



**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

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 auf Basis des von Google entwickelten Betriebssystems Android erreicht werden. Als Trägermedium dient hierbei ein handelsübliches Smartphone. Die App bietet jedem Mitbewohner die Möglichkeit Notizen sowie Kalendereinträge einzusehen und zu erstellen. Außerdem ist es möglich eine Einkaufsliste zu verwalten in der Artikel hinzugefügt, gelöscht und als gekauft markiert werden können. Des Weiteren zeigt ein Putzplan die noch anstehenden bzw. bereits erledigten Aufgaben aus dem WG Haushalt an. Für den Putzplan muss vor dem ersten Gebrauch der WG-Administrator, der auch unter anderem die Benutzer verwaltet, selbst definierbare Aufgaben mit Beschreibung, Zyklus und Mitbewohner anlegen.

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 calendar 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

<b>1</b>	<b>Einleitung</b> .....	1
1.1	Zielsetzung .....	2
1.2	Ähnliche Apps .....	3
<b>2</b>	<b>Vorgehensweise</b> .....	4
2.1	Projektmanagement .....	4
<b>3</b>	<b>Prototyp</b> .....	5
3.1	Prototyp Implementierung .....	5
3.1.1	Datenbankstruktur .....	6
3.1.2	PHP-Skripte .....	6
3.1.3	Prototyp-App .....	9
<b>4</b>	<b>Implementierung von <i>DorMApp</i></b> .....	17
4.1	Entwicklungsgrundlagen .....	17
4.2	App .....	19
4.2.1	Probleme .....	19
4.2.2	Lösungen .....	20
4.3	Putzplan .....	23
4.3.1	Umsetzung .....	23
4.3.2	Probleme und Lösungen .....	23
<b>5</b>	<b>Datenbank</b> .....	24
<b>6</b>	<b>PHP-Skripte</b> .....	25
<b>7</b>	<b>Zusammenfassung und Ausblick</b> .....	26
	<b>Literaturverzeichnis</b> .....	27
	<b>Glossar</b> .....	28
	<b>Erklärung der Kandidaten</b> .....	29

---

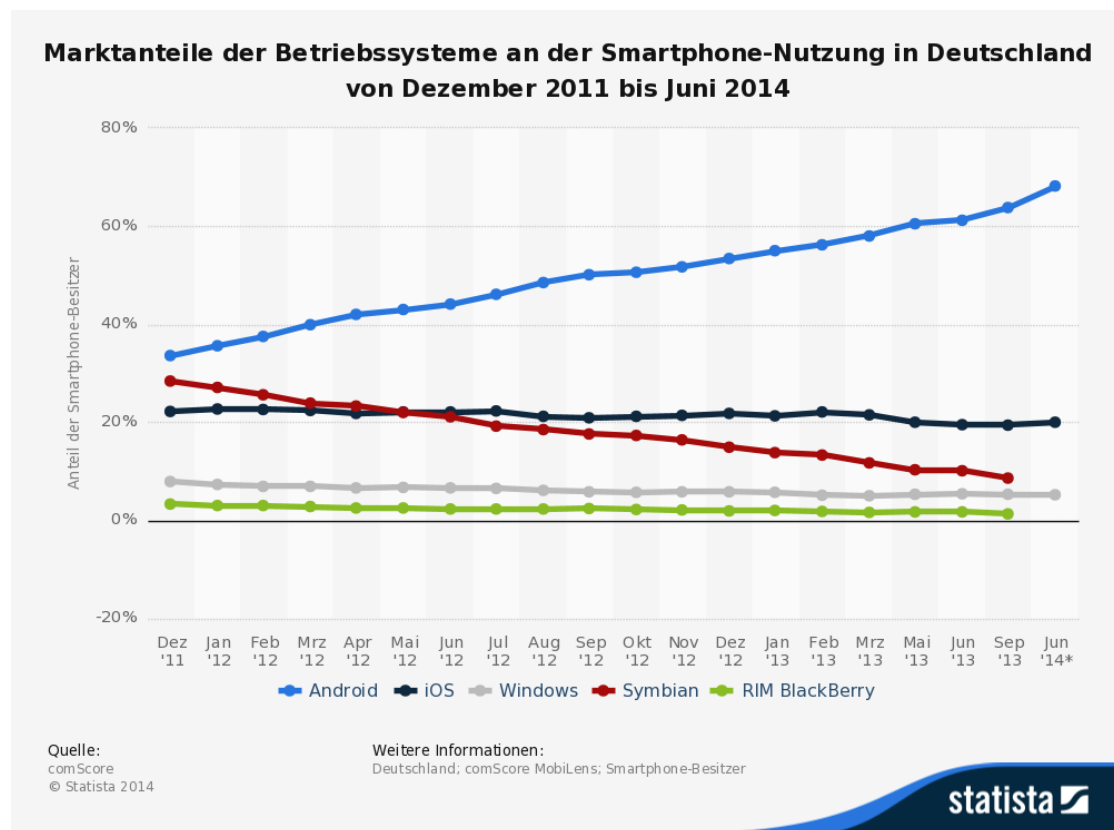
## Abbildungsverzeichnis

1.1	Marktanteile der Betriebssysteme an der Smartphone-Nutzung in Deutschland von Dezember 2011 bis Juni 2014.....	2
3.1	Aufbau der Tabelle tp_test .....	6
3.2	Aufbau der Tabelle users .....	6
3.3	Struktur der PHP-Skripte im Dateisystem .....	7
3.4	Auszug aus login.php .....	8
3.5	Auszug aus DB_Functions.php .....	9
3.6	Verzeichnisstruktur des DBPrototyp-Projekts.....	10
3.7	Start-Oberfläche .....	10
3.8	Login-Oberfläche .....	11
3.9	Auszug aus der MainActivity.java .....	12
3.10	Auszug aus der MainActivity.java .....	13
3.11	Auszug aus SyncRemoteDatabase.java .....	14
3.12	Auszug aus UserFunctions.java .....	15
3.13	Auszug aus JSONParser.java .....	16
3.14	Auszug aus der DatabaseHandler.java .....	16
4.1	Verwendung von Secure-Preferences .....	21
4.2	Auszug aus MessengerService.java .....	22

## 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 ist es Möglich eine Einkaufsliste zu verwalten, in der Artikel hinzugefügt, gelöscht und als gekauft markiert werden können. 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, weil 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% in Deutschland wird hier 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 bereit die Aufgaben des WG-Administrators zu übernehmen. Dieser WG-Administrator registriert sich und legt dabei eine neue WG 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 und wählt zu jeder Aufgabe einen Mitbewohner aus. 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 den Artikel als gekauft markieren. Alle Änderungen eines Mitbewohners ist für alle anderen Mitglieder der WG nach einer kurzen Synchronisation 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 Serverseite 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“<sup>1</sup> oder Putzpläne wie „Roomboard - Cleaning Roster“<sup>2</sup>. Jedoch gibt es eine weitere App die sich stark an unserer Idee mit dem Funktionsumfang orientiert. Die App „Flatastic: Die WG-App“<sup>3</sup> bietet neben einer Einkaufsliste, einem Putzplan und einer Pinnwand zusätzlich einen Ausgabenrechner, womit alle für die WG getätigten Einkäufe zusammengerechnet werden können. 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.

---

<sup>1</sup> „Shopping List“ in Google Play Store

<sup>2</sup> „Roomboard- Cleaning Roster“ in Google Play Store

<sup>3</sup> „Flatastic: Die WG-App“ in Google Play Store

## 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

Horizontales Prototyping mit Einflüssen aus Wasserfallmodell (dem Lastenheft)

1. Lastenheft 2. Mockups 3. Prototyp Datenbankverbindung 4. Implementierung

Horizontales Prototyping Prototyp für Datenbankverbindung Mockups für alle wichtigen Screens

Prototyp



## 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 entwickelt.

### 3.1 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.

### 3.1.1 Datenbankstruktur

Begonnen wurde die Umsetzung mit der Definition der Datenbankstruktur sowie deren Umsetzung. Dazu wurden zwei einfache Tabellen angelegt. Die Tabelle **tp\_test3.1** 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. 3.1.** Aufbau der Tabelle tp\_test

Um die Benutzerverwaltung für den Prototypen zu simulieren wurden außerdem noch eine Tabelle **users 3.2** 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.

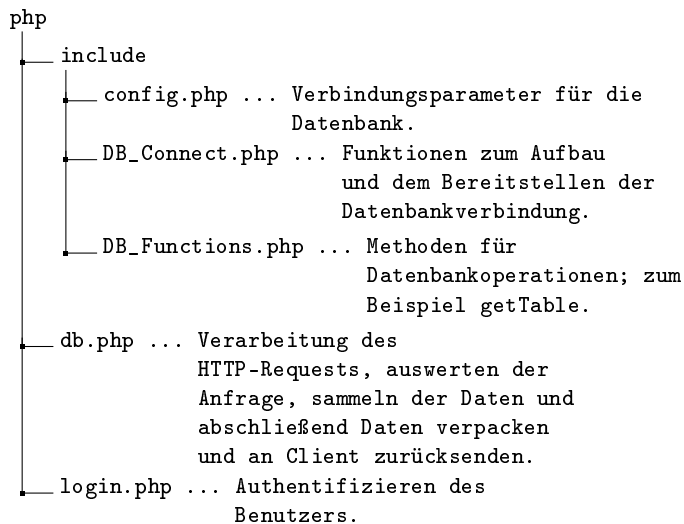
```
1 CREATE TABLE IF NOT EXISTS 'users' (  
2     'uid' INT(11) NOT NULL AUTO_INCREMENT,  
3     'unique_id' VARCHAR(23) NOT NULL,  
4     'firstname' VARCHAR(50) NOT NULL,  
5     'lastname' VARCHAR(50) NOT NULL,  
6     'username' VARCHAR(20) NOT NULL,  
7     'wgId' INT(11) NOT NULL,  
8     'email' 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. 3.2.** Aufbau der Tabelle users

### 3.1.2 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 3.3 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. 3.3.** 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 `chgpas` 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. 3.4). Sind alle Parameter korrekt übertragen worden, wird die Funktion `getUserByEmailAndPassword($email, $password)`, die sich im Skript `DB\_Functions.php` befindet, aufgerufen. Zum Authentifizieren wird aus der Datenbank der zur E-Mailadresse passende Datensatz geladen, siehe Auszug 3.5. Das in der Datenbank gespeicherte `SALT` wird mit dem übertragenen Passwort an die Funktion `checkhashSSHA($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`.

```
1 <?php
2 /**
3  * PHP API for Login, Register, Changepassword,
4  * Resetpassword Requests and for Email Notifications.
5  */
6 [...]
7 if (isset($tag) && $tag != '') {
8     // Include Database handler
9     require_once 'include/DB_Functions.php';
10    $db = new DB_Functions();
11    // response Array
12    $response = array("tag" => $tag, "success" => 0, "error" => 0);
13    // check for tag type
14    if ($tag == 'login') {
15        // Request type is check Login
16        $email = $_POST['email'];
17        $password = $_POST['password'];
18        // check for user
19        $user = $db->getUserByEmailAndPassword($email, $password);
20        if ($user) {
21            // user found
22            // echo json with success = 1
23            $response["success"] = 1;
24            $response["user"]["user_id"] = $user["user_id"];
25            $response["user"]["wg_id"] = $user["wg_id"];
26            $response["user"]["crea"] = $user["crea"];
27
28            echo json_encode($response);
29        } else {
30            // user not found
31            // echo json with error = 1
32            $response["error"] = 1;
33            $response["error_msg"] = $IncorrectEMail;
34            echo json_encode($response);
35        }
36    }
37    else if ($tag == 'chgpas') {
38        [...]
```

**Abb. 3.4.** 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['password'];
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. 3.5. Auszug aus DB\_Functions.php

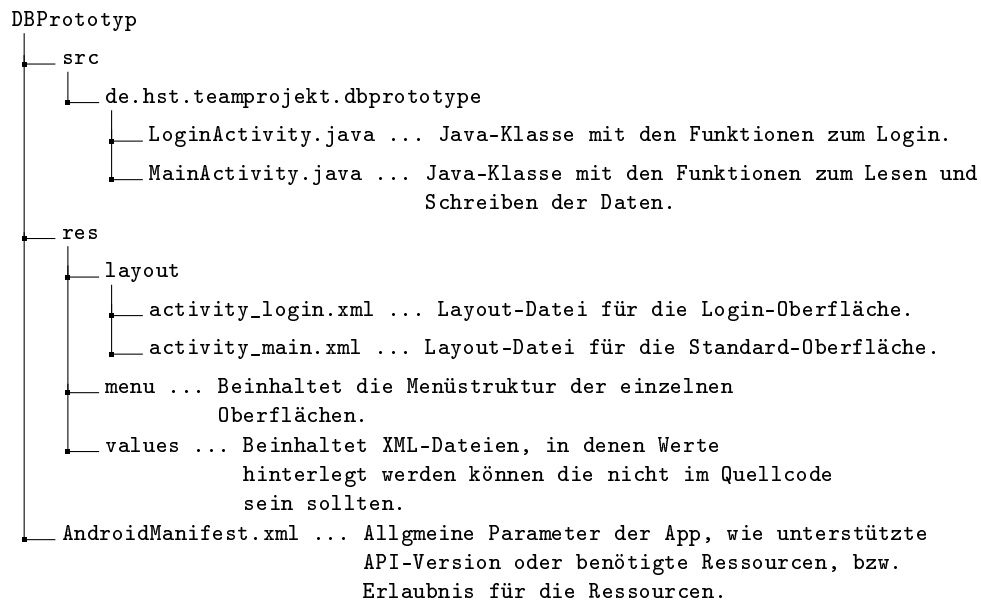
### 3.1.3 Prototyp-App

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

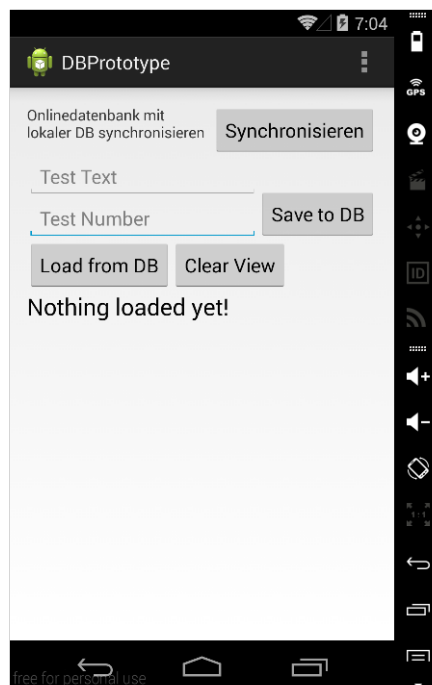
Das Projekt wurde noch in der Entwicklungsumgebung Eclipse erstellt und umfasst deshalb eine etwas tiefere Ordnerhierarchie. Da nicht alle für diese Implementierung relevant sind, werden nur die wichtigsten Ordner und Dateien näher erläutert. Der Ordnerbaum 3.6 zeigt die wichtigen Ordner und Dateien, 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



**Abb. 3.6.** Verzeichnisstruktur des DBPrototyp-Projekts

MainActivity.java mit dem Layout activity\_main.xml geladen 3.7.  
Das geschieht in Zeile 5 durch setContentView(R.layout.activity\_main) in der onCreate() Methode.



**Abb. 3.7.** 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 3.8.

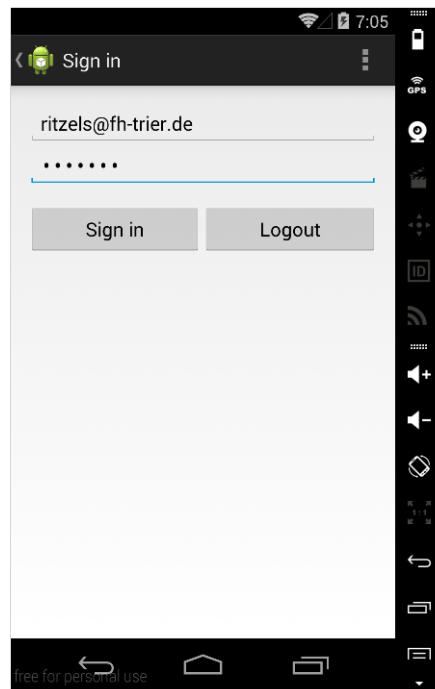


Abb. 3.8. Login-Oberfläche

Nach der erfolgreichen Authentifizierung wird über den Button **Synchronisieren** die Funktion `synchronizeDb(View view)` 7 aufgerufen. Dies generiert ein Objekt der Klasse `SyncRemoteDatabase.java` 3.11. Dabei handelt es sich um eine von `AsyncTask` abgeleitete Klasse, welche die Testdaten über das PHP-Skript aus der Datenbank abrufen und anschließend in die lokale `SQLite`-Datenbank schreibt. Diese Klasse wurde jedoch in die `DBConnectionLib` 3.1.3 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 3.1.3 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` 3.14 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. 3.9.** 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 wird 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 clearResultListView(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. 3.10. 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 3.1.3 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 Möglichkeit 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])`<sup>15</sup> aufgerufen und das damit erzeugte `JSONObject`-Objekt an die Methode `onPostExecute(JSONObject json)`<sup>19</sup> übergeben. Da diese Methode, wie die `onPreExecute`, im GUI-Thread läuft, kann der `ProgressDialog` mit einer neuen Nachricht versehen werden<sup>21</sup> 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      progressDialog = new ProgressDialog(appContext);
6      progressDialog.setTitle("Contacting Servers");
7      progressDialog.setMessage("Query database ...");
8      progressDialog.setIndeterminate(false);
9      progressDialog.setCancelable(true);
10     progressDialog.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     progressDialog.setMessage("Loading Test Data");
22     progressDialog.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. 3.11.** Auszug aus `SyncRemoteDatabase.java`

Die angesprochene Funktion `syncRemoteTable()` befindet sich in der Klasse `UserFunctions.java` 3.12. Darin wurden die Methoden zusammengefasst, 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. 3.12.** Auszug aus UserFunctions.java

Die so verpackten Parameter werden schließlich an den **JSONParser** 3.13 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* 3.11, 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 zusammengesetztes SQL-Statement in die Datenbank zu schreiben. 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         jsonObj = new JSONObject(json);
23     }
24     [...]

```

Abb. 3.13. 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. 3.14. 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 Wertvariablen 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` 3.13, 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.

## 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.

### 4.1 Entwicklungsgrundlagen

Durch die Prototyp-Entwicklung 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 in den Kapiteln Datenbank 5 und PHP-Skripte 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* [SAB14] 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* [Kai13] zurückgegriffen. Dadurch wird eine 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* [goo13], die auch als *Espresso*-Testkit bekannt ist, und *mockwebserver* [squ] 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 [gooa] mit in die Entwicklung ein und eine anfängliche Hilfestellung bei der Entwicklung bot die Seite *learn2crack.com* [Aut] konsultiert, die eine Vielzahl an Beispielprojekten und Lösungsansätzen aufweist.

## 4.2 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. Dies 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.

### 4.2.1 Probleme

Neben denen im Prototyp 3 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.

### 4.2.2 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 Instanzieren 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 Vorgehensweise erkennen. Obwohl die Entwickler keine hundertprozentige 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.getId());
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. 4.1. 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 [goob]. Mit dieser Klasse ist die Implementierung eines **gebundenen Service** einfacher als eine mit einer **AIDL**-Schnittstelle, erfüllt aber alle Voraussetzungen die für dieses Programm wichtig 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                 Log.d(TAG, "ClientService unbound");
23                 reService = null;
24                 reBound = false;
25                 break;
26             {...}
27             }
28             {...}
29         }
30         {...}
31     }
32 [...]
```

Abb. 4.2. Auszug aus MessengerService.java

## **4.3 Putzplan**

### **4.3.1 Umsetzung**

### **4.3.2 Probleme und Lösungen**





## Zusammenfassung und Ausblick

In diesem Kapitel soll die Arbeit noch einmal kurz zusammengefasst werden. Insbesondere sollen die wesentlichen Ergebnisse Ihrer Arbeit herausgehoben werden. Erfahrungen, die z.B. Benutzer mit der Mensch-Maschine-Schnittstelle gemacht haben oder Ergebnisse von Leistungsmessungen sollen an dieser Stelle präsentiert werden. Sie können in diesem Kapitel auch die Ergebnisse oder das Arbeitsumfeld Ihrer Arbeit kritisch bewerten. Wünschenswerte Erweiterungen sollen als Hinweise auf weiterführende Arbeiten erwähnt werden.

---

## Literaturverzeichnis

- Aut.     AUTHORS, VARIOUS: *Android Programming Samples*. Viele Programmierbeispiele für Android.
- gooa.    GOOGLE.COM: *Android Developer*.
- goob.    GOOGLE.COM: *Messenger*. Klassenbeschreibung von Messenger.
- goo13.   GOOGLE.COM: *android-test-kit*, 2013. Google's Testing Tools For Android.
- Kai13.    KAI, LIAO: *UndoBar*, 2013. The UI component for Android advanced UI pattern undo-bar, used in Gmail app, create by roman nurik.
- SAB14.   SCOTT ALEXANDER-BOWN, DANIEL ABRAHAM: *Secure-Preferences*, 2014. Android Shared preference wrapper that encrypts the keys and values of Shared Preferences. It's not bullet proof security but rather a quick win for incrementally making your android app more secure.
- squ.      SQUARE: *okhttp/mockwebserver*. An HTTP+SPDY client for Android and Java applications. <http://square.github.io/okhttp/>.

# 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