Oberfläche**
  - *Anforderung F19 - Benutzerverwaltung*
  - *Anforderung F20 - Terminkalenderverwaltung*
  - *Anforderung F21 - Putzplanverwaltung*
  - *Anforderung F22 - Systemeinstellungen*
  - *Anforderung NF23 - Design*

3.2 Risiken

Risiken können den Ablauf in einer Softwarearchitektur erheblich beeinflussen. Jedes Risiko, welche vor der Implementierung nicht berücksichtigt wurde, kann beim

Benutzer des Systems ein unkontrollierbares Verhalten verursachen. Um dies bestmöglich zu vermeiden haben wir uns bereits frühzeitig Gedanken um Mögliche Risiken gemacht. In einem Brainstorming kamen so eine Menge an Risikopunkten zusammen, die zu einem späteren Zeitpunkt Probleme machen könnten. In unserem Lastenheft **??**, werden zu jeder Anforderung alle Risiken aufgelistet.

- **Blackboard**
  - *Anforderung F1 - Notiz hinzufügen*
    - Benutzer bricht Vorgang ab
    - Speichern nicht möglich
    - Synchronisieren nicht möglich
  - *Anforderung F2 - Notiz bearbeiten*
    - Benutzer bricht Vorgang ab
    - Speichern nicht möglich
    - Synchronisieren nicht möglich
  - *Anforderung F3 - Notiz löschen*
    - Ungewolltes Löschen
  - *Anforderung NF4 - Design*
    - Unübersichtliches Layout
    - Missverständliche Abläufe

- **Putzplan**
  - *Anforderung F5 - Aufgabe erledigen*
    - Fehlerhafte Synchronisierung
    - Falsche Aufgabe ausgewählt
  - *Anforderung NF6 - Design*
    - Namen des Verantwortlichen zu Lang und/oder in nicht darstellbarem Zeichensatz
    - Rhythmus funktioniert nicht wie Benutzer es erwartet

- **Einkaufsliste**
  - *Anforderung F7 - Artikel hinzufügen*
    - Fehler beim Speichern in die Datenbank
    - Der Artikel steht bereits in der Einkaufsliste, allerdings mit anderer Buchstabenformatierung (Groß- und Kleinschreibung, Bindestrich, etc.)
  - *Anforderung F8 - Artikel löschen*
    - Fehler beim Speichern in die Datenbank

        - Der Artikel existiert bereits nicht mehr in der Datenbank, wird aber in der Applikation auf dem Endgerät noch angezeigt.

    - *Anforderung F9 - Artikel gekauft*
        - Fehler beim speichern in die Datenbank
        - Der Artikel existiert bereits nicht mehr in der Datenbank, wird aber in der Applikation auf dem Endgerät noch angezeigt.

    - *Anforderung NF10 - Design*
        - Sortierung der Artikel
        - Unübersichtliche Aufzählung
        - Schlecht leserliche Schlichtart und Schriftgröße

- **Kalender**
    - *Anforderung F11 - Kalendereintrag eintragen*
        - Fehler beim Speichern in die Datenbank
        - Titel ist zu lang
        - Datum liegt in der Vergangenheit
        - Datum liegt weit in der Zukunft (>50 Jahre)

    - *Anforderung F12 - Kalendereintrag bearbeiten*
        - Fehler beim Speichern in die Datenbank
        - Titel ist zu lang
        - Datum liegt in der Vergangenheit
        - Datum liegt weit in der Zukunft (>50 Jahre)

    - *Anforderung F13 - Kalendereintrag löschen*
        - Fehler beim speichern in die Datenbank
        - Kalendereintrag existiert nicht mehr in Datenbank, wird aber noch auf dem Endgerät angezeigt

    - *Anforderung NF14 - Design*
        - Benutzer benötigt zu lange um sich mit dem Design zurecht zu finden
        - Zu wenig Platz um alle Kalendereintragungen anzuzeigen
        - Titel zu lang für Gesamtansicht

- **Web Login**
    - *Anforderung F15 - Web Login*
        - E-Mailadresse existiert nicht
        - E-Mailadresse und/oder Passwort falsch
        - Passwort vergessen
        - Verbindung zum Server kann nicht hergestellt werden

  - *Anforderung F16 - Eingabekontrolle*
    - *Eingabe falscher Daten*

  - *Anforderung F17 - Passwort vergessen*
    - Missbrauch von Daten

  - *Anforderung NF18 - Design*
    - Design Gesetze nicht eingehalten (Gesetz der Nähe, Gleichheit, Geschlossenheit, etc.)
    - Größe der Textfelder nicht optimal

- **Web Oberfläche**
  - *Anforderung F19 - Benutzerverwaltung*
    - Datenbankverbindung schlägt fehl

  - *Anforderung F20 - Terminkalenderverwaltung*
    - Datenbankverbindung

  - *Anforderung F21 - Putzplanverwaltung*
    - Datenbankverbindung

  - *Anforderung F22 - Systemeinstellungen*
    - Datenbankverbindung

  - *Anforderung NF23 - Design*
    - Unübersichtlich

Die mit Abstand anfälligste Funktion unserer App wird die Verbindung zur Datenbank auf einem externen Server sein. Dabei kann zu jedem Zeitpunkt ein Fehler auftreten, der vom System möglichst gut behandelt werden soll. Darum hat die Verbindung zur Datenbank mehr Aufmerksamkeit bekommen und wir haben uns dafür entschieden einen lauffähigen Prototypen zu entwickeln. Wir haben uns bereits jetzt so stark mit der Mechanik beschäftigt, um anstelle eines einfachen Wegwerf-Prototypen, einen im späteren Verlauf der Implementierung weiter zu verwendenden Prototypen entwerfen können. Folgende Risiken können bei unserer Arbeit mit einer extern erreichbaren Datenbank auftreten:

- Server nicht erreichbar
- Verbindungsabbruch durch Zusammenbruch der Internetverbindung
- Benutzer killt die App während der aktiven Nutzung der Verbindung

Eine genaue Erläuterung der Implementierung des gesamten Prototypen finden Sie im folgenden Kapitel 4 Prototyp.

4

Prototyp

Um vermeidbare Verzögerungen in der späteren Entwicklung möglichst auszuschließen, wurde der Einsatz von Prototypen entschieden. Dazu werden die benötigten Komponenten genauer betrachtet und auf ihre Umsetzbarkeit hin untersucht. Bei einer Android-App, die auf Inhalte aus einer Datenbank zugreift, ist das die Schnittstelle zur Datenbank, beziehungsweise die Netzwerkverbindung.
Da eine direkte Verbindung zum Datenbankserver aus dem Android Betriebssystem nicht möglich ist, wurde die Authentifizierung des Benutzers als weitere Problemstelle markiert. Den die Verwendung von Datenbankbenutzern fällt dadurch weg und muss andersweitig gelöst werden. Um bei diesen kritischen Stellen nicht in bredouille zu geraten, wurde dafür ein Prototyp entwicklt.

4.1 Prototyp Implementierung

Als eine einfache und schnelle Art der Umsetzung haben wir uns für die PHP-Variante entschieden. Dabei werden die Daten von einem PHP-Skript aus der Datenbank gelesen und in eine JSON-Datenformat gebracht. Dieses Objekt wird in einem HTTP-Paket an die App übertragen. In der App sorgt dann ein JSON-Parser für das Auslesen der Daten, welche anschließend direkt verwertet werden oder zunächst in der lokalen SQLite-Datenbank vorgehalten werden.
Alternativ zu dieser Lösung wäre ein RESTful Web Service gewesen. Dieser Lösungsansatz wäre jedoch mit einem größeren programmiertechnischen Aufwand, sowie einer umfangreicheren Serverkonfiguration verbunden gewesen.
Da beide Schwerpunkte in einem Prototyp getestet wurden, wird auf eine weitere Trennung verzichtet.

4.1.1 Datenbankstruktur

Begonnen wurde die Umsetzung mit der Definition der Datenbankstruktur sowie deren Umsetzung. Dazu wurden zwei einfache Tabellen angelegt. Die Tabelle `tp_test`4.1 wird für die Schreib- und Lesevorgänge verwendet. Um alle nötigen Datentypen testen zu können wurden verschiedene Spalten verwendet. Dadurch konnte auch der spätere Einsatz besser simuliert werden.

```
1  CREATE TABLE IF NOT EXISTS `test_tp` (
2      `uid` VARCHAR(23) NOT NULL,
3      `msg` TEXT,
4      `nmbr` INT(11) DEFAULT NULL,
5      `created_at` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
6  ) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Abb. 4.1. Aufbau der Tabelle tp_test

Um die Benutzerverwaltung für den Prototypen zu simulieren wurden außerdem noch eine Tabelle `users` 4.2 angelegt. Darin wurden Informationen zu den Benutzern hinterlegt. Zum Beispiel eine eindeutige ID, sowie Vor- und Nachname. Zum Authentifizieren wurde die E-Mailadresse und ein beliebiges Passwort verwendet. Um das Passwort nicht im Klartext zu speichern, wurde es zusammen mit einem Salt als Hash-Wert abgelegt.

```
1  CREATE TABLE IF NOT EXISTS `users` (
2    `uid` int(11) NOT NULL AUTO_INCREMENT,
3    `unique_id` varchar(23) NOT NULL,
4    `firstname` varchar(50) NOT NULL,
5    `lastname` varchar(50) NOT NULL,
6    `username` varchar(20) NOT NULL,
7    `wgId` int(11) NOT NULL,
8    `email` varchar(100) NOT NULL,
9    `encrypted_password` varchar(80) NOT NULL,
10   `salt` varchar(10) NOT NULL,
11   `created_at` datetime DEFAULT NULL,
12   PRIMARY KEY (`uid`)
13 ) ENGINE=MyISAM  DEFAULT CHARSET=utf8 AUTO_INCREMENT=5 ;
```

Abb. 4.2. Aufbau der Tabelle users

4.1.2 PHP-Skripte

Der Aufbau der PHP-Schnittstelle ist simpel umgesetzt, da nicht viele Funktionen für den Prototyp benötigt werden. Trotzdem wurde auf eine übersichtliche Datei- und Ordnerstruktur, als auch auf einen modularen Aufbau geachtet. Wie in Abbildung 4.3 Struktur der PHP-Skripte im Dateisystem zu sehen, wurden der Aufbau zunächst in zwei Kategorien aufgeteilt. Funktionen, die direkt auf der Datenbank ausgeführt werden, sowie das Verbinden und Bereitstellen des Datenbankobjekts

übernehmen, sind im Ordner `php/include` untergebracht. Alle weiteren Funktionen die aus der App heraus erreichbar sein sollen, befinden sich im Hauptordner `php`.

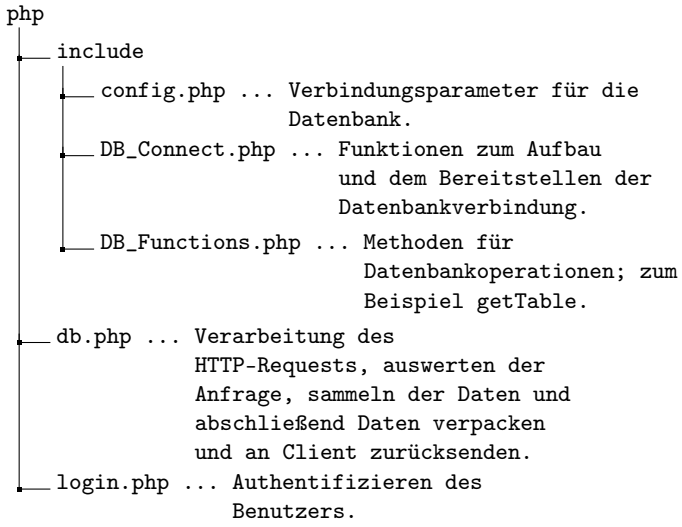

Abb. 4.3. Struktur der PHP-Skripte im Dateisystem

Die vollständigen Skripte befinden sich im Anhang und werde hier nur in Auszügen dargestellt.
Der Ablauf zum Aufrufen der einzelnen PHP-Funktionen ist immer derselbe. Dazu wird ein POST-Request an den Server geschickt, der die entsprechende Ressource, in diesem Fall entweder `db.php` oder `login.php` anfordert. Als POST-Parameter werden neben den Werten für die Funktion, zum Beispiel E-Mailadresse und Passwort für den Login, noch ein TAG-Parameter angehängt. Der TAG-Parameter ist für den Login-Prozess zum Beispiel leer, zum Ändern des Passworts kann dazu `chgpass` eingesetzt werden. Nachdem die zum TAG passende Funktion gefunden wurde, werden die übertragenen Werte ausgelesen und entsprechend verarbeitet.
Eine wichtige Funktion der PHP-Skripte ist der Login, beziehungsweise die Authentifizierung des Benutzers.
Dazu wird die Adresse `http://test.app1.raschel.org/php/db.php` mit den Parametern `TAG=''`, `EMAIL=‘‘ritzels@fh-trier.de’’` und `PASSWORD=‘‘ritzels’’` aufgerufen (Vgl. 4.4). Sind alle Parameter korrekt übertragen worden, wird die Funktion `getUserByEmailAndPassword($email, $password)`, die sich im Skript `DB\_Functions.php` befindet, aufgerufen. Zum Authentifizieren wird aus der Datenbank der zur E-Mailadresse passende Datensatz geladen, siehe Auszug 4.5. Das in der Datenbank gespeicherte `SALT` wird mit dem übertragenen Passwort an die Funktion `checkhashSSHA($salt, $password)` übergeben, welche den Hash aus `SALT` und Passwort bildet und zurückgibt. Dieser generierte Hash wird dann mit dem in der Datenbank gespeichertem `encrypted\_password` verglichen. Stimmen die beiden Hashs überein, wird der geladene Datensatz an die Login-Funktion zurückgeliefert, ansonsten liefert die Funktion `getUserByEmailAndPassword() -> false`.

```php
<?php
  /**
   PHP API for Login, Register, Changepassword,
   Resetpassword Requests and for Email Notifications.
 **/
  [...]
  if (isset($tag) && $tag != '') {
  // Include Database handler
  require_once 'include/DB_Functions.php';
  $db = new DB_Functions();
  // response Array
  $response = array("tag" => $tag, "success" => 0, "error" => 0);
  // check for tag type
  if ($tag == 'login') {
  // Request type is check Login
  $email = $_POST['email'];
  $password = $_POST['password'];
  // check for user
  $user = $db->getUserByEmailAndPassword($email, $password);
    if ($user) {
      // user found
      // echo json with success = 1
      $response["success"] = 1;
      $response["user"]["user_id"] = $user["user_id"];
      $response["user"]["wg_id"] = $user["wg_id"];
      $response["user"]["crea"] = $user["crea"];

      echo json_encode($response);
    } else {
      // user not found
      // echo json with error = 1
      $response["error"] = 1;
      $response["error_msg"] = $IncorrectEMail;
      echo json_encode($response);
    }
  }
  else if ($tag == 'chgpass') {
  [...]
```

Abb. 4.4. Auszug aus login.php

Die Skripte befinden sich in vollem Umfang noch als Anhang an diese Arbeit.

```
1       [..]
2       private $db;
3
4      // constructor
5      function __construct() {
6          require_once 'DB_Connect.php';
7          // connecting to database
8          $this->db = new DB_Connect();
9          $this->db->connect();
10     }
11       /**
12       * Verifies user by email and password
13       */
14      public function getUserByEmailAndPassword($email, $password) {
15          $resultSQL = mysql_query("SELECT * FROM user WHERE email = '$email'")
16              or die(mysql_error());
17          // check for result
18          $no_of_rows = mysql_num_rows($resultSQL);
19
20          if ($no_of_rows > 0) {
21              $result = mysql_fetch_array($resultSQL);
22              $salt = $result['salt'];
23              $encrypted_password = $result['passwort'];
24              $hash = $this->checkhashSSHA($salt, $password);
25              // check for password equality
26              if ($encrypted_password == $hash) {
27                  // user authentication details are correct
28                  return $result;
29              }
30          }
31          else {
32              // user not found
33              return null;
34          }
35      }
36      [..]
```

Abb. 4.5. Auszug aus DB_Functions.php

4.1.3 Prototyp-App

Die Implementierung der Prototyp-App wurde in zwei Projekte aufgeteilt. In dem Projekt `DatabaseConnectionLib` wurden die Funktionen zum Schreiben und Lesen der MySQL-Datenbank ausgelagert, da diese mit Sicherheit für die spätere Implementierung wieder Verwendung finden werden. Das Anzeigen und Manipulieren der Daten wurde im Projekt `DBPrototyp` zusammengefasst.

DBPrototyp

Das Projekt wurde noch in der Entwicklungsumgebung Eclipse erstellt und umfasst deshalb eine etwas tiefere Ordnerhirarchie. Da nicht alle für diese Implementierung relevant sind, werden nur die wichtigsten Ordner und Dateien näher erläutert.
Der Ordnerbaum 4.6 zeigt die wichtigen Ordner und Dateien, und erläutert kurz deren Funktion.

Für die Funktion des Prototyp und das Umsetzen der definierten Schwerpunkte sind in diesem Projekt die beiden Klassen `LoginActivity.java` und `MainActivity.java` von Bedeutung. Beim Start der App wird zunächst die

```
DBPrototyp
├── src
│   └── de.hst.teamprojekt.dbprototype
│       ├── LoginActivity.java ... Java-Klasse mit den Funktionen zum Login.
│       └── MainActivity.java ... Java-Klasse mit den Funktionen zum Lesen und
│                                  Schreiben der Daten.
├── res
│   ├── layout
│   │   ├── activity_login.xml ... Layout-Datei für die Login-Oberfläche.
│   │   └── activity_main.xml ... Layout-Datei für die Standard-Oberfläche.
│   ├── menu ... Beinhaltet die Menüstruktur der einzelnen
│   │            Oberflächen.
│   └── values ... Beinhaltet XML-Dateien, in denen Werte
│                  hinterlegt werden können die nicht im Quellcode
│                  sein sollten.
└── AndroidManifest.xml ... Allgmeine Parameter der App, wie unterstützte
                            API-Version oder benötigte Ressourcen, bzw.
                            Erlaubnis für die Ressourcen.
```

Abb. 4.6. Verzeichnisstruktur des DBPrototyp-Projekts

`MainActivity.java` mit dem Layout `activity_main.xml` geladen 4.7.
Das geschieht in Zeile 5 durch `setContentView(R.layout.activity_main)` in der `onCreate()` Methode.

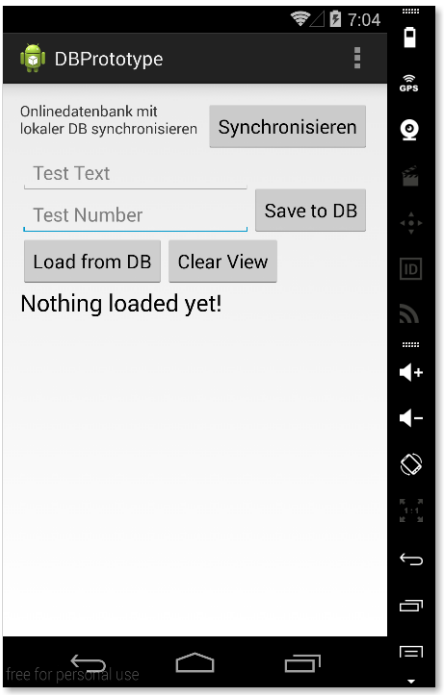

Abb. 4.7. Start-Oberfläche

Ist alles vollständig geladen, kann man sich über das Menü anmelden. Dafür wird in der `onOptionsItemSelected(MenuItem item)` ein `startActivity()` aufgerufen, welches die `LoginActivity.java` startet 4.8.

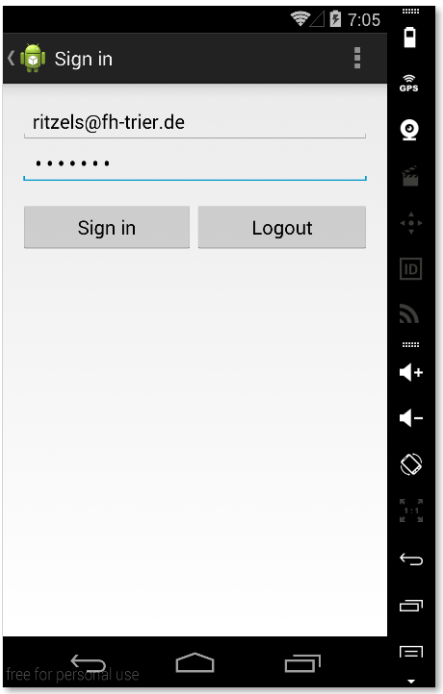

Abb. 4.8. Login-Oberfläche

Nach der erfolgreichen Authentifizierung wird über den Button `Synchronisieren` die Funktion `synchronizeDbs(View view)` 7 aufgerufen. Dies generiert ein Objekt der Klasse `SyncRemoteDatabase.java` 4.11. Dabei handelt es sich um eine von `AsyncTask` abgeleitete Klasse, welche die Testdaten über das PHP-Skript aus der Datenbank abruft und anschließend in die lokale `SQLite`-Datenbank schreibt. Diese Klasse wurde jedoch in die `DBConnectionLib` 4.1.3 ausgelagert und anschließend an diese Kapitel genauer erläutert.

Wurden die Tabellen erfolgreich synchronisiert, kann über die beiden Felder Testwerte eingegeben werden. Durch einen klick auf den Button `Save to DB` werden die Daten zunächst in der Methode `saveToDb(View view)` 2 in *BasicNameValuePair*-Objekte überführt (Zeile 5 - 9). Zum Speichern werden diese Objekte an eine Instanz der Klasse `InsertIntoDatabase.java` übergeben. Diese ist, wie die Klasse `SyncRemoteDatabase.java`, eine Subklasse von `AsyncTask` und wird ebenfalls im Kapitel 4.1.3 genauer erklärt. Zum Anzeigen der lokalen Tabelleninhalte kann der Button `Load from DB` betätigt werden, wodurch die Funktion `loadFromDb(View view)` ausgeführt wird. Dabei werden die Daten zunächst in einem *Cursor*-Objekt bereitgestellt, welches vom `DatabaseHandler`4.14 bereitgestellt wird (vgl. Zeile 17). Die Darstellung wird durch eine *ListView* übernommen, die im Layout der Aktivität hinterlegt und über den Befehl //*

```
1  [...]
2  @Override
3  protected void onCreate(Bundle savedInstanceState) {
4    super.onCreate(savedInstanceState);
5    setContentView(R.layout.activity_main);
6  [...]
7  public void synchronizeDbs(View view) {
8    {...}
9    SyncRemoteDatabase queryTask =
10     new SyncRemoteDatabase(MainActivity.this, this.creds);
11   queryTask.execute(Constants.TABLE_TEST);
12 }
13 [...]
```

Abb. 4.9. Auszug aus der MainActivity.java

`findViewById(R.id.listView1)` in Zeile 29 referenziert wurde. Um den *Cursor* in der *ListView* anzeigen zu können, muss dieser in einem *SimpleCursorAdapter* für die *ListView* zugänglich gemacht werden. Dazu wird in Zeile 28 dem Konstruktor des *SimpleCursorAdapter* zunächst der aktuelle *Context*, das gewünschte Layout der einzelnen Zeilen (`R.Layout.list_item`) sowie der *Cursor* `result` übergeben. Damit der Adapter weiß, welche Werte er in welche *TextView* packen soll, werden dem Konstruktor noch `from` und `to` übergeben.

Dabei handelt es sich bei dem Objekt `from` um eine *String*-Liste mit den Namen der Tabellenspalten. Die *int*-Liste `to` enthält die IDs der *TextViews* im verwendeten Zeilenlayout. Die Reihenfolge muss dabei beachtet werden. Abschließend erhält der Konstruktor noch eine Flag mit der das Verhalten bei Änderung der Datengrundlage gesteuert wird. Dieser Adapter wir mittels dem Befehl `.setAdapter(sca)` der *ListView* übergeben. Wodurch nun die Daten auf der Oberfläche sichtbar werden.

Zum Leeren der *ListView* wird der Button `Clear View` gedrückt, wodurch mittels der Methode *clearResultListView(View view)* 34 die *ListView* vom Adapter getrennt und unsichtbar gemacht wird.

```
1  [...]
2  public void saveToDb(View view) {
3    {...}
4    BasicNameValuePair tablename =
5      new BasicNameValuePair("table", Constants.TABLE_TEST);
6    BasicNameValuePair testText = new BasicNameValuePair("msg",
7          ((EditText) findViewById(R.id.editTestText)).getText().toString());
8    BasicNameValuePair testNumber = new BasicNameValuePair("nmbr", ((EditText)
9    findViewById(R.id.editTestNmbr)).getText().toString());
10
11   InsertIntoDatabase saveTask =
12     new InsertIntoDatabase(MainActivity.this, this.creds);
13   saveTask.execute(tablename, testText, testNumber);
14 }
15 public void loadFromDb(View view) {
16   {...}
17   Cursor result = db.getTestRow(1);
18
19   if (result.getCount() > 0) {
20       String[] from = new String[]
21       { Constants.KEY_UID, Constants.KEY_TEST_STRING,
22         Constants.KEY_TEST_INT, Constants.KEY_CREATED_AT };
23
24       int[] to = new int[]
25       { R.id.uid, R.id.msg, R.id.nmbr, R.id.created_at };
26
27       SimpleCursorAdapter sca = new SimpleCursorAdapter
28         (this, R.layout.list_item, result, from, to, 0);
29       ListView lv = (ListView) findViewById(R.id.listView1);
30       lv.setAdapter(sca);
31       lv.setVisibility(ListView.VISIBLE);
32   {...}
33 }
34 public void clearResultListVew(View view) {
35   TextView txtView = (TextView) findViewById(R.id.textView2);
36   txtView.setText(R.string.txtNothingLoaded);
37   txtView.setVisibility(TextView.VISIBLE);
38   ListView lv1 = (ListView) findViewById(R.id.listView1);
39   lv1.setAdapter(null);
40   lv1.setVisibility(ListView.INVISIBLE);
41 }
42 [...]
```

Abb. 4.10. Auszug aus der MainActivity.java

DatabaseConnectionLib

Das Projekt `DatabaseConnectionLib` wurde, wie bereits erwähnt, zum Auslagern von wiederverwendbaren Modulen erstellt. Aus diesem Grund befinden sich darin keine Layout, Values oder ähnliche Ressourcen die zur Darstellung und Bedienung nötig sind. Da die Ordner- und Dateistruktur der Projekte nahezu identisch ist, wird von einer weiteren Aufführung dieser Details abgesehen. In diesem Kapitel werden dementsprechend nur die Klassen näher erläutert, die im Kapitel DBPrototyp 4.1.3 schon erwähnt wurden.

Klasse SyncRemoteDatabase.java

Instanziiert wird diese Klasse in der `MainActivity.saveToDb()` Funktion. Um die Bedienbarkeit der Oberfläche nicht zu verhindern oder die App zum Einfrieren zu bringen, was durch eine langanhaltende Operation passieren könnte, wur-

de diese Klasse von der `AsyncTask`-Klasse abgeleitet. Diese ermöglicht eine komfortable Möglichtkeit mit Threads zu arbeiten und somit die Operationen vom GUI-Thread auszulagern. Der Vorteil gegenüber dem Ausführen eines einfachen `Runnable`-Objekts in einem gesonderten Thread ist, dass die `AsyncTask`-Klasse ableitbare Methoden besitzt mit denen bevor und während der Thread läuft, sowie nach dem Beenden des Threads, Operationen ausgeführt werden können. Im Fall der `SyncRemoteDatabase`-Klasse wird vor dem Ausführen des Task, mittels der Methode *onPreExecute()*, ein `ProgressDialog` erstellt (vgl. Zeilen 3 ff.). Anschließend wird im asynchronen Teil *doInBackground(String... params)*, die Funktion `syncRemoteTable(creds, params[0])`15 aufgerufen und das damit erzeugte `JSONObject`-Objekt an die Methode *onPostExecute(JSONObject json)*19 übergeben. Da diese Methode, wie die *onPreExecute*, im GUI-Thread läuft, kann der `ProgressDialog` mit einer neuen Nachricht versehen werden 21 und die erhaltenen Daten aus dem `JSONObject`-Objekt extrahieren (Zeile 26). Im selben Schritt werden die Testdaten der Testtabelle als neuen Datensatz angefügt (siehe Zeile 28).

```
1  [...]
2  @Override
3  protected void onPreExecute() {
4    super.onPreExecute();
5    pDialog = new ProgressDialog(appContext);
6    pDialog.setTitle("Contacting Servers");
7    pDialog.setMessage("Query database ...");
8    pDialog.setIndeterminate(false);
9    pDialog.setCancelable(true);
10   pDialog.show();
11 };
12 @Override
13 protected JSONObject doInBackground(String... params) {
14 /** {... **/
15   JSONObject json = userFunction.syncRemoteTable(creds, params[0]);
16   return json;
17 }
18 @Override
19 protected void onPostExecute(JSONObject json) {
20 /** {... **/
21   pDialog.setMessage("Loading Test Data");
22   pDialog.setTitle("Getting Data");
23   JSONArray json_array = json.getJSONArray("result");
24   /** {...} **/
25     db.dropSyncTable();
26     for (int i = 0; i < Integer.parseInt(res); i++) {
27       JSONObject json_data = json_array.getJSONObject(i);
28       db.addTestRow(json_data.getInt(Constants.KEY_UID),
29         json_data.getString(Constants.KEY_TEST_STRING),
30         json_data.getInt(Constants.KEY_TEST_INT),
31         json_data.getString(Constants.KEY_CREATED_AT));
32     }
33 [...]
```

Abb. 4.11. Auszug aus SyncRemoteDatabase.java

Die angesprochene Funktion *syncRemoteTable()* befindet sich in der Klasse `UserFunctions.java` 4.12. Darin wurden die Methoden zusammgefasst, die vom

Benutzer oder mit dem Benutzer zusammenhängen. Neben den Methoden *loginUser(...)* und *loggedInUser(...)*, enthält sie auch die Methode *syncRemoteTable(AuthCredentials creds, String table)* 2, die zum Abrufen der Testdaten aus der MySQL-Datenbank verwendet wird.
Dazu werden zunächst die *POST*-Parameter in eine Liste aus `BasicNameValuePair`-Objekten gepackt, wie in den Zeilen 3 bis 7 zu sehen.

```
1  [...]
2  public JSONObject syncRemoteTable(AuthCredentials creds, String table) {
3    List<BasicNameValuePair> params = new ArrayList<BasicNameValuePair>();
4    params.add(new BasicNameValuePair("tag", syncTag));
5    params.add(new BasicNameValuePair("email", creds.getEmail()));
6    params.add(new BasicNameValuePair("password", creds.getPassword()));
7    params.add(new BasicNameValuePair("table", table));
8    JSONObject json = jsonParser.getJSONFromUrl(syncUrl, params);
9    return json;
10 }
11 [...]
```

Abb. 4.12. Auszug aus UserFunctions.java

Die so verpackten Parameter werden schließlich an den `JSONParser` 4.13 übergeben. Die darin enthaltene Funktion
getJSONFromUrl(String url, List<BasicNameValuePair> params) der Klasse SyncRemoteDatabase.java schickt die erhaltenen Attribute mittels `DefaultHttpClient()`- und `HttpPost`-Objekt zur angegebenen *url* (Zeile 4 ff). Die Serverantwort, die vom Socket über den `InputStream` erreichbar wird, wird durch einen `BufferedReader` lesbar. Der `BufferedReader` ist hier besonders geeignet, da dadurch ein zeilenweises Lesen ermöglicht wird, was bei *HTTP*-Kommunikationen das Standardverfahren ist (Zeile 12 und 15). Jede Zeile wird an einen *StringBuilder* gehängt und nach dem Lesen der letzten Zeile von einem `String` (Zeile 19) in das gewünschte *JSONObject* umgewandelt.

Die so verpackte Tabelle, wird dann mit einer `for`-Schleife, siehe Zeile 26 im Auszug aus SyncRemoteDatabase.java 4.11, in die einzelnen Zeilen zerlegt und von der Funktion *addTestRow(...)* 3 in die lokale `SQLite`-Datenbank geschrieben (Zeile 28). Dazu werden die einzelnen Werte in zunächst mit den zugehörigen Spaltennamen in ein sogenanntes `ContentValues`-Objekt geschrieben, was im Groben einer Tabellenzeile entspricht. Eine solche Kapselung ist theoretisch nicht nötig, da die Möglichkeit besteht, die Werte durch ein zusammengesetzten SQL-Statement in die Datenbank zu schrieben. Diese Lösung ermöglicht aber die Verwendung der *insert()*-Methode wie in Zeile 12, wodurch der Code nicht nur weniger fehleranfällig, sondern auch übersichtlicher und sicherer wird.

Klasse InsertIntoDatabase

Ebenfalls aus der `MainActivity` heraus wird diese Klasse instantziiert und ausgeführt. Sie dient zum Schreiben von Testdaten in die `MySQL`-Datenbank und ist

```
1  public JSONObject getJSONFromUrl(String url,
2      List<BasicNameValuePair> params) {
3  [...]
4    DefaultHttpClient httpClient = new DefaultHttpClient();
5    HttpPost httpPost = new HttpPost(url);
6    httpPost.setEntity(new UrlEncodedFormEntity(params));
7    HttpResponse httpResponse = httpClient.execute(httpPost);
8    HttpEntity httpEntity = httpResponse.getEntity();
9    is = httpEntity.getContent();
10 [...]
11   BufferedReader reader =
12     new BufferedReader(new InputStreamReader(is, "iso-8859-1"), 8);
13   StringBuilder sb = new StringBuilder();
14   String line = null;
15   while ((line = reader.readLine()) != null) {
16     sb.append(line + "\n");
17   }
18   is.close();
19   json = sb.toString();
20 [...]
21 else {
22   jObj = new JSONObject(json);
23 }
24 [...]
```

Abb. 4.13. Auszug aus JSONParser.java

```
1  [...]
2  public void addTestRow(int id, String text, int number,
3      String created_at) {
4    SQLiteDatabase db = this.getWritableDatabase();
5    ContentValues vals = new ContentValues();
6    if (id > 0 & created_at != null & !created_at.equals("")) {
7      vals.put(Constants.KEY_UID, id);
8      vals.put(Constants.KEY_TEST_STRING, text);
9      vals.put(Constants.KEY_TEST_INT, number);
10     vals.put(Constants.KEY_CREATED_AT, created_at);
11   }
12   db.insert(Constants.TABLE_TEST, null, vals);
13   db.close();
14 }
15 [...]
```

Abb. 4.14. Auszug aus der DatabaseHandler.java

somit auch eine Subklasse von `AsyncTask`.
Analog zur Klasse SyncRemoteDatabase.java wird auch hier zunächst ein Objekt der Klasse erstellt, dem dann mit der *execute(...)*-Methode die Testwerte als Parameter übergeben werden. Nachdem der `ProgressDialog` erstellt wurde, werden die Daten in Schlüssel- und Wertevariablen gekapselt und der *insertIntoRemoteTable(...)*-Methode übergeben.
Um einen standardkonformen *HTTP*-Request abzuschicken, werden dort wieder die *POST*-Parameter in einer Liste gekapselt und dem `JSONParser` 4.13, zum Abschicken der Anfrage übergeben. Die Antwort wird wieder analog zur Klasse SyncRemoteDatabase.java in der *onPostExecute()*-Methode ausgewertet und eine Benachrichtigung angezeigt, ob die Übertragung erfolgreich war oder nicht.

5

Implementierung von *DorMApp*

Die App wurde zum Großteil in der aktuell von Google empfohlenen Umgebung Android Studio umgesetzt. Zu Beginn fand die Entwicklung noch in Eclipse mit dem entsprechenden Plug-In statt. Jedoch erschwerten die Bugs und die Behäbigkeit der Eclipse-IDE den zügigen Fortschritt. Aus diesem Grund wurde nach dem Legen des Grundsteins das Projekt auf die neue Entwicklungsumgebung migriert, wo es auch fertiggestellt wurde.

5.1 Entwicklungsgrundlagen

Durch die Prototyp-Enwicklung konnte zu Beginn der Implementierung schon auf einige Bausteine zurückgegriffen werden. Zunächst wurde die Kommunikation mit der Datenbank nicht verändert. Somit ruft die App weiterhin ein PHP-Skript auf, welches dann die Tabelleninhalte ausliest, aufbereitet und im *JSON*-Format zurücksendet. Die von der ***DorMApp***-App benötigten Skripte, sowie der Zugriff auf die Datenbank werden im Kapitel App/Datenbank Schnittstelle 5.6 genauer erläutert. Sollte es das Verständnis eines Sachverhaltes fordern, so wird auf das entsprechende Kapitel vorgegriffen und mit in die Erklärung aufgenommen.

Externe Bibliotheken

Um Entwicklungszeit zu sparen, wurden vier externe Bibliotheken eingesetzt. Um eine bessere Sicherheit zu erreichen und die Logindaten der Benutzer unverschlüsselt im `SharedPreferences`-Bereich der App abzulegen, wurde die Bibliothek *secure-preferences* [**?**] eingebunden. Diese Bibliothek stellt eine Schnittstelle für die eigentlichen `SharedPreferences` bereit, die zwar die gleichen Methoden bereitstellt, aber alle Daten verschlüsselt im App-Speicherbereich ablegt. Das bringt, neben der gleichen Verwendung, den Vorteil, dass die vom Android-Betriebssystem bereitgestellten `SharedPreferences` weiterhin für unsensible Daten genutzt werden kann.
Weiterhin wurde auf die Bibliothek *UndoBar* [**?**] zurückgegriffen. Dadurch wird ein einfache und schnelle Möglichkeit zum Erstellen der `Toasts` mit `Undo`-Button integriert. Neben der visuellen Komponente stellt *UndoBar* auch noch die Logik

hinter dieser intuitiven Art der Benutzerführung dem Entwickler zur Verfügung. Bei Bedarf kann nach dem Verschwinden des `Toasts` direkt die Änderung speichern oder, wenn der Benutzer die Änderung rückgängig machen will, bei Buttonklick den vorigen Zustand wiederherstellen.
Zu Testzwecken wurden die Bibliotheken *android-test-kit* [**?**], die auch als *Espresso*-Testkit bekannt ist, und *mockwebserver* [**?**] in das Projekt aufgenommen. In dieser Kombination lassen sich ohne großen Aufwand die Abläufe auf der GUI testen, sowie durch gestellte Serverantworten, die Verarbeitung der Daten in verschiedenen Testfällen testen.
Abgesehen von diesen externen Lösungen, flossen noch Teile der Android-Samples [**?**] mit in die Entwicklung ein und eine anfängliche Hilfestellung bei der Entwicklung bot die Seite *learn2crack.com* [**?**] konsultiert, die eine Vielzahl an Beispielprojekten und Lösungsansätzen aufweist.

5.2 App

Dieses Kapitel umfasst die Implementierung der App, auf der die weiteren Teile aufbauen. Dabei wird näher auf die service-ähnliche Struktur und deren Umsetzung, sowie die sichere Speicherung von Benutzerinformationen eingegangen. Anfangs wird noch der Ablauf beim ersten Start, den folgenden Starts sowohl mit als auch ohne angemeldetem Benutzer beschrieben.
Da eine Verwendung der App ohne Anmeldung, sprich ohne personalisierte Daten, nicht vorgesehen ist, wird zunächst der App-Speicher auf vorhandene Login-Informationen überprüft. Aufgrund der Annahme, dass es sich um die erste Verwendung nach der Installation handelt, können noch keine Benutzerdaten vorhanden sein und es wird direkt in die *LoginActivity* weitergeleitet, die ohne Anmeldung nicht verlassen werden kann. Hat sich der Benutzer erfolgreich authentifiziert, startet die Synchronisierung aller Tabellen um die aktuellsten Daten zu erhalten. Ein Beenden ohne Logout hat zur folge, dass E-Mailadresse und Passwort über die `Secure-Preferences`-Schnittstelle verschlüsselt im Speicherbereich abgelegt werden. Die erfolgt in der *onPause()*-Methode, um den Verlust der Daten beim Zerstören der App aufgrund von Ressourcenknappheit, zu verhindern. Beim darauffolgenden Start wird in der *onResume()*-Methode geprüft, ob verschlüsselte Credentials im Speicher hinterlegt sind. Nach dem Auslesen werden damit direkt die aktuellen Bewegungsdaten aus der Datenbank abgerufen und lokal gespeichert. Hat sich der Benutzer vor dem Beenden der App abgemeldet, wurden beim Schließen keine Informationen im App-Speicher abgelegt. Dadurch landet der Benutzer, wie beim Erststart, wieder in der Loginmaske.

5.2.1 Probleme

Neben denen im Prototyp 4 gelösten Problemen, die schon vor dem Beginn der Implementierung erkannt wurden, sind auch während der Umsetzung einige Stolperfallen aufgetreten. Dazu gehört zunächst das Problem, die Logininformationen so sicher wie möglich zu speichern. Desweiteren sollte die Server- und Datenbankkommunikation möglichst unabhängig vom Darstellungsteil der App gehalten werden.

Unsichere App-Speicher

Da der `SharedPreference`-Speicher nicht verschlüsselt ist und durch einfache Mittel ausgelesen werden kann, können dort keine sensiblen Daten ohne weiteres gespeichert werden. Die sicherste Art wäre es, die Logininformationen nicht zu speichern, was aber dazu führen würde, dass sich der Benutzer bei jedem Start der App neu anmelden müsste. Das ist dem Benutzer aber unter keinerlei Umständen zumutbar.

Datenbank-Service

Das Trennen der Oberfläche von der Datenhaltung hat mehrere Vorteile und wurde deswegen auch in diesem Projekt umgesetzt. Zunächst ist es dadurch möglich die

Kommunikation zwischen App und Datenbank zu ändern, ohne dass die Oberfläche angepasst werden muss. Weiterhin ist es dadurch einfacher möglich die Umsetzung auf verschiedene Entwickler aufzuteilen, da keine ständige Rücksprache nötig ist, sondern nur zu Beginn die Schnittstelle definiert werden muss.

5.2.2 Lösungen

Zur Lösung der vorangegangenen Probleme, sind nachfolgend kurze Auszüge aus dem Quelltext mit einer knappen Erläuterung der Funktion

Secure-Preferences

Wie zuvor erwähnt, ist der App-Speicher nicht verschlüsselt und somit eigentlich für die Ablage von Passwörtern ungeeignet. Die Lösung dieses Problems bringt der Einsatz einer Verschlüsselung beim Schreiben der `SharedPreferences`. Dabei kommt die schon erwähnte Bibliothek *Secure-Preferences* ins Spiel, wodurch die Informationen vor dem Schreiben in den App-Speicher verschlüsselt werden. Dabei wird einfach die schon vorhandene `SharedPreferences`-Funktion von Android mit einer extra Schnittstelle dazwischen verwendet, die beim Schreiben ver- und beim Lesen entschlüsselt. Die Verwendung der *Secure-Preferences* ist einfach und analog zur Verwendung der Standard-`SharedPreferences`. Das physikalische erstellen der Datei auf dem Datenträger geschieht durch das Instanziieren der Klasse `SecurePreferences` im `Context` der App. Wie auch bei den `SharedPreferences` kann auch hier optional ein eigener Name für die Datei übergeben werden, ist hier jedoch nicht notwendig. Hat sich ein Benutzer angemeldet und wird die App geschlossen, wird die Funktion *storeCredentials(...)* 8 aufgerufen und die Referenz zum *SecurePreferences*-Objekt übergeben, sowie die `AuthCredentials`. Damit überhaupt Daten geschrieben werden können, muss zunächst ein `Editor`-Objekt erstellt werden, was durch `.edit()` in Zeile 9 geschieht. Damit einen eventuell verwaister Eintrag keine Probleme bereitet, wird der Speicher in der folgenden Zeile sicherheitshalber komplett gelöscht. Danach werden die Attribute aus dem `AuthCredentials`-Objekt ausgelesen und zusammen mit einem eindeutigen Bezeichner durch den `.put(\ldots{})`-Befehl dem `Editor` zum Speichern übergeben (vgl. Zeile 12 ff). Um die Daten nun physikalisch zu schreiben, wird auf dem `Editor` der `.commit()` (vgl. Zeile 17) ausgeführt. Ausgelesen werden die Daten einfach in der umgekehrten Reihenfolge, mit dem einzigen Unterschied, dass hierzu kein `Editor` benötigt wird. Wird zum Beispiel die App gestartet, ruft die *onResume()*-Methode die *loggedInUser(...)* in Zeile 19 auf. Darin wird nach dem Sicherstellen ob Zugangsdaten vorhanden sind, die einzelnen Schlüssel-Wert-Paare wieder ausgelesen (siehe Zeile 22 ff). Ist keiner der Werte `null`, werden sie in einem `AuthCredentials` verpackt zurückgegeben.

Hat man schon mit den `SharedPreferences` gearbeitet, kann man klar die analoge Vorgehenseweise erkennen. Obwohl die Entwickler keine hunderprozentige Sicherheit garantieren können und möchten, ist es dennoch dem unverschlüsselten Ablegen vorzuziehen.

```
1  [...]
2  public void resetCredentials(final SecurePreferences secPrefs) {
3    Editor secPrefEditor = secPrefs.edit();
4    secPrefEditor.clear();
5    secPrefEditor.commit();
6  }
7  public static void storeCredentials(final SecurePreferences secPrefs,
8    AuthCredentials _creds) {
9    Editor secPrefEditor = secPrefs.edit();
10   secPrefEditor.clear();
11   secPrefEditor.putString(EnumSqLite.KEY_UID.getName(),
12       _creds.getUid());
13   secPrefEditor.putString(EnumSqLite.KEY_PASSWORD.getName(),
14       _creds.getPassword());
15   secPrefEditor.putString(EnumSqLite.KEY_EMAIL.getName(),
16       _creds.getEmail());
17   secPrefEditor.commit();
18 }
19 public static AuthCredentials loggedInUser(final SecurePreferences secPrefs)
     {
20   String uid = null, uname = null, upassword = null, email = null;
21   if (!secPrefs.getAll().isEmpty()) {
22     uid = secPrefs.getString(EnumSqLite.KEY_UID.getName(), null);
23     upassword = secPrefs.getString(EnumSqLite.KEY_PASSWORD.getName(), null);
24     email = secPrefs.getString(EnumSqLite.KEY_EMAIL.getName(), null);
25   }
26   if (uid != null & upassword != null & email != null ) {
27     AuthCredentials creds = new AuthCredentials(uid, email, upassword);
28     return creds;
29   }
30   return null;
31 }
32 [...]
```

Abb. 5.1. Verwendung von Secure-Preferences

Messenger-Klasse

Wie schon erwähnt, sollte eine Trennung von Oberflächen- und Datenlogik erstrebt werden. Um diese Trennung zu erreichen, wurde die `Messenger`-Klasse verwendet [?]. Mit dieser Klasse ist die Implementierung eines `gebundenen Service` einfacher als eine mit einer `AIDL`-Schnittstelle, erfüllt aber alle Vorraussetzungen die für dieses Programm witchtig sind.
Der Aufbau der *Messenger*-Schnittstelle ist übersichtlich und mit wenigen Schritten erreicht. Zunächst wird eine `MessengerService`-Klasse erstellt, die von der `Service`-Klasse erbt. Somit muss die Methode *onBind(...)* (vgl. Zeile 5) implementiert werden, welche als `Binder` eine Instanz der inneren Klasse `IncomingHandler` zurückgibt (vgl. Zeile 2. Der `IncomingHandler` arbeitet die eingehenden Anfragen seriell ab und führt mittels einer `switch-case` die erwünschten Operationen aus wie in Zeile 13 ff zu sehen ist.

Neben den erwähnten Methoden enthält die Klasse `MessengerService` außerdem noch einige Hilfsmethoden, die zum Beispiel zum Entpacken der `Bundles` verwendet werden.

```
1  [...]
2    private final Messenger mMessenger = new Messenger(new IncomingHandler());
3  [...]
4  @Override
5  public IBinder onBind(Intent intent) {
6    return mMessenger.getBinder();
7  }
8  public class IncomingHandler extends Handler {
9
10   public static final String TAG = Constants.TAG_PREFIX + "IncomingHandler";
11
12   @Override
13   public void handleMessage(Message msg) {
14     // TODO
15     String[] tablesToSync;
16     int ppAufgId;
17     Bundle _bundle;
18     Map<String, String> params;
19     int remItemId, shoppingListId;
20     switch (msg.what) {
21         case MessageConstants.MSG_UNREBIND:
22             reService = null;
23             reBound = false;
24             break;
25       {...}
26      }
27      {...}
28    }
29    {...}
30 }
31 [...]
```

Abb. 5.2. Auszug aus MessengerService.java

5.3 Putzplan

Der Putzplan soll den Benutzern zeigen, wer als nächstes für eine Aufgabe an der Reihe ist. Dabei werden bei der Synchronisierung die für die gesamte WG anfallenden Aufgaben kopiert. Somit ist es auch möglich die Arbeiten eines anderen WG-Mitbewohners abzuarbeiten. Für die einzelnen Aufgaben, wie Küche oder Badezimmer, können noch Schritte definiert werden. Diese sind zu erledigen, bevor die Aufgabe als erledigt angesehen wird. Sobald alle Schritte markiert sind, wird beim Übertragen automatisch die Aufgabe auf erledigt gesetzt.

5.3.1 Umsetzung

Die Anzeige der Aufgaben wurde durch ein, von `Fragment` abgeleitetes, `ChorePlanFragment` realisiert. Wechselt man zu dem Besagten `Fragment`, wird zunächst das passende Layout geladen (vgl. Zeile 4). Danach wird aus der *onView-Created(...)* der Inhalt des Fragments initialisiert. Dabei wird sowohl für die Aufgaben als auch die Schritte eine `SQL`-Statement auf der `SQLite`-Datenbank ausgeführt und in einen Cursor geladen. Die aus der Datenbank geladenen Daten werden anschließend in eine `ArrayList` hinzugefügt, beziehungsweise in eine `HashMap` gesetzt. Diese können dann dem `ChorePlanAdapter` übergeben werden, der die Daten für die Anzeige aufbereitet und dem `ListView`-Element im `ChorePlanFragment` anhängt. Schlussendlich wird dem *Erledigt*-Button die Funktion hinterlegt, die Schritte mit geänderter Markierung aus dem `ChorePlanAdapter` auszulesen und in der lokalen Datenbank zu speichern.

5.3.2 Probleme und Lösungen

Bei diesem Teil der App kam es an zwei Stellen zu leichten Problemen. Dazu zählt zum einen das Aufklappen der `ListView`-Zeilen, zum anderen war das nicht-persistente Ändern eine kleine Herausforderung.

Probleme

Um die Liste der anfallenden Aufgaben übersichtlicht zu halten, aber trotzdem die Schritte bei den zugehörigen Aufgaben anzuzeigen, bewerkstelligen zu können, wurden die einzelnen `ListView`-Elemente klappbar gemacht. Somit klappt beim klick auf eine Zeile der Teil mit den Aufgabeschritten auf. Womit die Zusammengehörigkeit symbolisiert wird und die Bedienbarkeit intuitiv ist.
Wie im Sourcecode-Auszug 5.1 in der letzten Zeile zu erkennen ist, werden dort noch keine Daten geschrieben, sondern nur ein `Bundle` mit den geänderten Daten erzeugt. An diese Stelle soll die Funktion des *Undo*-Buttons kommen. Das bedeutet, eine Art `Toast`, der erzeugt wird sobald der Benutzer Daten geändert hat, und über einen Button verfügt, mit dem die Änderungen wieder rückgängig gemacht werden können. Bis dato stellt die ANDROID-API keine derartige Funktion direkt bereit, noch wird auf der Developer-Homepage [**?**] ein Lösung für das Problem angeboten.

```
1  [...]
2  @Override
3  public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
       savedInstanceState) {
4    rootView = inflater.inflate(R.layout.fragment_chores, container, false);
5    return rootView;
6  }
7  [...]
8  cpAdapter = new ChorePlanAdapter(getActivity(), chores, steps);
9  lvChorePlan.setAdapter(cpAdapter);
10 query = generateStepsQueryString();
11
12 try {
13     result = dbHandler.getCursorForQuery(query, null);
14 }
15 catch (SQLiteException sqe) {
16     sqe.printStackTrace();
17     Log.e(TAG, sqe.getLocalizedMessage());
18 }
19 if (result != null && result.moveToFirst()) {
20 [...]
21 final Button btnChoreDone = (Button) rootView.findViewById(R.id.btnChoreDone)
       ;
22 btnChoreDone.setOnClickListener(new View.OnClickListener() {
23   private ArrayList<ChoreStepItem> selectedSteps = new ArrayList<
         ChoreStepItem>();
24
25   @Override
26   public void onClick(View v) {
27     Map<Integer, ChoreStepItem> steps = cpAdapter.getSelectedSteps();
28     ChorePlanStep commitStep = new ChorePlanStep(getActivity());
29
30     ArrayList<Integer> ids = new ArrayList<Integer>();
31     ArrayList<Integer> choreStepIds = new ArrayList<Integer>();
32     for (ChoreStepItem step : steps.values()) {
33         Long longId = commitStep.doChoreStep(step.getChorePlanId(), step.
               getChoreStepId(), step.getChoreId()) ? step.getChoreStepId() : 0L
               ;
34         ids.add(Integer.parseInt(String.valueOf(longId)));
35         choreStepIds.add(step.getChoreStepId());
36         selectedSteps.add(step);
37     }
38     Bundle token = new Bundle();
39     token.putIntegerArrayList("insertedRowIds", ids);
40     token.putIntegerArrayList("choreStepId", choreStepIds);
41 [...]
```

Listing 5.1. ChorePlanFragment.java

Lösungen

Um ein Aufklappen des Eintrags zu simulieren wurde zunächst eine Animations-Klasse 5.2 geschrieben. Die sorgt dafür, dass die übergebene `View` über eine angegebene Zeitspann hinweg aufgeklappt wird (Zeile 2).

Diese Animation wird bei einem Klick auf einen beliebigen Punkt in der Zeile der `ListView` ausgelöst. Der dafür benötigten `onClickListener(...)` wird im *ChorePlanAdapter* an die einzelnen `Views` gehängt (Zeile 1). Da es immer nur eine aufgeklapte Zeile geben soll, muss noch überprüft werden ob, es eine offene gibt, die geklickte die offene oder ob noch keine geöffnet Zeile in der Liste war (vgl. Zeile 7). Außerdem wird im Falle eines Zeilenwechsels die Auswahl der Schritte zurück-

```
1  \centering
2  public ExpandAnimation(View view, int duration) {
3    setDuration(duration);
4    mAnimatedView = view;
5    mViewLayoutParams = (LayoutParams) view.getLayoutParams();
6
7    mIsVisibleAfter = (view.getVisibility() == View.VISIBLE);
8
9    mMarginStart = mViewLayoutParams.bottomMargin;
10   mMarginEnd = (mMarginStart == 0 ? (0- view.getHeight()) : 0);
11
12   view.setVisibility(View.VISIBLE);
13 }
```

Listing 5.2. ExpandAnimation.java

```
1  convertView.setOnClickListener(new View.OnClickListener() {
2
3  @Override
4  public void onClick(View v) {
5    View toolbar = v.findViewById(R.id.lstViewChoreSteps);
6
7    if (prevToolbar != null
8          && prevToolbar.getVisibility() == ListView.VISIBLE
9          && toolbar.getVisibility() != ListView.VISIBLE) {
10     ExpandAnimation tmpAnimation = new ExpandAnimation(prevToolbar, 0);
11     prevToolbar.startAnimation(tmpAnimation);
12
13     TableLayout lvChoreSteps =
14       (TableLayout) prevToolbar.findViewById(R.id.lstViewChoreSteps);
15     for(int i= 0; i < lvChoreSteps.getChildCount(); i++) {
16       TableRow row = (TableRow) lvChoreSteps.getChildAt(i);
17       CheckedTextView chdTxtView =
18         (CheckedTextView) row.findViewById(R.id.chkTxtViewStep);
19
20       if (!selectedSteps.isEmpty()
21             && selectedSteps.containsValue(
22               steps.get(Integer.parseInt(row.getTag().toString()))
23             )) {
24         chdTxtView.setChecked(false);
25       }
26     }
27
28     resetSelection();
29   }
30
31   ExpandAnimation expandAni = new ExpandAnimation(toolbar, 0);
32   toolbar.startAnimation(expandAni);
33   prevToolbar = toolbar;
34 }
35 });
```

Listing 5.3. ChorePlanAdapter.java

gesetzt, um beim späteren Speichern der Änderungen nicht die falschen Schritte auf 'erledigt' zu setzen, wozu in Zeile 28 ein *resetSelection()* aufgerufen wird. In Zeile 32 wird dann die Animation auf dem entsprechenden Element ausfgeführt.

Das Problem mit dem rückgängig machen der letzten Änderung über ein `Toast` zieht sich durch das gesamte Projekt und wird hier nun beispielweise erklärt. Nachdem in 5.1 in Zeile 40 die Daten soweit aufbereitet wurden, dass ein umsetzen der

Änderungen möglich ist, folgt nun der Einsatz der *UndoBar* 5.4. Zunächst wird ein neues Objekt des `UndoBarController.UndoBar` erstellt. Dieses wird keiner Variablen zugewiesen, da kein weiterer Zugriff darauf erfolgt. Nacheinander werden der Schaltfläche nun die Eigenschaften zugewiesen. Zunächst das in 5.1 in Zeile 40 erstellte `Bundle`, in dem die zu ändernden Objekte stecken. In Zeile 4 wird die anzuzeigende Nachricht gesetzt, in diesem Fall ein aus den Ressourcen geladener String. Nun folgen die besonderen Eigenschaften, der `Listener` zum Persistieren (Zeile 7), beziehungsweise in Zeile 18 die Undo-Funktion. Möchte der Benutzer die Änderung nicht rückgängig machen, werden in diesem Fall mit der *onHide(. . .)*-Methode die ausgewählten Schritte über das im Kapitel 5.2.2 erwähnte `Messenger`-Interface an den Datenbank-Service übergeben und synchronisiert. Wünscht der Benutzer jedoch, die getätigten Änderunge zu annulieren, kann er das durch einen klicken des *Undo*-Buttons anstoßen. Dadurch wird die erwähnte *onUndo(. . .)*-Methode in Zeile 18 ausgeführt. Darin werden zuerst die lokal geänderten Tabelleneinträge aus dem Token gelesen (Zeile 20). Darauf folgt die Löschung der Einträge aus der `SQLite`-Datenbank in Zeile 26 sowie die Entfernung der Markierung auf der grafischen Oberfläche (vgl. Zeile 31). Abschließend wird noch durch einen simulierten Klick auf die `ListView`-Zeile das Schließen der aufgeklappten Schritte (Zeile 36) erwirkt.

```
1  [...]
2  new UndoBarController.UndoBar(getActivity())
3       .token(token)
4       .message(getString(R.string.textChoreStepSaved))
5       .listener(new UndoBarController.AdvancedUndoListener() {
6         @Override
7         public void onHide(Parcelable _token) {
8           if (_token != null) {
9             if (selectedSteps != null && selectedSteps.size() >= 1) {
10              for (ChoreStepItem cStepItm : selectedSteps) {
11                Message msg = Message.obtain(null,
12                  MessageConstants.MSG_COMMIT_CHORE_STEP_DONE,
13                  cStepItm.getChorePlanId(), cStepItm.getChoreStepId());
14                ((MainActivity) getActivity()).sendMessage(msg);
15          } } } }
16 [...]
17        @Override
18        public void onUndo(Parcelable _token) {
19            if (_token != null) {
20                ArrayList<Integer> arrayList =
21                  ((Bundle) _token).getIntegerArrayList("insertedRowIds");
22                ArrayList<Integer> cPlChIds =
23                  ((Bundle) _token).getIntegerArrayList("choreStepId");
24                for (int rowId : arrayList) {
25                    ChorePlanStep commitStep = new ChorePlanStep(getActivity())
                          ;
26                    commitStep.undoChoreStep(rowId);
27
28                    for (Integer id : cPlChIds) {
29                    CheckBox chkBoxChoreDone =
30                      ((CheckBox) rootView.findViewWithTag(id));
31                    chkBoxChoreDone.setChecked(false);
32                    chkBoxChoreDone.setEnabled(false);
33                    }
34                }
35                rootView.findViewWithTag(selectedSteps.get(0).
36                  getChorePlanId()).performClick();
37      } } }).show();
38 [...]
```

Listing 5.4. ChorePlanFragment.java

5.4 Einkaufsliste

Wie der Name schon sagt, soll diese Liste die für die WG benötigten Artikel zusammengetragen werden. Dazu kann jedes Mitglied Einträge hinzufügen oder kaufen. Abschließend soll dadurch auch die Abrechnung vereinfacht werden, da den gekauften Artikel jeweils der Käufer, als auch der bezahlte Preis zugeordnet werden kann.

5.4.1 Implementierung

Wie die Aufgaben wurde auch für die Einkaufliste eine Klasse `ShoppingListFragment` von der Klasse `Fragment` abgeleitet. Da hierbei bis einschließlich zum Einsatz des `ShoppingListAdapter` analog zum `ChorePlanFragment` vorgegangen wurde, ist eine erneute Ausführung nicht notwendig. Im Vergleich dazu wurde jedoch ein `AutoCompleteTextView` verwendet. Da meist die gleichen Artikel hinzugefügt werden müssen, ist der Einsatz der Autovervollständigung hier eine große Erleichterung. Dazu werden zunächst die gewünschten Vorschläge aus der Datenbank in ein `String-Array` geladen (vgl. Zeile 2). Danach wird ein `ArrayAdapter` erzeugt, dem das `Array` mit den Vorschlägen, sowie ein Layout übergeben werden. Dieser Adapter wird dem vorbereiteten `AutoCompleteTextView` auf dem Einkaufslisten-`Fragment` gesetzt. Als weitere Erleichterung wurde `Listener` zur automatischen Eingabebestätigung implementiert (Zeile 9), welcher beim Hinzufügen von Waren zum Einsatz kommt, die noch nicht als Vorschlag verfügbar hinterlegt sind. Dazu kommt der `TextView.OnEditorActionListener()` zum Einsatz und fängt das Drücken der Tasten auf der Tastatur ab und überprüft ob es sich dabei um die `Senden`-Taste handelt (siehe Zeile 13). In diesem Fall wird die Weitergabe des Events unterbrochen und die *handleItemAddAction(...)* aufgerufen (vgl. Zeile 14, der der eingegebene Text übergeben wird und diesen als neue Auswahl zur Verfügung stellt, sowie ein Dialog öffnet mit dem die benötigte Anzahl für die Einkaufsliste übergeben werden kann.

Dieser Teil der App konnte ohne weitere Probleme gelöst werden, weswegen auf die hier üblichen Paragraphen Probleme und Lösungen verzichtet.

```
1  [...]
2  final String[] grocieries = dbHandler.getGrocieries();
3  ArrayAdapter<String> adapter = new ArrayAdapter<String>(getActivity(),
4      android.R.layout.simple_list_item_1, grocieries);
5  final AutoCompleteTextView editTextNew =
6      (AutoCompleteTextView) rootView.findViewById(R.id.
           autoCompleteShoppingListNewItem);
7  editTextNew.setAdapter(adapter);
8
9  editTextNew.setOnEditorActionListener(new TextView.OnEditorActionListener() {

10   @Override
11   public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
12     boolean handled = false;
13     if (actionId == EditorInfo.IME_ACTION_SEND) {
14       handleItemAddAction(v.getText().toString());
15       handled = true;
16     }
17     return handled;
18 } });
19 [...]
```

Listing 5.5. ShoppingListFragment.java

5.5 Blackboard

Eine einfache Möglichkeit alle WG-Mitglieder zu erreichen bietet ein Blackboard auf dem jeder Nachrichten hinterlassen kann. Obwohl der Einsatz von Zugriffsbeschränkungen auf die Nachrichten leicht umsetzbar wäre, wurde bewusst darauf verzichtet, um die Eigentschaften eines physikalischen Blackboards gerecht zu werden.

5.5.1 Implementierung

Die Klassenhierarchie der Fragmente ist aus den vorhergehenden Beispielen schon bekannt und wurde auch in beim `BlackboardFragment` beibehalten. Dabei wird in der überschriebenen Methode *onCreateView(...)* durch ein SQL-Statement die Nachrichten aus der `SQLite`-Datenbank gelesen, welche in einem `Cursor` vorgehalten werden. Da es sich bei dem Zeilenlayout der `ListView` um kein Standard-Layout handelt, muss zunächst der `Cursor` schrittweise durchgearbeitet werden und die Werte in eine `HashMap` übertragen werden. Auf eine `HashMap` wurde zurückgegriffen, um beim Löschen des Eintrags einfach über die *BlackboardId* an die Nachricht zu gelangen und aus der Liste zu löschen, das spart einen direkten Datenbankzugriff für das Löschen und weitere Zugriffe beim Aktualisieren der Liste, sowie dem etwaigen Wiederherstellen der Nachricht. Über eine `EditText`-Feld kann die neue, mehrzeilige Nachricht eingegeben werden und durch den +-Button dem schwarzen Brett hinzugefügt werden. Das Löschen der einzelnen Nachrichten kann durch ein Klick auf das Löschen-Symbol ausgelöst werden und ist durch die `UndoBar`-Funktion revidierbar. Zum Bearbeiten wurde ein `onLongClickListener` an die `View` der Zeile gebunden. Wird dieser ausgelöst, so generiert er einen Dialog mit dem aktuellen Inhalt der Nachricht und zeigt diesen an. Gespeichert wird dann die Nachricht mit der ID des Bearbeiters. Eine Erweiterung um die `UndoBar`-Funktion sollte hier noch ergänzt werden.

Probleme

Bei der Umsetzung des Blackboards kam es beim Anzeigen der Nachrichten zu einem komplexen Problem, was zunächst nicht nachvollziehbar war. In der `ListView` war zwar die Anzahl der Einträge richtig, aber der Anfang der Liste wurde am Ende der Liste, also bei Zeilen die ausserhalb des anfänglich darstellbaren Bereichs lagen, wiederholt.

Lösungen

Um die Datenquelle als Fehler auszuschließen wurde zunächst ein `DISTINCT` in das `SQL`-Statement eingefügt. Das bewirkt, dass doppelte Einträge ausgefiltert werden. Das war aber nicht die Ursache des Problems, da die erwarteten Daten im `Cursor` waren und auch an den `BlackboardAdapter` weitergegeben wurden. Somit lies sich das Problem auf die Anzeige, beziehungsweise das die Vorbereitung der Daten zu Anzeige, eingrenzen. Demnach muss sich der Fehler im erwähnten `BlackboardAdapter` befinden. Nachdem weitere Gedanken über die Funktion des

```
1  [...]
2  @Override
3  public View getView(int position, View convertView, ViewGroup parent) {
4    if (convertView &=& null) {
5      final BlackboardMessage bbMsg =
6          (BlackboardMessage) blackboardMessages.values().toArray()[position];
7    [...] }
8  }
9  [...]
```

Listing 5.6. Alte BlackboardAdapter.java

Adapters gemacht wurden, kam die Erkenntnis, dass die Zeile 4 im alten Quelltext 5.6 nicht funktionieren kann, sobald es mehr Einträge gibt, als auf anhieb anzeigbar sind.

Mit der `if`-Abfrage, ob die übergebene `View` noch `null` ist, wird verhindert, dass wenn die `ListView` gescrollt wird, die neue Zeile überschrieben werden kann. In der `ListView` befinden sich nämlich immer gleich viele `Views` als Zeilen, vorrausgesetzt dass mehr Einträge dargestellt werden sollen als auf den sichtbaren Bereich passen. Tritt der Fall ein dass neue Zeilen angezeigt werden müssen, sprich es wird gescrollt, werden die angezeigten `Views` der *getView(int position, View convertView, ViewGroup parent)* (vgl. Zeile 3 im Alte BlackboardAdapter.java 5.6) als *converView* übergeben. Somit kann diese Variable nicht `null` sein und die Bedingung der `if`-Abfrage ist falsch. Demzufolge können die alten Daten nicht in den vorhandenen `Views` durch die neuen ersetzt werden und die selbe Nachricht wird noch mal angezeigt. Die Wiederholung des Anfangs wird dadurch erzeugt, dass die Zeilen-`Views` wiederverwendet werden, die aus dem angezeigten Bereich geschoben werden, also die zuvor erste `View`.
Durch das Entfernen der `if`-Abfrage wurde die erwünschte Funktion erreicht und das Scrollen der Liste war möglich.

5.6 App/Datenbank Schnittstelle

In Kapitel Prototyp 4, als auch in Kapitel App 5.2 wurde bereits die Problematik aufgegriffen, dass es für eine Android-App nicht möglich ist über einen `JDBC`-Treiber eine Verbindung mit einer Datenbank aufzubauen. Neben der Verwendung eines RESTfull Webservice zum Abrufen von Datenbankinhalten, gibt es die hier verwendete Methode über PHP-Dokumente die über eine HTTP-Verbindungen abgerufen werden und einen JSON-formatierte Antwort liefern. Diese Vorgehensweise kann mit einfachen Mitteln realisiert werden und Bedarf keiner komplexen Serverkonfiguration. Der Aufbau wird ausgehend vom aufgerufenen PHP-Skript, über die PHP-Datenbankschnittstelle bis schließlich zur Datenbank hin erklärt.

5.6.1 PHP-Skript

Der Aufruf des Skripts erfolgt über eine einfache HTTP-POST-Anfrage an die Server-URL. Der Aufbau wurde bereits im Kapitel DatabaseConnectionLib 4.1.3 Paragraph 4.1.3 erläutert.
Wurden die notwendigen Parameter angegeben, so erreicht die Abarbeitung die Zeile 2 in der die gewünschte Operation dem HTTP-Request entnommen wird. Nach der erfolgreichen Authentifizierung der übermittelten Zugangsdaten in Zeile 7, findet die Auswertung der *tag*-Parameters statt. Dabei gibt es drei mögliche Zustände. Entweder wurde eine Synchronisierung angestoßen, ein Datensatz soll in eine Tabelle geschrieben werden oder ein Änderung wird committed. Der zweite Zustand, schreiben eines Datensatz, wurde nur für den Prototyp benötigt und kann hier vernachlässigt werden. Beim Synchonisieren wird dann aus den Parametern die gewünschte Tabelle gelesen (Zeile 10). Die Parametervariable wird mit der User- und WG-Id der PHP-Datenbankschnittstelle übergeben, welche die Daten in einem `Array` zurückliefert (siehe Zeile 11). Wurde kein leeres `Array` erhalten, kann es dem *response*-`Array` angehängt werden (vgl. Zeile 14. Danach wird die Anfrage mit einem db.php 16 an den Client zurückgeschickt. Befindet sich weder *sync* noch *write* im *tag*, so handelt es sich um den Commit einer Benutzeränderungen. In diesem Fall wird der *tag* durch eine `switch-case`-Unterscheidung ausgewertet (Zeile 28). Da die Datenbankfunktion entweder WG-übergreifend oder benutzerspezifische Änderungen vornehmen werden in Zeile 23 die übertragene WG-, sowie User-ID aus dem *$user*-Objekt gelesen. Hier wurden nur die Fälle zum Erstellen und Bearbeiten von Blackboard-Nachrichten dargestellt. Dieser Ausschnitt genügt, um zu erkennen, dass je nach Fall unterschiedliche Parameter aus dem HTTP-POST gelesen werden. Diese werden dann in einem `Array`, Zeile 33, 39 und 44, zusammengepackt und der `commitFunc`-Methode der Datenbankschnittstelle zur Ausführung übergeben (Zeile 51). Der Grund für die Verwendung des `Arrays` waren die unterschiedlichen Parameter, welche dadurch einfacher einer Funktion übergeben werden konnten. Das Ergebnis der Funktion wird wieder als `Array` geliefert und, wie schon beim *write*, dem Rückgabeobjekt *$response* als *result* angehängt (Zeile 53).

```php
[...]
$tag = $_POST['tag'];
$response = array("tag" => $tag, "success" => 0);

$email = $_POST['email'];
$password = $_POST['password'];
$user = $db->getUserByEmailAndPassword($email, $password);
if ($user) {
 if ($tag == 'sync') {
  $table = $_POST['table'];
  $result = $db->getTable($table, $user['user_id'], $user['wg_id']);
  // put result in array
 if (isset($result) && count($result) > 0) {
      $response["result"] = $result;
      $response["success"]=count($result);
      echo json_encode($response);
 } else {
      $response["success"] = 0;
      $response["error_msg"] = "Got no results back";
 } else if ($tag &=& 'write') {
  [...]
 } else {
  $userId = $user['user_id'];
  $wgId = $user['wg_id'];
  $sqlFunc = "";
  $params = "";
  $result = 0;
  switch ($tag) {
  [...]
    case "commitBlackboardMessageAdd":
      $bbMsg = $_POST['nachricht'];
      $sqlFunc = "funcBlackboardMessageAdd";
      $params = array($wgId, $userId, $bbMsg);
    break;
    case "commitBlackboardMessageEdit":
      $newMsg = $_POST['nachricht'];
      $bbId = $_POST['blackboard_id'];
      $sqlFunc = "funcBlackboardMessageEdit";
      $params = array($bbId, $newMsg, $userId);
    break;
    case "commitBlackboardMessageRemove":
      $blackboardId = $_POST['blackboard_id'];
      $sqlFunc = "funcBlackboardMessageRemove";
      $params = array($blackboardId);
    break;
  [...]
    default:
      $result = -1;
  }
  if ($result != -1 &&
    ($result = $db->commitFunc($sqlFunc, $params)) >= 1) {
    $response["success"] = 1;
    $response["result"] = $result;
    echo json_encode($response);
  } else {
    $response["error_msg"] = "Fehler beim Commit!";
    $response["error"] = $result;
 } }
[...]
```

Listing 5.7. db.php

5.6.2 PHP-Datenbankschnittstelle

Im wesentlichen ist die PHP-Datenbankschnittstelle ebenso auf PHP-Skripten basierend wie die im Kapitel 5.6.1 beschriebene Verarbeitung der HTTP-Anfragen. Am Anfang der öffentlichen PHP-Skripte wird die Datenbankschnittstelle intitialisiert (siehe Datenbankschnittstelle initialisieren Zeile 1 f).

```
1    require_once 'include/DB_Functions.php';
2    $db = new DB_Functions();
```

Listing 5.8. Datenbankschnittstelle initialisieren

Durch das Initialiseren der `DB_Functions`-Klasse wird dessen Konstruktor 5.9 aufgerufen. Darin wird dann die Verbindung zur Datenbank aufgebaut, indem ein Objekt der Klasse `DB_Connect` erstellt und darin die Methode *connect* 5.10 aufgerufen wird.

```
1  <?php
2  class DB_Functions {
3    private $db;
4    function __construct() {
5        require_once 'DB_Connect.php';
6        $this->db = new DB_Connect();
7        $this->db->connect();
8  [...]
```

Listing 5.9. Konstruktor der DB_Functions.php

```
1  [...]
2  public function connect() {
3    require_once 'include/config.php';
4    $con = mysql_connect(DB_HOST, DB_USER, DB_PASSWORD);
5    mysql_select_db(DB_DATABASE);
6    return $con;
7  [...]
```

Listing 5.10. Konstruktor der DB_Connect.php

Sind beide Klassen erfolgreich instanziiert worden, kann mit den Methoden der `DB_Functions` gearbeitet werden. Eine besondere Methode ist die *getTable(...)* 5.11. Das Besondere daran ist, dass dieser Methode sowohl Tabellennamen als auch Prozeduren übergeben werden können. Dazu wird, wie in Zeile 2 zu sehen, im übergebenen Attribute der String *proc* gesucht. Da die Ergebnisse die selben sind, kann damit eine Redundanz des Codes vermieden werden.

Eine weitere interessante Methode ist die *commitFunc(...)* 5.12. Damit kann eine `SQL`-Funktion mit einer variablen Anzahl an Paramtern aufgerufen werden. Wie

```
1  public function getTable($table_name, $userId, $wgId) {
2    if (strrpos($table_name, "proc") === false) {
3      $query = "SELECT * FROM $table_name;";
4    } else {
5      $query = "CALL $table_name('$userId','$wgId')";
6  {...}
```

Listing 5.11. getTable() aus DB_Functions.php

in db.php 5.7 in Zeile 51 zu sehen, wird der Methode ein `Array` mit den passenden Parametern übergeben, sowie der Name der gewünschten Methode. Zunächst wird die `SQL-Query` mit einem `SELECT $dbFunc` geöffnet (siehe Zeile 2), wobei für `$dbFunc` der übergebene Funktionsname steht. Danach wird geprüft, ob überhaupt Parameter im `Array` übergeben wurden. Da die Parameter mit Kommata aneinandergereiht werden, wird der erste Parameter vor der Iteration extrahiert (vgl. Zeile 5), was die Kommasetzung erleichtert und eine Fallabfrage in der `for`-Schleife, ob es sich um den letzten Parameter handelt, spart. In der Schleife werden dann einfach die Parameter mit Komma und dem vorhandenen String konkateniert (Zeile 7) und mit einer Klammer wird die Abfrage geschlossen (Zeile 9). Mit dem Befehl `mysql_query(\ldots{})` wird dann das Statement auf der Datenbank ausgeführt und per `return` zurückgegeben (vgl. Zeile 11).

```
1  public function commitFunc($dbFunc, $params) {
2    $query = "SELECT $dbFunc ( '";
3
4    if (sizeOf($params) >= 1) {
5      $paramStr = $params[0];
6      for ($i = 1; $i < sizeOf($params); $i++ ) {
7        $paramStr .= "', '" . $params[$i];
8      }
9      $query .= $paramStr . "' ) ";
10   }
11   return mysql_query($query) or die(mysql_error());
12 }
```

Listing 5.12. commitFunc() aus DB_Functions.php

Neben den zwei gerade erläuterten Methoden besitzt die `DB_Functions` noch einige weitere Funktionen, bei denen von einer detailierten Ausarbeitung abgesehen wird. Einige Funktionen, wie zum Beispiel die Registrierung, Passwortänderung oder Erzeugung eines zufälligen Strings, wurden aus dem Programmierbeispiel ***Android Login and Registration*** [?], als auch ***Android Programming Samples*** [?] übernommen.

5.6.3 Datenbank

Da eine seperate Tabellenstruktur für jeden WG sehr umständlich und, bei vielen angemeldeten Wohngemeinschaften, sehr ressourcenhungrig wäre, werden die Daten in den Tabellen zusammengefasst. Damit müssen aber auch die Daten dynamisch an den gerade anfragenden Benutzer angepasst werden. Desweiteren ist

eine sinnvolle Aufarbeitung und Zusammenstellung der Daten notwendig, um die übertragene Menge so gering wie möglich zu halten.

Views - Datenbanksichten

Da zum einen die Übertragungsgeschwindigkeit bei mobilen Endgeräten meist etwas geringer ausfällt und zum anderen das Volumen begrenzt ist, empfiehlt es sich auf das komplette Synchronisieren der Tabellen zu verzichten. Statt die Abfragen direkt auf den Tabellen auszuführen, werden zuerst sogenannte `Views` oder `Sichten` zwischengeschaltet. Dabei handelt es sich um gespeicherte Abfragen, die wie Tabellen verwendet werden können. Da solche `Sichten` jedoch statisch sind, werden sie nur zum Zusammenfassen von verschiedenen Tabellen benutzt. Ein einfaches Beispiel einer solchen `View` ist die ViewBlackboard 5.13. Darin werden die *user*- und *blackboards*-Tabellen so vereinigt, dass eine Tabelle entsteht in der zu jeder Nachricht die Benutzer- als auch die WG-Id angezeigt werden (siehe Zeile 4) und ausgeblendete Nachrichten herausgefiltert sind (Zeile 6).

```
1 VIEW `ViewBlackboard` AS
2 SELECT DISTINCT `b`.`blackboard_id` AS `blackboard_id`,
3   `b`.`nachricht` AS `nachricht`,`b`.`crea` AS `crea`,
4   `u`.`wg_id` AS `wg_id`,`b`.`creator_id` AS `creator_id`
5 FROM (`user` `u` JOIN `blackboards` `b` ON((`u`.`wg_id` = `b`.`wg_id`)))
6 WHERE (`b`.`anzeigen` > 0);
```

Listing 5.13. ViewBlackboard

Würde die App diese `View` abfragen, so müssten entweder im `PHP`-Skript oder später in der App die, für die WG des angemeldeten Benutzers, relevanten Nachrichten herausgefiltert werden. Das würde entweder die Laufzeit auf dem Server drastisch erhöhen oder ein sehr großes Sicherheitsrisiko darstellen, da zunächst die Nachrichten sämtlicher WGs übertragen werden würden. Abgesehen vom unnötig großen Übetragungsvolumen, wurden hier Prozeduren eingesetzt. Wie in Funktion *funcBlackboardMessageRemove(...)* 5.14 zu sehen, wird ähnlich einer JAVA-Methode, mit dem Methodenkopf definiert welche Parameter erwartete werden und was sie für einen Rückgabetyp hat (vgl. Zeile 1). Anschließend wird mit BEGIN in Zeile 3 der Anfang des Funktionskörper markiert, der mit RETURN und END beendet wird (vgl. Zeile 8). Dazwischen kann jeder beliebige, valide `SQL`-Code stehen, in diesem Fall wird beim Blackboardeintrag mit der *blackboard_id* gleich der übergebenen `$blackboardId` die Spalte ANZEIGEN auf 0 gesetzt. Das führt dazu, dass die Nachricht beim Synchronisieren ausgefiltert wird (siehe Zeile 6 im Listing 5.13).

Das komplette Datenbankschema inklusiv der hier beispielhaft aufgeführten `View` und `Funktion` sind im Anhang einzusehen.

```
1  FUNCTION `funcBlackboardMessageRemove`(`$blackboardId` INT(8)) RETURNS
       tinyint(1)
2      MODIFIES SQL DATA
3  BEGIN
4    UPDATE LOW_PRIORITY blackboards
5    SET anzeigen = 0
6    WHERE blackboard_id = $blackboardId
7    LIMIT 1;
8  RETURN 1;
9  END
```

Listing 5.14. Funktion *funcBlackboardMessageRemove(...)*

5.7 Weboberfläche

Der Administrator einer WG muss in der Lage sein, die Eigenschaften der WG zu konfigurieren und die Funktionen zu verwalten. Wir haben uns dazu entschieden, eine extern jederzeit erreichbare Weboberfläche dafür bereit zu stellen. Wir hätten uns genauso gut für die Implementierung in die Android App entscheiden können, haben uns jedoch bewusst dagegen entschieden. Wir möchten mit den unterschiedlichen Programmiersprachen und den daraus resultierenden Herangehensweisen eine möglichst große Vielfalt der Informatik widerspiegeln und die Aufgaben fair auf die Teammitglieder verteilen, die bisher noch keine Erfahrung mit der Programmierung für die Android Plattform sammeln konnten.
Im folgenden Kapitel wird die Struktur der Weboberfläche mit Hilfe von einzelnen Auszügen des Quellcodes erläutert.

5.7.1 Umsetzung

Als Programmiersprache haben wir uns für die sehr beliebte und weit verbreitete Skriptsprache PHP entschieden. PHP bietet durch den Einfluss von Java, C++ und Perl einen leichten Einstieg für diejenigen, die bereits erste Erfahrungen mit einer der Programmiersprachen sammeln konnten. Ausserdem bietet PHP die einfache Umsetzung von dynamischen Webseiten und eine sehr gute Unterstützung von Datenbankverbindungen. Mit PHP ist automatisch sichergestellt, dass die Weboberfläche von jedem gängigen Browser aus in deren Desktop- sowie Mobilversion angezeigt werden kann.

Die Weboberfläche gliedert sich in neun Seiten:

```
/
├── admin.php
├── benutzer.php
├── blackboard.php
├── einkaufsliste.php
├── login.php
├── logout.php
├── putzplan.php
├── regist.php
└── system.php
```

Jede dieser PHP Dateien stellt eine Seite der Weboberfläche dar. Jede Datei beinhaltet gewöhnlichen HTML Code für die Anzeige im Browser und PHP Code für den dynamischen Teil der Datenabfrage von der Datenbank.
Mehrzeilige Abfragen in PHP sind in externe Dateien ausgelagert. So ist z.B. die Abfrage zum löschen eines WG Mitglieds in der Datei *benutzer_ edit_ delete.php* zu finden. So verfahren wir mit allen weiteren Abfragen. Dies soll die Übersichtlichkeit erhöhen.
Alle Eingaben die der Benutzer in der Weboberfläche machen kann, werden durch

Daraus ergibt sich folgende Struktur für die Weboberfläche mit ihren Seiten inklusive aller ausgelagerten PHP Abfragen:

```
/
├── admin.php
├── benutzer.php
│   ├── benutzer_aktivierung.php
│   ├── benutzer_edit_delete.php
│   ├── benutzer_script.php
│   └── benutzer_update.php
├── blackboard.php
│   ├── blackboard_add.php
│   ├── blackboard_delete_edit.php
│   └── blackboard_update.php
├── einkaufsliste
│   ├── einkaufsliste_add.php
│   ├── einkaufsliste_delete_edit.php
│   ├── waren_add_delete.php
│   └── waren_add.php
├── login
│   └── login_script.php
├── logout.php
├── putzplan.php
│   ├── putzplan_add.php
│   ├── putzplan_delete_edit.php
│   ├── putzplan_unteraufgaben_add.php
│   ├── putzplan_unteraufgaben_update.php
│   └── putzplan_update.php
├── regist.php
│   └── regist_script.php
├── system.php
│   ├── system_delete_edit.php
│   └── system_update.php
└── style.css
```

Textfelder oder Checkboxen erfasst. Wir verwenden für alle zu übertragenden Daten die Methode POST. Damit werden die Daten für die Benutzer unsichtbar im Rumpf des HTTP-Requests gesendet. Die Methode GET, die die Daten für alle sichtbar an die URI anhängen würde, könnte theoretisch ebenfalls genutzt werden. Jedoch würde das in unserem Fall ein erhöhtes Sicherheitsrisiko darstellen. Darum verzichten wie, für bis auf wenige ID's, auf die Methode GET.

Auf jeder HTML Seite und in jedem PHP Skript wird zu Beginn überprüft, ob der Nutzer im System angemeldet ist. Falls die Prüfung fehlschlägt, wird ihm mit

dem Hinweis *Bitte loggen Sie sich erst ein!* die Anzeige der Seite verwehrt und das PHP Script wird mit *exit;* gestoppt.

```
1  if(!isset($_SESSION["email"]))
2  {
3     echo("<a href=\"login.php\" />Bitte loggen sie sich erst ein!</a>");
4     exit;
5  }
```

Listing 5.15. Login des Benutzers überprüfen

Ist der Benutzer eingeloggt, wird das PHP Skript nicht gestoppt, sondern weiter verarbeitet. Mit einem *require_once 'db_inc.php* wird die Datei db_inc.php eingebunden und ausgeführt. Das passiert nur, wenn die Datei nicht schon im vorhergehenden Teil des Codes eingebunden wurde.

```
1  require_once 'db_inc.php';
```

Listing 5.16. Verbindung aufbauen falls noch nicht geschehen

Im Anschluss können die über die Methoden POST und GET übergebenden Variablen abgefragt und zwischengespeichert werden. In dem Beispiel wird mit einer if-Abfrage geprüft, ob eine Variable ëinkaufsliste_idän das PHP Skript geliefert wurde. Falls dies der Fall ist, wird der Inhalt der Variablen lokal zwischengespeichert.

```
1  if(isset($_POST['einkaufsliste_id']))
2  {
3     $einkaufsliste_id = $_POST['einkaufsliste_id'];
4  }
```

Listing 5.17. Variable *einkaufsliste_id* lockal zwischenspeichern

6

Zusammenfassung und Ausblick

In dieser Projektarbeit wurde die Umsetzung einer mobilen Applikation zur zentralen Verwaltung WG-typischer Aufgaben erläutert. Dabei wurde mit Hilfe der Entwicklungsumgebung *Android Studio* und der Programmiersprache *Java* eine Applikation implementiert, die auf allen Smartphones mit Android 4.2 oder höheren Versionen des Betriebssystems lauffähig sind. Die Daten der Nutzer werden in einer externen, über eine Internetverbindung erreichbare Datenbank gespeichert. Bei jedem Start der App werden die Daten des Benutzers aus der Datenbank geladen und bei jedem Beenden der App wieder in diese Datenbank geschrieben.

Die im Vorfeld getätigten umfangreichen Vorbereitungen, von Use-Cases-Digrammen, über Mockups bis hin zum Lastenheft, haben uns im nach hinein in der Implementierungsphase sehr gut weitergeholfen. Insbesondere durch das Lastenheft waren wir zu jederzeit in der Lage den Überblick über die Funktionen und deren Anforderungen zu behalten. Damit hatten wir zur finalen Version der Applikation eine gute Referenz zum vergleichen.

Die Datenbankverbindung hat im Vergleich zu anderen Anforderungen die höchste Aufmerksamkeit bekommen. Bereits in der ersten Vorbesprechung war uns klar, dass es bei der Arbeit mit externen Quellen zu den verschiedensten Fehlern kommen kann. Die Entwicklung eines Prototypen vor Beginn der finalen Implementierung der App, hat uns vermutlich im späteren Verlauf erheblich viel Zeit und Ärger erspart.

Leider gestaltete sich der Einstieg in die Programmierung für die Plattform *Android* schwieriger als erwartet. Am Anfang gab es Kompatitiblitätssprobleme bei der Konfiguration von *Eclipse* mit dem *Android Studio Plugin*, weshalb wir kurzerhand auf die Standalone Variante *Android Studio* umgestiegen sind. Doch auch bei dieser Software gehören bei Konfiguration und Implementierung verwirrende Mechaniken, sich wiedersprechende Anleitungen und undeutliche Fehlermeldungen fast zur Tagesordnung. Die Tastsache dass lediglich 1/3 des Teams erste grundlegende Erfahrungen in der Programmierung für *Android* sammeln konnte, hat die Situation während der Implementierungsphase weiter erschwert. Für weitere Projekte wäre zu Beginn ein Teammeeting zu empfehlen, in dem alle Mitglieder auf

den gleichen Stand der Technik und des Wissens gebrachten werden.

Eine Veröffentlichung in den *Google Play Store* und dem damit verbundenen erhöhten Arbeitsaufwand der Qualitätssicherung, dem Support der Nutzer und der Erwartung der Nutzer auf immer weitere Steigerung des Umfangs, ist zu dem jetzigen Zeitpunkt nicht geplant.

A

Glossar

DisASTer	DisASTer (Distributed Algorithms Simulation Terrain), A platform for the Implementation of Distributed Algorithms
DSM	Distributed Shared Memory
AC	Linearisierbarkeit (atomic consistency)
SC	Sequentielle Konsistenz (sequential consistency)
WC	Schwache Konsistenz (weak consistency)
RC	Freigabekonsistenz (release consistency)

B

Erklärung der Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist ...

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser: ...

Datum

Unterschrift der Kandidaten