ulm university universität uulm

17.10.2017

Track Your Stress: Konzeption und Realisierung einer mobilen (Android) Anwendung zur Messung des Stresslevels

Bachelorarbeit

im Studiengang Informatik

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie

Institut für Datenbanken und Informationssysteme

Verfasser: Julian Haug
Matrikelnummer: 843443
1.Prüfer: Prof. Dr. Manfred Reichert
Betreuer: Dr. Rüdiger Pryss

Kurzfassung

Ein immer wichtiger werdendes Thema in unserer heutigen Gesellschaft ist Stress. Von vielen Personen unbewusst oder bewusst unterschätzt, schleichen sich stetig verschiedene Stressfaktoren und Stressauslöser in unser Leben. Sei dies im alltäglichen Leben, Zuhause, bei der Arbeit, beim Sport oder sogar im Familienurlaub. Die meisten Menschen kommen nicht darum herum, hin und wieder gestresst zu sein und stressbedingte Krankheiten werden in unserer Gesellschaft leider immer häufiger. Wie negativ sich andauernder Stress oder zu viel Stress auf die Gesundheit eines jeden Menschen auswirken kann, wird immer öfter studiert und ist auch teilweise schon erwiesen worden. Langfristig sollte es das Ziel sein, möglichst alle Stressursachen zu verringern oder gar zu eliminieren, um der Gesellschaft ein stressfreies und somit auch gesünderes Leben zu ermöglichen, denn schon in der Antike gab es die Weisheit „*Mens sana in corpore sano*“, was grob übersetzt „*ein gesunder Geist in einem gesunden Körper*“ bedeutet[1]. In Zukunft sollen daher auch Unternehmen und Arbeitgeber dazu verpflichtet sein, den Stresslevel eines Arbeitnehmers zu erfassen und auf Grundlage dieser Erhebungen den Stresslevel einer jeden Person möglichst gering zu halten.

Das Projekt *Track Your Stress* soll ein Anfang sein, um das Stresslevel einer jeden Person messen und bestimmen zu können, sowie die teilnehmenden Personen sensibler auf dieses oftmals unterschätzte Thema reagieren zu lassen. Hierfür zeigt die Arbeit, wie auf mobilen (Android) Geräten das jeweils eigene Stresslevel, unter unterschiedlichen Bedingungen und zu unterschiedlichen Zeitpunkten, wahrgenommen wird und wie dies visualisiert werden kann. Dies wird im Detail vorgestellt, sowie aus technischer Sicht betrachtet. Weiter wird aufgezeigt, wie ein solches Projekt architektonisch zu realisieren ist. Zusätzlich wird in dieser Arbeit auf spezielle Implementierungsthemen eingegangen. Am Ende entstand eine voll funktionsfähige mobile Anwendung zur Unterstützung der Messung des Stresslevels einer jeden Person bzw. eines jeden Nutzers der mobilen Anwendung. Des Weiteren kann das Projekt in dieser Form auch auf andere Zwecke übertragen und genutzt werden, wie z.B. zur Ermessung und Unterstützung bei chronischen Erkrankungen, bezüglich deren Symptome und daraus entstehenden Einschränkungen etc..

[1] [3]

Inhalt

Kurzfassung 1
Abbildungsverzeichnis 3
Listingverzeichnis 4
1. Einleitung 5
1.1. Stress 5
1.2. Motivation 5
1.2.1. Vision 6
1.2.2. Projektkontext 6
1.3. Aufbau der Arbeit 6
2. Anforderungen 7
2.1. Funktionale Anforderungen 7
2.2. Nichtfunktionale Anforderungen 8
3. Architektur 9
3.1. Architekturübersicht 9
3.2. Genereller Ablauf 9
3.3. Datenstruktur/Datenmodell 12
3.3.1. Datenmodell der App / ViewModel + ApiController 13
3.4. Controller- Manager- und Handlerklassen 16
3.5. Lokale Datenbank 18
3.6. Views 19
3.6.1. ListAdapter 21
3.7. Zusammenfassung der Struktur 22
4. Vorstellung des Track Your Stress Rahmenwerks 23
4.1. Vorstellung der App 23
4.1.1. Anmeldung mit Username und Passwort 23
4.1.2. Registrierung 23
4.1.3. Passwort vergessen / Passwort zurücksetzen 24
4.1.4. Ansicht aller Fragebögen / Die Fragebögen-Liste 25
4.1.5. Ausfüllen von (statistischen) Fragebögen 26
4.1.6. Antwortmöglichkeiten eines (statistischen) Fragebogens 27
4.1.7. Alle Studien 29
4.1.8. Benachrichtigungen, Sprache ändern, Ergebnisse, Logout 30
4.1.9. Offline Modus 32
5. Implementierung 33
5.1. Alle Studien 33

6. Anforderungsabgleich 44
6.1. Funktionale Anforderungen 44
6.2. Nichtfunktionale Anforderungen 45
7. Vergleichbare Projekte 46
7.1. Track Your Tinnitus (TYT) 46
7.2. Assess Your Stress (AYS) 46
7.3. MyKind 47
8. Fazit 48
8.1. Zusammenfassung 48
8.2. Ausblick 48
8.2.1. Verbesserung am User-Interface und Hilfe 48
8.2.2. Verbesserter Offline-Modus 49
8.2.3. Benachrichtigungen 49
Literaturverzeichnis 50

Abbildungsverzeichnis

Abbildung 1: Grundsätzliche Komponenten 9
Abbildung 2: Genereller Ablauf 10
Abbildung 3: Kommunikation von Activity- ApiController- ApiConnectionTask 11
Abbildung 4: LoadingDialog und parallele AsyncTasks 12
Abbildung 5: ModelView der App 14
Abbildung 6: Questionmodel 15
Abbildung 7: Datenmodel/ViewModel generieren mittels GSON 16
Abbildung 8: Request und Result 17
Abbildung 9: Struktur lokale Datenbank 18
Abbildung 10: Unterschied Kommunikation Offline vs. Online 19
Abbildung 11: Views Vererbung AppCompatActivityEx 20
Abbildung 12: Views Vererbung MenuActivity 20
Abbildung 13: Views Vererbung EBase 21
Abbildung 14: Model-ModelView + Controller-View 22
Abbildung 15: Login Ansicht 23
Abbildung 16: Registrierungsansicht 24
Abbildung 17: Passwort vergessen 25
Abbildung 18: Fragebögen-Liste 25
Abbildung 19 : Fragebögen 27
Abbildung 20: TextDate 28
Abbildung 21: Pflichtfragen 29
Abbildung 22: Alle Studien 30
Abbildung 23: Benachrichtigungen, Sprache, Ergebnisse 31
Abbildung 24: Offline Modus 32
Abbildung 25: Studien: Status und Statuswechsel 34

Listingverzeichnis

Listing 1: Study 34
Listing 2: StudyEx 35
Listing 3: StudiesActivity 36
Listing 4: StudyListAdapter 38
Listing 5: ApiController Studies Funktionen 40
Listing 6: ApiCallback Studies 41
Listing 7: Weitere ApiController Studies Funktionen 42

1. Einleitung

Dieses Kapitel umschreibt zuerst kurz den Begriff „Stress“ und warum dies in unserer heutigen Gesellschaft ein so wichtiges Thema ist. Danach wird erläutert, warum die mobile Anwendung *Track Your Stress* diesbezüglich von Bedeutung ist und am Ende wird ein kurzer Überblick über den Aufbau dieser Arbeit gegeben.

1.1. Stress

Laut dem deutschen Duden wird Stress als eine *„erhöhte Beanspruchung, Belastung physischer oder psychischer Art“* beschrieben[2]. Stress ist schon fast eine „Volkskrankheit“ und jeder kennt das Gefühl gestresst zu sein. Oftmals reagieren Menschen gereizt oder angespannt. Stress kann unter gewissen Umständen fördernd sein, z.B. hilft er dem Körper in Extremsituationen die Spannung zu halten, wie es etwa ein Marathonläufer am Ende seines Rennens benötigt. Unter dauerhaftem Stress lässt die Leistung aber nach und im schlimmsten Fall schadet es auf Dauer der Gesundheit. Dies kann so weit führen, dass Betroffene einen sogenannten „Burnout“ oder Ähnliches durchleiden müssen[3]. Patienten mit stressbedingten Erkrankungen benötigen professionelle Unterstützung und Behandlung, welche oftmals mit erheblichen Kosten verbunden ist, ganz zu schweigen von dem oftmals langwierigen Leidensweg der Patienten. In Deutschland z.B. empfinden ca. sechs von zehn Personen ihr Leben als stressig und jeder Fünfte steht angeblich unter andauerndem Druck[4]. Ziel ist es also, ein möglichst stressfreies Leben zu ermöglichen.

1.2. Motivation

Dieses Projekt wird im Rahmen meiner Bachelorarbeit meines Informatikstudiums an der Uni Ulm entwickelt. Auch auf Grund steigender Anzahl an Krankentagen in Unternehmen und Organisationen, sowie steigender Anzahl an Erkrankungen psychisch bedingter Krankheiten, hat der Bundestag 2015 das Präventionsgesetz verabschiedet. Dieses sieht verschiedene Präventionsmaßnahmen für die Erhaltung und Förderung der Gesundheit von Kindern, Jugendlichen und Erwachsenen vor. Vor allem auf individuelle Belastungen und Risikofaktoren soll in Zukunft verstärkte Aufmerksamkeit gelegt werden[5]. Weiter sind Arbeitgeber nach dem Arbeitsschutz-Gesetz §5,6 dazu verpflichtet Maßnahmen zu ergreifen, um Erkrankungen vorzubeugen. „Besonders wichtig ist Paragraf 5, der Arbeitgeber verpflichtet, eine Gefährdungsanalyse für die Arbeitsplätze in ihrem Unternehmen durchzuführen. Zu den maßgeblichen Risiken zählt der Paragraf auch ausdrücklich Stress.“ [6]. Da übermäßiger Stress oftmals als Ursache von psychischen Erkrankungen vermutet wird oder einen großen Teil der Ursache ausmacht, ist es vor allem für die Unternehmen von recht großer Bedeutung, Stressursachen auf den Grund zu gehen. Um im Endeffekt solchen stressbedingten Erkrankungen vorzubeugen oder auch um eine genaue Gefährdungsanalyse durchzuführen, ist es zuallererst nötig das Stresslevel von Arbeitnehmer/Personen unter alltäglichen Bedingungen und Gegebenheiten festzustellen und zu untersuchen. Diese mobile Anwendung soll ein erster Schritt in diese Richtung sein, um das Stresslevel eines jeden Einzelnen zu bestimmen.

Hierzu wird im Rahmen dieser Arbeit eine mobile (Android) Anwendung zur Messung des Stresslevels erstellt, auf der das Stresslevel von Nutzern, durch kontinuierliches, aber zeitlich zufälliges Beantworten von Fragen und Tests, erfasst werden kann. Die Anwendung ist ein Versuch, den Stresslevel von Personen unter alltäglichen Bedingungen zu bestimmen, zu messen und zu untersuchen. Darüber hinaus soll die Anwendung eventuelle Ursachen und Hintergründe eines

---

[2] [16]
[3] [8]
[4] [33]
[5] [9]
[6] [22]

hohen Stresslevels anhand der beantworteten Fragebögen identifizieren, um diese in Zukunft möglichst zu verringern oder komplett zu egalisieren.

1.2.1. Vision

Im Idealfall ist das System/die mobile Anwendung einfach aufgebaut und benutzerfreundlich zu bedienen. Mit dieser Anwendung soll gezielt das Stresslevel eines/r jeden Arbeitnehmers/Person erfasst und gemessen werden. Weiterhin soll es dazu führen, dass die Nutzer eine gesündere, kritischere und bewusstere Selbsteinschätzung, in Bezug auf oftmals unterschätzten, unterbewussten Stress erlangen. Alles in Allem soll die Anwendung dazu beitragen, über das ermessene Stresslevel, stressbedingten Erkrankungen vorzubeugen, Krankentage zu verringern und für zufriedene Arbeitnehmer und Arbeitgeber zu sorgen.

1.2.2. Projektkontext

Aufgrund der gesetzlichen Verpflichtungen von Arbeitgebern, Maßnahmen zu ergreifen, um Krankheiten vorzubeugen und Gefährdungsanalysen durchzuführen, besteht die Nachfrage an Instrumenten um diese Maßnahmen umzusetzen. Eine mobile Anwendung zur Messung des Stresslevels ist somit einer der ersten Schritte, um mögliche Maßnahmen zur Stressvermeidung zu realisieren. Es ist also mehr als sinnvoll eine solche mobile Anwendung zu entwickeln, gerade im Smartphone-Zeitalter und zunehmendem Stressempfinden der Bevölkerung.

1.3. Aufbau der Arbeit

Dieser Abschnitt stellt eine Übersicht der Struktur dieser Arbeit dar. Sie ist in acht Kapitel unterteilt. Nach der eben erfolgten Einleitung folgt zuerst die Erläuterung der Anforderungen der Anwendung *Track Your Stress* (Kapitel 2). In Kapitel 3 erfolgt die Darstellung der Architektur des Projekts, wobei eine allgemeine Übersicht der Architektur, sowie Datenstruktur und Aufbau der mobilen Anwendung gegeben wird. Im 4. Kapitel wird das Projekt an sich vorgestellt und die App beschrieben. Es wird darauf eingegangen, welche Funktionen dem Benutzer zur Verfügung stehen und wie er diese nutzen kann. Im darauffolgenden Kapitel wird auf die Implementierung des Projektes eingegangen, sowie auf etwaige, spezielle Besonderheiten und Schwierigkeiten (Kapitel 5). Im Anschluss werden die gesetzten Anforderungen mit dem Stand der Entwicklungen abgeglichen (Kapitel 6). In den letzten beiden Kapiteln wird zum einen auf ähnliche, vergleichbare Projekte eingegangen (Kapitel 7) und zum anderen eine kurze Zusammenfassung, sowie ein Ausblick auf mögliche Weiterentwicklungen des Projekts gegeben (Kapitel 8).

2. Anforderungen

Dieses Kapitel definiert die Anforderungen an das *Track Your Stress* Projekt. Dabei sind diese Anforderungen in funktionale und nichtfunktionale Anforderungen unterteilt.

2.1. Funktionale Anforderungen

Nummer	Beschreibung	Problembeschreibung
1	Kontinuierliche Fragebögen zur Überwachung des Stresslevels	Die App sollte verschiedene Fragebögen anzeigen, mit denen ein Benutzer in bestimmten Abständen, seinen von ihm persönlich gefühlten Stress angeben und überwachen kann.
2	(Demographischer) Fragebogen zur Überwachung des Stresslevels	Die App sollte (zu Beginn) einen einmalig auszufüllenden Fragebogen anzeigen, mit dem verschiedene, für die Überwachung des Stresslevels relevante Daten des Nutzers erhoben werden.
3	Anzeige der möglichen Ergebnisse	Um die zeitliche Entwicklung in der App direkt anzeigen zu können, sollten die Ergebnisse aus dem Fragebogen (zur Überwachung und Erhebung des Stresslevels) visualisiert werden und dem Nutzer angezeigt werden können.
4	Registrierung in der App	Die Benutzung der App kann ohne Benutzerkonto nicht erfolgen. Es sollte allerdings möglich sein, direkt auf einem (Android) Gerät ein solches Benutzerkonto zu erstellen.
5	Sliderfragen ohne initialen Wert	Ein Benutzer lässt sich womöglich beim Ausfüllen eines Fragebogens davon beeinflussen, welcher Wert voreingestellt ist. Daher darf der Slider in einem Fragebogen in der Track Your Stress App keinen initialen Wert haben.
6	Ausfüllen der statistischen Fragebögen in der App	Da das Ausfüllen der statistischen Fragebögen Voraussetzung für die Benutzung der App ist, sollte dies in der App möglich sein.
7	Synchronisierung der Ergebnisse	Zur Visualisierung der Ergebnisse aus dem Fragebogen, zur Überwachung sowie Messung des Stresslevels und für Forschungszwecke, sollten diese Ergebnisse aus der App an den Server übertragen werden.
8	Meldung von Benachrichtigungen	Ein Benutzer sollte, zu von der Datenbank vorgegebenen Zeiten, Benachrichtigungen auf dem Gerät erhalten, die den User dazu auffordern, entsprechende Fragebögen zu bearbeiten.
9	Messung des Geräuschpegels	Um festzustellen, ob der Stress von den Umgebungsgeräuschen überdeckt oder beeinflusst wird, sollte die App während des Ausfüllens des Fragebogens zur Überwachung der Schwankungen der Stresswahrnehmung

		den Pegel der Hintergrundgeräusche messen können.
10	App auch ohne Internetverbindung nutzbar machen	Eine funktionierende Internetverbindung auf dem Smartphone sollte keine Voraussetzung für das Benutzen der App sein, da ein Benutzer evtl. nur schlechten oder gar keinen Empfang haben kann.
Studienteilnahme		
11	An verschiedenen Studien teilnehmen	Ein User sollte die Möglichkeit haben, sich für verschiedene Studien einzuschreiben und dadurch an ihnen teilzunehmen. Dabei können Studien verschiedene Zustände haben: privat, offen. Offenen Studien kann direkt beigetreten werden, private erfordern einen speziellen Aufnahmeprozess.
12	Studieneinladung annehmen	Ein User sollte eine Einladung zu einer privaten Studie annehmen und dadurch dieser privaten Studie beitreten können.
13	An privater Studie teilnehmen	Der User kann sich in der App in eine private Studie einschreiben, sofern er das entsprechende Passwort der Studie kennt.

2.2. Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen definieren die Anforderungen an das Projekt im Hinblick auf Aussehen, Handhabung und Datenschutz. Die folgende Tabelle zeigt die nichtfunktionalen Anforderungen an das *Track Your Stress* Projekt

Nummer	Beschreibung	Problembeschreibung
1	Ausfüllen des Fragebogens zur Überwachung der Stresswahrnehmung sollte in möglichst kurzer Zeit gelingen	Das Ausfüllen der kontinuierlichen Fragebögen zur Überwachung der Stresswahrnehmung wird vom Benutzer mehrmals am Tag/ mehrmals in der Woche/ mehrmals im Monat ausgefüllt. Daher sollte dieser Vorgang möglichst schnell zu erledigen sein.
2	Benutzbarkeit	Die App sollte einfach und intuitiv, ohne große Erklärungen verstanden und genutzt werden können. Falsche Eingaben sollten im besten Fall erkannt, abgefangen und dem Nutzer zurückgemeldet werden, so dass dieser reagieren kann.

3. Architektur

Dieses Kapitel beschreibt die Architektur des *Track Your Stress* Projekts. Zuerst wird eine kurze Übersicht über die Architektur gegeben (Kapitel 3.1). Danach wird ein typischer Ablauf angeführt (Kapitel 3.2), gefolgt von der Beschreibung der Datenstruktur (Kapitel 3.3). Im Weiteren werden die Controller- , Manager- , und Handlerklassen, wie z.B. der *ApiController* kurz erläutert (Kapitel 3.4). Im vorletzten Kapitel (3.5) wird auf die lokale Datenbank eingegangen, welche eine Offline-Nutzung der App ermöglicht. Am Ende dieses Kapitels werden noch die *Views* (*Activities*) der App erklärt (Kapitel 3.6).

3.1. Architekturübersicht

Track Your Stress besteht bisher generell aus zwei Komponenten: Server und mobiler Android Anwendung. Der Server bzw. das gesamte Backend (*API* inkludiert) war bereits von Seiten der Universität gestellt. Die App wurde nativ mit Java auf Android entwickelt. Abbildung 1 zeigt die Verknüpfungspunkte der einzelnen Komponenten. Die App greift nicht direkt auf die Daten auf dem Server zu, sondern jegliche Kommunikation zwischen App und Server läuft über eine *JSON-API*.

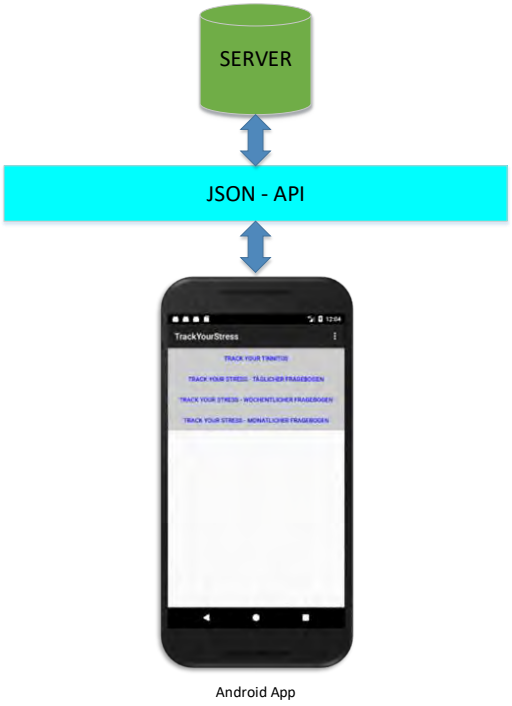

Abbildung 1: Grundsätzliche Komponenten

3.2. Genereller Ablauf

Die Überwachung der Stresswahrnehmung in der Android App setzt ein Benutzerkonto voraus. *Track Your Stress* bietet einem Benutzer die Möglichkeit, ein Benutzerkonto in der App zu erstellen. Abbildung 2 zeigt einen typischen Ablauf. Wenn ein Benutzer die App das erste Mal öffnet, muss er sich zunächst registrieren. Nach der Eingabe aller erforderlichen Daten (Benutzername, E-Mail-Adresse und Passwort) erhält der Benutzer eine Bestätigung per E-Mail. In dieser E-Mail befindet sich ein Link, mit dem er sein Benutzerkonto aktivieren kann. Erst nach erfolgreicher Aktivierung ist eine Anmeldung möglich. Im Benutzerbereich findet ein Benutzer zuallererst die Fragebögen, welche seiner Studie, in diesem Fall zur Überwachung der Stresswahrnehmung, zugeordnet sind (Sollte er mehreren Studien beigetreten sein, so werden ihm hier alle Fragebögen seiner Studien angezeigt; das Schema ist aber immer dasselbe). Sind die Einmaligen z.B. der demographische Fragebogen der

Stressstudie, bereits bearbeitet und abgeschickt worden, so werden ihm dort seine kontinuierlich wiederkehrenden Fragebögen zu Stresswahrnehmung angezeigt. Die verschiedenen verfügbaren Studien erreicht der Nutzer über das Menü. Dort kann man sich für gewünschte Studien einschreiben, ausschreiben oder auch eine private Studie anfragen. Weiter kann der Nutzer über das Menü die Benachrichtigungseinstellungen überprüfen, die Sprache ändern oder auch seine Ergebnisse von bereits abgegebenen Fragebögen einsehen. Wenn bereits ein Benutzerkonto vorhanden ist, ist ein direkter Login möglich. Die App überprüft, ob alle einmaligen Fragebögen bereits beantwortet sind. Wenn nicht, müssen diese in der App ausgefüllt werden, bevor die eigentliche Studie mit den kontinuierlichen Fragebögen zur Stresswahrnehmung beginnen kann. Sind die einmaligen Fragebögen beantwortet, kann auch direkt mit den immer wiederkehrenden Fragebögen begonnen werden.

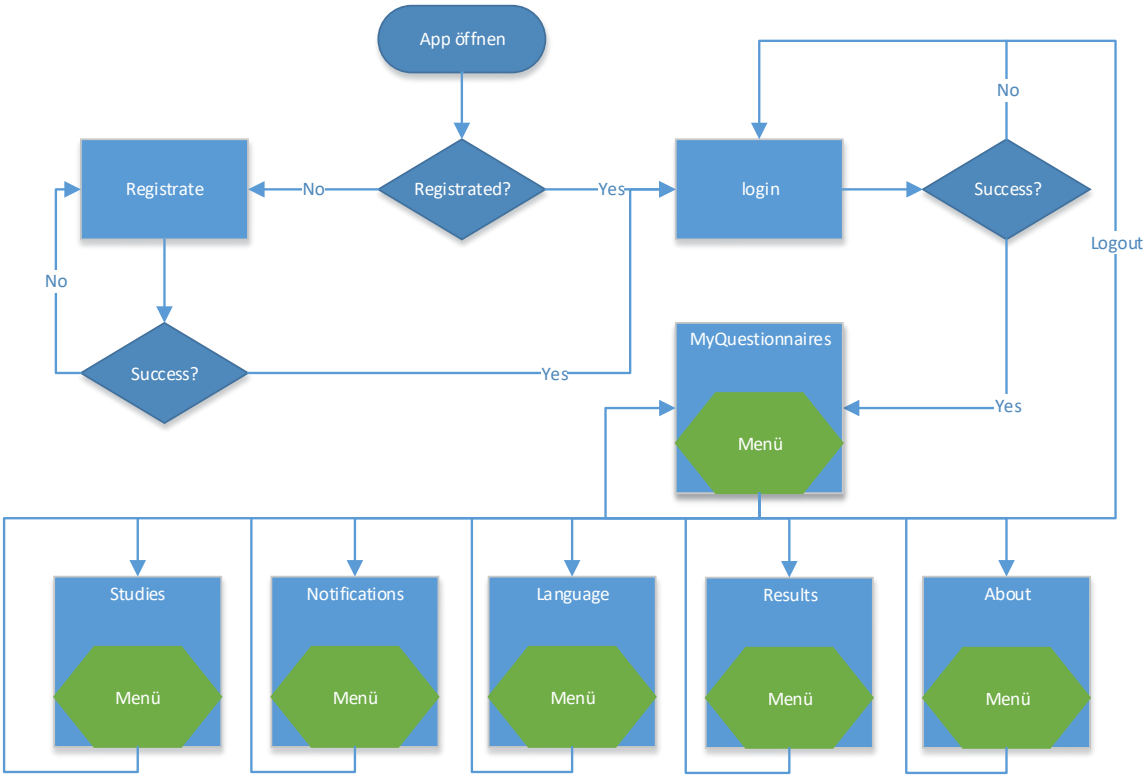

Abbildung 2: Genereller Ablauf

Sofern eine Internetverbindung besteht, laufen die Anfragen und Sendungen generell immer nach dem gleichen Schema ab. In Abbildung 3 wird dies anhand des *Logins* schematisch angedeutet.

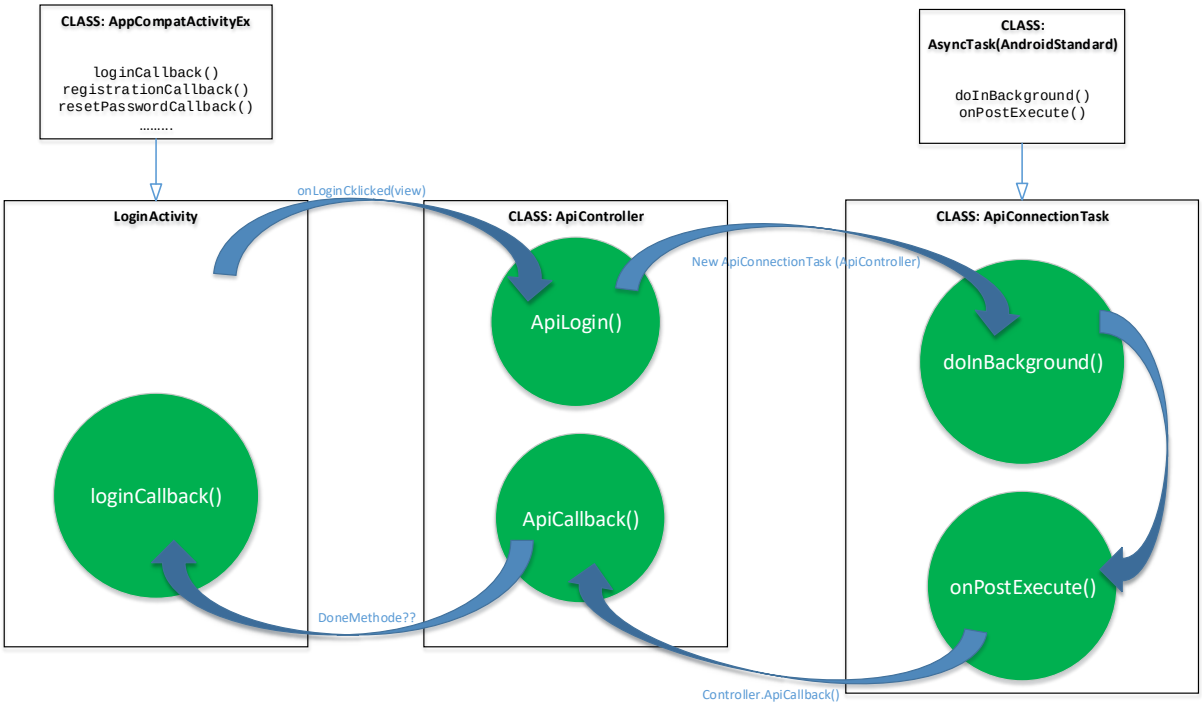

Abbildung 3: Kommunikation von Activity- ApiController- ApiConnectionTask

Durch eine entsprechende Aktion des Users (hier *Login*), wird zuerst ein neues Objekt vom Typ *ApiController* erzeugt. Dieser bekommt die aktuelle *Activity* übergeben, um später vom *AsyncTask* zur richtigen *Activity* zurückzukehren. Der *ApiController* stößt die entsprechende Funktion (hier die *Login*-Funktion) an, in welcher die *ApiConnection* geöffnet und ein *AsyncTask* gestartet wird. Nach der Funktion *doInBackground()* – in welcher die eigentliche Kommunikation mit der *API* erfolgt – wird mit der *onPostExecute()*-Funktion der *ApiCallback*() angestoßen. Dort wird anhand der *doneMethod* unterschieden, welche *CallbackFunktion* (hier die *loginCallback()*) aufgerufen werden muss. Anhand der zu Anfang mitgegebenen *Activity* ist nun auch klar, zu welcher *Activity* zurückgesprungen werden soll. Generell können mehrere Tasks parallel nebeneinander laufen, wie z.B. beim Laden mehrerer Fragebögen auf einmal. Durch den entsprechenden Pointer auf die entsprechende *Activity* ist jedoch immer klar, zu welcher *Activity* das Ergebnis des *AsyncTasks,* über die *ApiCallback*-Funktion, zurückkehren muss.

Wird etwas von der Datenbank, über die *API* geladen oder gesendet, und sind womöglich mehrere *AsyncTasks* parallel gestartet, so wird dies dem User, durch entsprechende Dialogfenster (z.B. *Loading*, *Sending*) ebenso mitgeteilt, wie auch erfolgreiche Lade- bzw. Sendevorgänge, oder auch gescheiterte Kommunikationen. Dem User wird dies visualisiert, wie es in Abbildung 4 auf dem Smartphone zu sehen ist. Ersichtlich wird auch, dass mehrere *AsyncTasks* echt parallel ablaufen. Der *LoadingDialog* bleibt hier, beim Laden der Fragebögen und der zugehörigen Fragen, solange bestehen, bis im *ApiController* alle gestarteten *AsyncTasks* auch wieder zurückgekommen sind. Erst dann erfolgt – vom *ApiController* aus – der Callback zur *Activity*. Analog oder nur unwesentlich unterschiedlich sind jegliche Kommunikationen mit der *API* aufgebaut. Dabei spielt es keine Rolle, ob es sich um einen Sendevorgang oder Ladevorgang handelt.

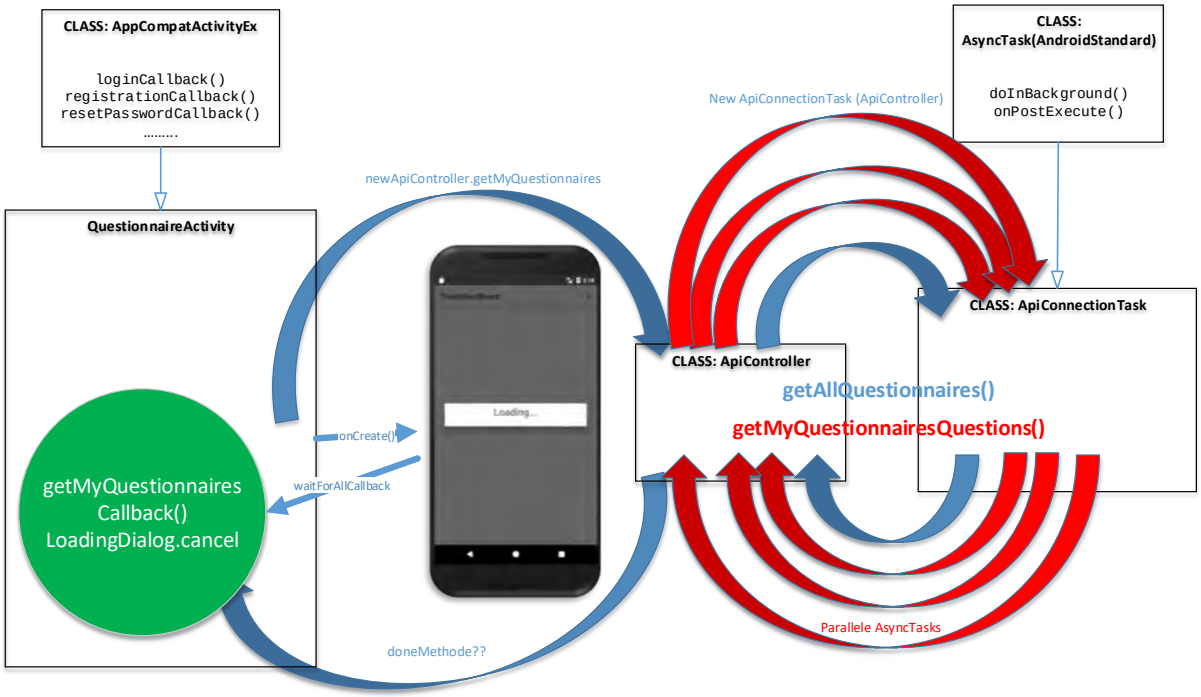

Abbildung 4: LoadingDialog und parallele AsyncTasks

3.3. Datenstruktur/Datenmodell

Die generelle Datenstruktur wird von der *API* und der zugehörigen Datenbank vorgegeben. Die App ist nach *dem Model-View-Controller* Prinzip aufgebaut. Genauer wäre es den Aufbau als *Modell-ModelView (mit ApiController)-View* zu bezeichnen. Die Daten zu Fragebögen, Studien etc. liegen in der *API* als *JSON*-Objekte vor. Aus diesen *JSON* „Paketen“ wird mittels der *GSON*-Class das Datenmodell typisiert für die App *Track Your Stress* generiert. Strukturell anschaulicher erklärt wird dies auch in Abbildung 7. Für die Generierung – mittels *GSON* – aus der *JSON-API* und evtl. zurück, wurden auf Grundlage der gegebenen *JSON*-Objekte folgende Klassen wie folgt definiert (Abbildung 5):

3.3.1. Datenmodell der App / ViewModel + ApiController

```java
public class Locales {

    public String type;
    public String id;
    public Attributes
attributes;
    public Links links;

    public class Attributes {
        public String locale;
        public String title;
    }
    public class Links {
        public String self;
    }
}
```

```java
public class ApiText {


    public String type;
    public String id;
    public Attributes attributes;
    public Links links;


    public class Attributes{
        public String name;
        public String identifier;
        public String text;
    }
    public class Links {
        public String self;
    }
}
```

```java
public class Answersheet
{
    public String type;
    public String id;
    public Attributes attributes;
    public Meta meta;
    public Links links;

        public class Attributes
        {
        public int questionnaire_id;
        public long collected_at;
        }
        public class Links
        {
        public String self;
        }
        public class Meta
        {
            public int likes;
            public int avg;
        }
}
```

```java
public class Feedback {
    public String id;
    public boolean result;
    public String rule;
    @SerializedName("true")
    public True truth;
    @SerializedName("false")
    public False wrong;

    public class True{
        public String headline;
        public String text;
    }

    public class False{
        public String headline;
        public String text;
    }
}
```

```java
public class Study {

    public String type;
    public String id;
    public Attributes attributes;

    public class Attributes {
        public String name;
        public String title;
        public String description;
        public String accesstype;
        public int starts_at;
        public int ends_at;
        public int is_private;
        public boolean is_running;
        public String text;
        public String consenttext;
    }
}
```

```java
public class Answer {
    public int id;
    public String qId;
    public String email;
    public String jsonString;
    public int posted;

    public Answer() {
    }
}
```

```java
public class MessageTime {
    public int id;
    public long time;
    public int every;

    public MessageTime(int id, long time, int every) {
        this.id = id;
        this.time = time;
        this.every = every;
    }
    public MessageTime() {
    }
}
```

```java
public class QuestionnaireEx {
    public Questionnaire questionnaire;
    public List<Question> questions;
}
```

```java
public class StudyEx {
    public String state;
    public Study study;
    public boolean isClicked = false;
}
```

```java
public class Questionnaire {

    public String type;
    public String id;
    public Attributes attributes;
    public Links links;

    public class Attributes {
        public String name;
        public String title;
        public String origin;
        public int is_active;
        public List<Can_use> can_use;
        public int is_onetime;
        public int is_multiple;
        public boolean is_filled_out;
        public List<Schedule> schedule;
        public int is_schedule_changeable;
        public String description;
        public String introtext;
        public String outrotext;

        public class Can_use{
            public String name;
            public String type;
            public int value;
        }
        public class Schedule{
            public String type;
            public int every;
            public int start_day;
            public String start;
            public String end;
            public int amount;
            public String at;
        }
    }
    public class Links {
        public String self;
    }
}
```

```java
public class NoiseRecorder {

    String name = "microphone";
    long collected_at;
    double loudness;

    public NoiseRecorder(long collected_at, double loudness) {
        this.collected_at = collected_at;
        this.loudness = loudness;
    }
}
```

```java
public class MySchedule {
    public int id;
    public String qID;
    public String qTitle;
    public int is_changeable;

    public Questionnaire.Attributes.Schedule schedule;

    public MySchedule(String qID, String qTitle, Questionnaire.Attributes.Schedule schedule) {
        this.qID = qID;
        this.qTitle = qTitle;
        this.schedule = schedule;
        this.id = 0;
    }
```

```java
public class QuestionMultipleChoice {
    public String type;
    public String id;
    public Attributes attributes;
    public Links links;

    public class Attributes {
        public Content content;
        public String name;
        public String elementtype;

        public class Content{
            public String question;
            public List<String> answers;
            public String label;
            public String questiontype;
            public List<String> values;
            public int required;
        }
    }
    public class Links {
        public String self;
    }
}
```

```java
public class QuestionSingleChoice {
    public String type;
    public String id;
    public Attributes attributes;
    public Links links;

    public class Attributes {
        public Content content;
        public String name;
        public String elementtype;

        public class Content{
            public String question;
            public List<String> answers;
            public String label;
            public String questiontype;
            public List<String> values;
            public int required;
        }
    }
    public class Links {
        public String self;
    }
}
```

```java
public class QuestionTextDate {
    public String type;
    public String id;
    public Attributes attributes;
    public Links links;

    public class Attributes {
        public Content content;
        public String name;
        public String elementtype;

        public class Content{
            public String question;
            public List<String> answers;
            public String label;
            public String questiontype;
            public List<String> values;
            public int required;
        }
    }
    public class Links {
        public String self;
    }
}
```

```java
public class QuestionSlider {
    public String type;
    public String id;
    public Attributes attributes;
    public Links links;

    public class Attributes {
        public Content content;
        public String name;
        public String elementtype;

        public class Content{
            public String question;
            public List<Answers> answers;
            public String label;
            public String questiontype;
            public Values values;
            public int required;

            public class Answers{
                public int value;
                public String label;
            }

            public class Values {
                public int min;
                public int max;
                public int step;
            }
        }
    }
    public class Links {
        public String self;
    }
}
```

```java
public class QuestionHeadline {
    public String type;
    public String id;
    public Attributes attributes;
    public Links links;

    public class Attributes {
        public Content content;
        public String name;
        public String elementtype;

        public class Content{
            public String headline;
        }
    }
    public class Links {
        public String self;
    }
}
```

```java
public class QuestionSamScale {
    public String type;
    public String id;
    public Attributes attributes;
    public Links links;

    public class Attributes {
        public Content content;
        public String name;
        public String elementtype;

        public class Content{
            public String question;
            public List<String> answers;
            public String label;
            public String questiontype;
            public Values values;
            public int required;

            public class Values {
                public int min;
                public int max;
                public int step;
            }
        }
    }
    public class Links {
        public String self;
    }
}
```

```java
public class QuestionText {
    public String type;
    public String id;
    public Attributes attributes;
    public Links links;

    public class Attributes {
        public Content content;
        public String name;
        public String elementtype;

        public class Content{
            public String headline;
            public String text;
        }
    }
    public class Links {
        public String self;
    }
}
```

Abbildung 5: ModelView der App

```java
public class Question {

    public String type;
    public Object qo;
    public long collected_at;
    public List<String> result;
    public View QFrame;

    public Question()
    {
        result = new ArrayList<>();
    }

    public void setResult (String s){
        switch (type){
            case "SingleChoice":
            case "YesNoSwitch":
                QuestionSingleChoice qsc = (QuestionSingleChoice) qo;
                int id = qsc.attributes.content.answers.indexOf(s);
                String resultstring = qsc.attributes.content.values.get(id);
                result.add(resultstring);
                break;
            case "SAMScaleFace":
            case "SAMScaleBody":
                result.clear();
                result.add(s);
                break;
            case "MultipleChoice":
                QuestionMultipleChoice mc = (QuestionMultipleChoice) qo;
                int id1 = mc.attributes.content.answers.indexOf(s);
                String resultstring1 = mc.attributes.content.values.get(id1);
                result.add(resultstring1);
                break;
            case "Slider":
                result.clear();
                result.add(s);
                break;
            case "TextDate":
            case "TextArea":
            case"TextString":
                result.clear();
                result.add(s);
                break;
        }
        collected_at = System.currentTimeMillis()/1000;
    }

    public void deleteResult(String s){
        switch (type){
            case "SingleChoice":
            case "YesNoSwitch":
                QuestionSingleChoice qsc = (QuestionSingleChoice) qo;
                int id = qsc.attributes.content.answers.indexOf(s);
                String resultstring = qsc.attributes.content.values.get(id);
                result.remove(resultstring);
                break;
            case "MultipleChoice":
                QuestionMultipleChoice mc = (QuestionMultipleChoice) qo;
                int id1 = mc.attributes.content.answers.indexOf(s);
                String resultstring1 = mc.attributes.content.values.get(id1);
                result.remove(resultstring1);
                break;

        }
    }
}
```

Abbildung 6: Questionmodel

Das Datenmodell der App (*ModelView*) bzw. die mittels *GSON* generierbaren Klassen-Objekte werden ergänzt durch drei weitere Klassen: *StudyEx*, *Question* (Abbildung 6) und *QuestionnaireEx*. Diese werden aus implementierungstechnischen Gründen benötigt. So wurde es möglich, einem Fragebogen (*Questionnaire*) eine Liste an Fragen zuzuordnen bzw. einer *Study* zusätzlich einen Status zuzuweisen. Die Klasse *Question* dient dazu, die Antworten der verschiedenen Fragen (unterschieden über den Typ (*type*) der Frage) zu setzen oder auch wieder zu löschen. Ein kleiner Einblick in spezielle Implementierungsdetails sind in Kapitel 5 zu finden, u.a. speziell für die verschiedenen Studien.

In Abbildung 7 wird grob ersichtlich, wie das Datenmodell des *ViewModel*, welches von der App benötigt wird, mittels *GSON* generiert wird. Auf gleichem Wege zurück, können ebenso wieder *JSON* Objekte erstellt werden. Meist werden in diesem Projekt jedoch die an die *API* gesendeten *JSON* „Pakete“ individuell bzw. manuell im *JsonParser* erstellt und nicht automatisiert über *GSON* generiert. Generell ist beides möglich. *GSON* ist eine Java Klasse, die die Generierung von (Java-)

Objekten aus *JSON* Objekten und zurück ermöglicht, sofern die Struktur der „Models“ genau mit derer in der *API/Datenbank* übereinstimmt.

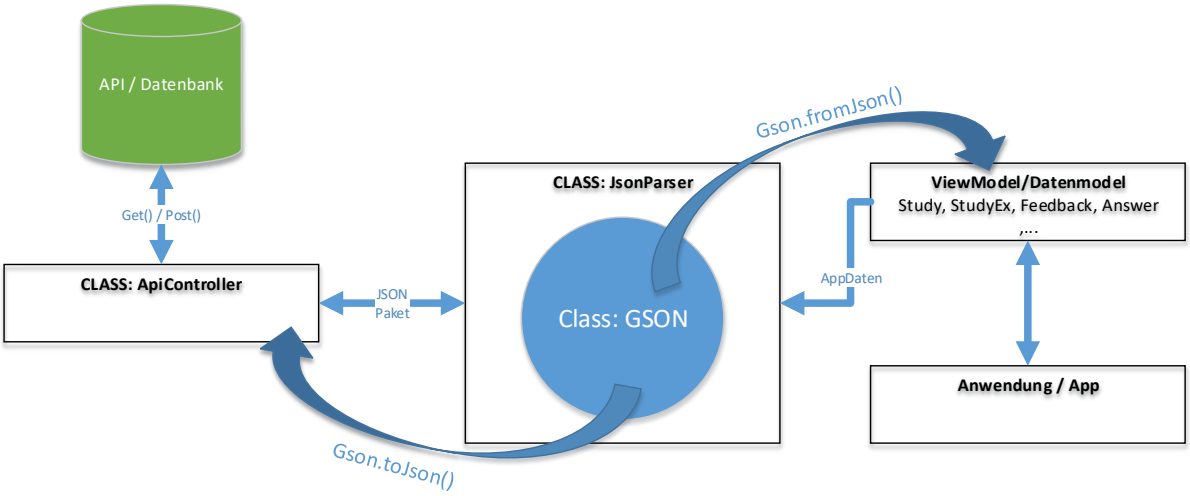

Abbildung 7: Datenmodel/ViewModel generieren mittels GSON

Die Daten zu Studien, Fragebögen, etc. liegen in der Datenbank als *JSON* Objekte vor und werden über die *API* bereitgestellt. Über die jeweilige Funktion (*GET* oder *POST*) werden Daten vom *ApiController* angefordert oder gesendet. Diese Daten in dieser Kommunikation werden stets als *JSON* Objekte bzw. als *Strings* übermittelt. Der *ApiController* ruft jeweils die Klasse *JsonParser* auf. Dort wird aus den *JSON* Objekten, wie sie über die *API* bereitgestellt werden, das in der App benötigte Datenmodell kreiert. Die *JSON* Objekte werden dabei mittels *GSON* automatisch zu den, in der App, benötigten Daten-Objekten geparst. In diesem Projekt werden die benötigten Datenmodelle stets über *GSON* generiert. Lediglich die *POST*, also die zu sendenden Antworten an die *API* wurden größtenteils manuell implementiert, was implementierungstechnische Gründe hatte.

3.4. Controller- Manager- und Handlerklassen

Komplettiert wird das *ViewModel* der App unter anderem durch die bereits genannte Klasse *ApiController*. Diese Klasse stellt, wie auch in Abbildung 3 zu sehen, sozusagen das Bindeglied zwischen Datenbank/*API* – mit internem, an die Anwendung angepasstem Datenmodel (z.B. den Klassen *StudyEx* oder *Question*) – und der Präsentationsschicht dar. Im *Offline Modus* stellt der *ApiController* außerdem die Bindung zwischen lokaler Datenbank und *Activities* dar. Die Klasse *ApiController* hält zudem verschiedene statische Variablen, welche für die spätere Ver- und Abarbeitung, sowie für die Funktionalität der Anwendung (wie zum Beispiel zur Visualisierung verschiedener Daten (z.B. Fragebögen in *static List<QuestionnaireEx> allQuestionnaireEx*)) notwendig sind. Weitere benötigte und implementierte Klassen für die definierten Anforderungen sind: *ApiConnectionTask*, *ApiJob*, *ApiRequest*, *ApiResult*, *BootCompleteReceiver*, *JsonParser*, *MessageController*, *MessageServer*, *SetMessageService* und *MyOfflineDB*.

Der *ApiConnectionTask* erbt von der Android-Klasse *AsyncTask*[7] und überschreibt dessen Funktionen *doInBackground* und *onPostExecute*. In dieser Klasse wird die eigentliche Verbindung zum Server bzw. zur *API* aufgebaut, Daten übertragen und die Verbindung wieder geschlossen. Die Daten werden dabei über *BufferedOutputStreams*, *BufferedReader* sowie *InputStreamReader*[8] verarbeitet.

[7] [15]
[8] [24] [25] [26]

Die *doInBackground*-Funktion bekommt immer einen *ApiJob* übergeben und arbeitet dessen Informationen ab. Es können hierbei mehrere *ApiConnectionTasks* bzw. *AsyncTasks* parallel ablaufen. In der *onPostExecute*-Methode wird dann die *ApiCallback*-Funktion aufgerufen, wobei wiederrum der *ApiJob* übergeben wird. Somit kann über den *ApiJob* später im *ApiController* die richtige *Callback*-Funktion aufgerufen werden, auch wenn mehrere *AsyncTasks* parallel laufen.

Der *ApiJob* kann dabei als eine Art Container aufgefasst werden. Er führt stets eine *ApiRequest* und ein *ApiResult* mit sich. So werden immer alle Infos und Daten zu einem Aufruf an die *API* zusammengehalten.

Die *ApiRequest* sowie das *ApiResult* halten die entsprechenden Information für eine korrekte Anfrage an die *API* bzw. um zurückgegebene Daten der *API* im Folgenden verarbeiten und nutzen zu können. Die Struktur dieser zwei Klassen ist in Abbildung 8 zu sehen.

```java
public class ApiResult {
    public String resultString;
    public Exception ex = null;
    public int statusCode = 0;
    public String locale;

    public String getLocale() {
        return locale;
    }

    public void setLocale(String locale)
{
        this.locale = locale;
    }

    public String getResultString() {
        return resultString;
    }

    public void setResultString(String
resultString) {
        this.resultString = resultString;
    }

    public Exception getEx() {
        return ex;
    }

    public void setEx(Exception ex) {
        this.ex = ex;
    }

    public int getStatusCode() {
        return statusCode;
    }

    public void setStatusCode(int
statusCode) {
        this.statusCode = statusCode;
    }
}
```

```java
public class ApiRequest {
    public String urltitle ;
    public String httpMethod;
    public String jsonData ;
    public String doneMethod ;
    public String id;
    public List<String> Headers;

    public void addHeader (String k, String v)
    {
        if (Headers == null) Headers = new ArrayList<>();
        Headers.add(k);
        Headers.add(v);
    }

    public void setUrltitle(String urltitle) {
        this.urltitle = urltitle;
    }
    public void setHttpMethod(String httpMethod) {
        this.httpMethod = httpMethod;
    }
    public void setJsonData(String jsonData) {
        this.jsonData = jsonData;
    }

    public String getDoneMethod() {
        return doneMethod;
    }

    public void setDoneMethod(String doneMethod) {
        this.doneMethod = doneMethod;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}
```

Abbildung 8: Request und Result

Der *JsonParser* dient dazu, die Daten und Dateien, die von der *API* geladen, oder umgekehrt an die *API* gesendet werden, entsprechend zu parsen, d.h. die *JSON*-Objekte, wie sie in der *API* vorliegen, so zu verarbeiten, dass diese von der App verwendet und abgearbeitet werden können. Auf der anderen Seite wird der *JsonParser* natürlich auch dazu verwendet, von der App verarbeitete, geänderte oder neue Daten dementsprechend zu parsen, damit diese dann auch von der *API* wiederverwendet und in der Datenbank abgelegt werden können. Die „Parser-Funktionen“ können dabei, wie bereits erwähnt, mittels der Hilfsklasse *GSON* oder auch manuell implementiert werden (grober Ablauf in Abbildung 7 zu sehen).

Der *BootCompleteReceiver*, der *MessageController*, der *MessageServer* und der *SetMessageService* sind für die Benachrichtigungen der App an den User von Nöten. Die Benachrichtigungen werden hierbei jeweils lokal über das Android Gerät erstellt, um die Funktionalität der Benachrichtigungen nicht abhängig von einer funktionierenden Internetverbindung zu machen. Benutzt wird dafür in

Android ein sogenannter *AlarmManager*[9]. Dieser sendet zu den bestimmten Zeitpunkten einen Broadcast, welcher von einem Receiver (hier der *MessageServer*, welcher von *BroadcastReceiver* erbt) empfangen wid. Dort wird dann die eigentliche Benachrichtigung für den User erstellt und auf dem Gerät visualisiert. Allerdings müssen die gespeicherten Benachrichtigungen, nach einem Neustart des Gerätes, erneut beim *AlarmManager* registriert werden. Dieses Problem wird durch einen Receiver (hier *BootCompleteReceiver*, erbt ebenfalls von *BroadcastReceiver*) gelöst, welcher beim Neustart des Android Gerätes aktiviert wird. Dieser startet in seiner *onReceive*-Funktion die *SetMessageService*-Klasse, welche wiederum den *MessageController* aufruft, um dort erneut die Zeitpunkte für die Benachrichtigungen im *AlarmManager* zu registrieren.

MyOfflineDB dient lediglich zur Verwaltung der lokalen Datenbank. Weitere Infos in im folgenden Kapitel (3.5).

3.5. Lokale Datenbank

Die lokale Datenbank umfasst die folgenden Tabellen und wurde so simpel wie möglich gehalten: *Questionnaires* (Fragebögen), *QuestionnaireStructure* (Fragen zu den Fragebögen) sowie *Answers* (die zugehörigen Antworten) und *MessageTimes* (Zeitpunkte für die Benachrichtigungen). Verwaltet wird die lokale Datenbank in der Klasse *MyOfflineDB*. Diese Klasse erbt von der Superklasse *SQLiteOpenHelper*[10]. Dies ist eine Hilfsklasse, die das Kreieren sowie das Organisieren einer Datenbank unterstützt. In *MyOfflineDB* werden dann die Funktionen *onCreate*() und *onUpgrade*() implementiert. *MyOfflineDB* sorgt nun dafür, dass eine bestehende Datenbank geöffnet, wenn nötig erstellt und aktualisiert wird. Immer die zuletzt online geladenen Daten werden in der lokalen Datenbank gespeichert. Abbildung 9 verdeutlicht dieses Schema.

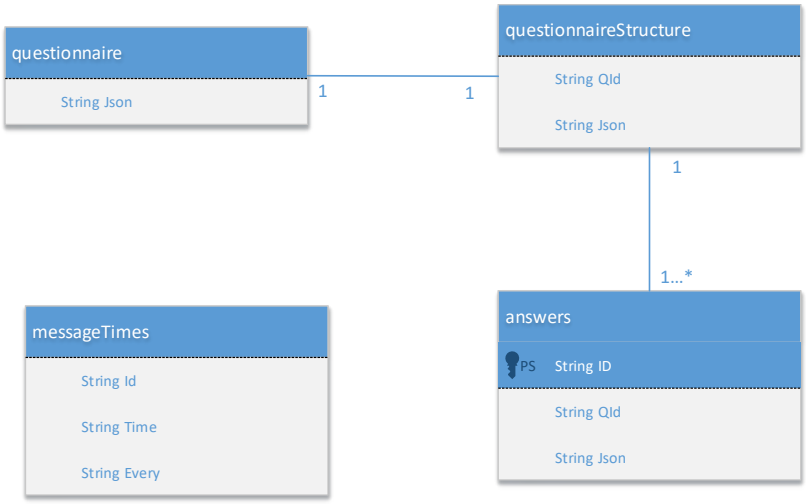

Abbildung 9: Struktur lokale Datenbank

Generell bedeutet dies, dass je ein Fragebogen jeweils eine Struktur bzw. ihm zugehörige Fragen hat. Weiter hält eine Fragebogenstruktur auch jeweils eine Referenz zu evtl. ihm zugehörigen Antworten eines Users. Im *Offline Modus* ist es in dieser App lediglich möglich, die zuletzt im *Online Modus* geladenen Fragebögen zu bearbeiten. Da lokal bisher darauf verzichtet wird zwischen verschiedenen Usern zu unterscheiden, ist die App im *Offline Modus* nur vom zuletzt eingeloggten User nutzbar. Sobald sich der User wieder online anmeldet, werden die evtl. ausgefüllten Antworten der *Answers* Tabelle an die *API* gesendet und die komplette, lokale Datenbank gelöscht. Weiter werden die neuen Daten von der *API* geladen und wieder auf der lokalen Datenbank abspeichert, um später einen

[9] [11]
[10] [12]

erneuten *Offline Modus*, mit aktuellen Daten, zu ermöglichen. Die zugehörigen Daten zu den Fragebögen, Fragen und Antworten werden jeweils in einem kompletten String gehalten, als würden diese an die *API* gesendet werden. Allerdings wird im *Offline Modus* keine *APIConnectionTask* mit *AsyncTask* gestartet, sondern der Callback erfolgt direkt aus dem *APIController* mit entsprechender *doneMethode,* wie in Abbildung 10 zu sehen. Die Referenz, welcher Fragebogen zu welcher *QuestionnaireStructure* gehört bzw. welche Antworten zu welchem Fragebogen, erfolgt über die mitgespeicherte *Questionnaire ID* (*QId*).

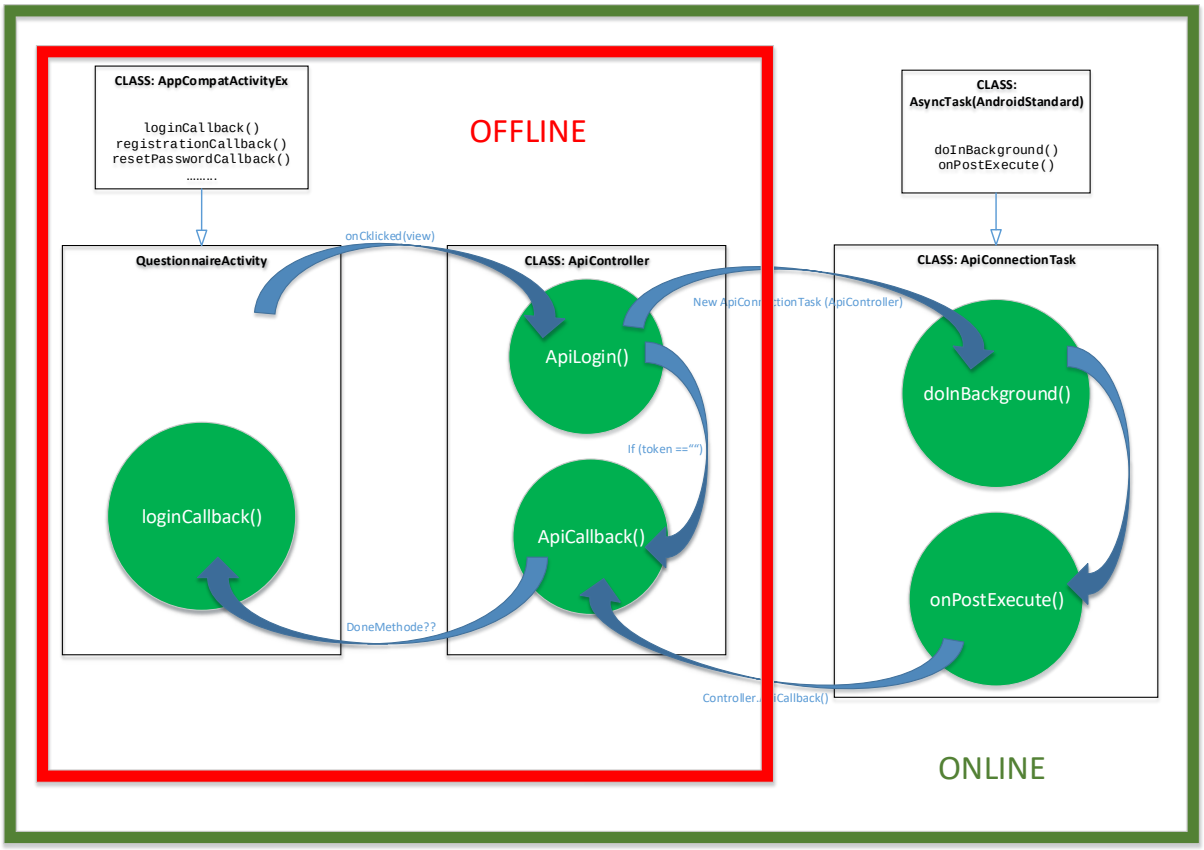

Abbildung 10: Unterschied Kommunikation Offline vs. Online

3.6. Views

Für die Präsentationsschicht der App wurden verschiedene Views, in Android auch *Activities* genannt, entwickelt und implementiert. Dabei ergeben sich mehrere *Vererbungen*. Jede *Activity*-Klasse erbt dabei im Endeffekt von der Super-Klasse *AppCompatActivity*[11]. Nötig ist dies, damit verschiedene Standard Funktionen in Android direkt für jede *Activity* verfügbar sind. Allerdings wurde zwischen die Super-Klasse und den *Activities* eine Art „Hilfsklasse“ *AppCompatActivityEx* gestellt, wie in Abbildung 11 zu sehen ist. Diese ermöglicht es, Objekte, Funktionen und Variablen, welche von allen *Activities* benötigt werden, direkt dort zu deklarieren. Als Beispiel wären hier die *DialogBuilder*[12] und die

[11] [13]
[12] [27]

Dialogfunktionen zu nennen. Weiter werden dort auch die *ActivityCallbacks* deklariert, die dann nur noch in der entsprechenden *Activity* überschrieben werden.

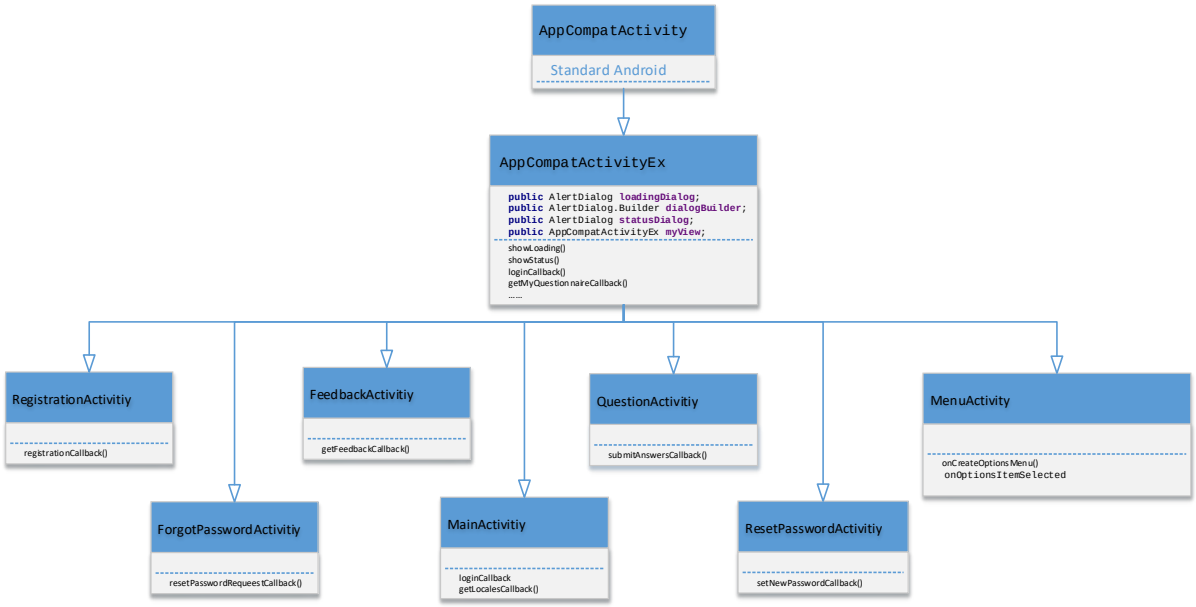

Abbildung 11: Views Vererbung AppCompatActivityEx

Die *MenuActivity* stellt dabei nochmals eine spezielle *Activity*-Klasse dar. Von ihr erben alle weiteren *Activitys*, die ein Menü benötigen, um durch die App zu navigieren. Welche *Activities* dies sind, ist in Abbildung 12 zu sehen.

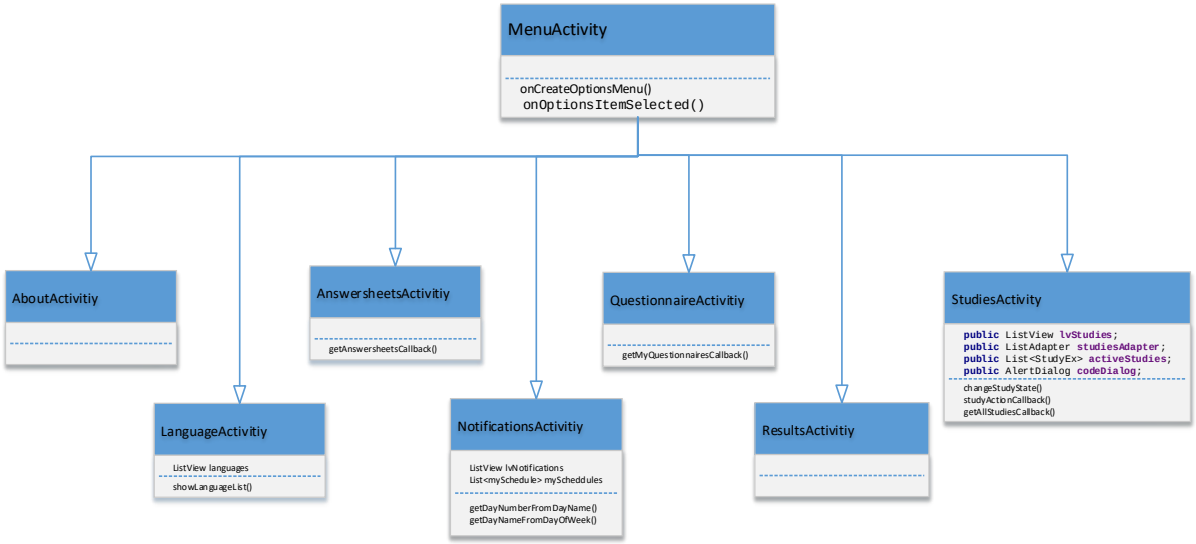

Abbildung 12: Views Vererbung MenuActivity

Zusätzlich zu diesen *Activity*-Klassen wurden noch eine *QuestionView*- und eine *QuestionnaireExView*-Klasse implementiert. Benötigt wird die *QuestionnaireExView*-Klasse, um die jeweiligen, einzelnen Fragebögen für die *QuestionnaireActivity* zu erstellen und diese zusätzlich mit einem *OnClickListener* zu versehen, wodurch man zu den zugehörigen Fragen navigieren kann. Die *QuestionView*-Klasse dient dazu, die einzelnen, verschiedenen Fragen, bezüglich ihres Typs zu unterscheiden, diese jeweils zu realisieren und gleichzeitig, anhand des Fragetyps eine entsprechend

geforderte, richtige Eingabemöglichkeit zu kreieren. Für die verschiedenen, nötigen Eingabeformate wurden Klassen, wie in Abbildung 13 zu sehen, implementiert.

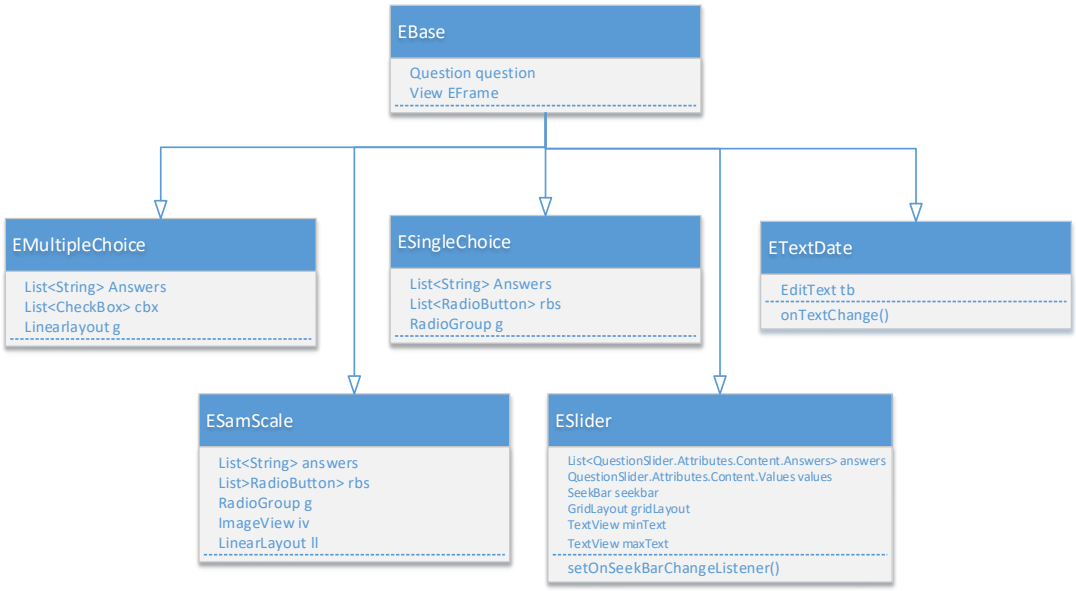

Abbildung 13: Views Vererbung EBase

Die *EBase*-Klasse ist eine abstrakte Klasse und dient lediglich dazu, die Grundstruktur der Unterklassen zu setzen, sowie den Pointer auf den entsprechenden View zu erhalten. Die einzelnen Unterklassen erhalten die benötigten Strukturen, Variablen und Objekte. So wird in *ESingleChoice* z.B eine *RadioButton-Group* implementiert, während in *EMultipleChoice Checkboxes*, im *ESlider* der *Slider,* in *ESamScale* ein *ImageView* und in *ETextDate* u.a. ein editierbares *Textfeld* implementiert wird.

3.6.1. ListAdapter

Weiter werden mehrere *ListAdapter* benötigt. Deshalb wurden im *View*-Paket zusätzlich ein *StudyListAdapter*, ein *LanguageListAdapter* sowie ein *NotificationListAdapter* implementiert. Wie die Namen der Klassen schon andeuten, sind diese vorhanden, um dabei zu helfen die entsprechenden Listen zu organisieren und zu visualisieren. Allesamt erben jeweils von der Android-Klasse *ArrayAdapter*[13]. Der StudyListAdapter unterstützt den Aufbau und die Funktionalität der Studienliste in der StudiesActivity (Implementierungsdetails in Kapitel 5.1), der LanguageListAdapter die Liste an Sprachen und der NotificationListAdapter die Benachrichtigungen der App an den User.

[13] [14]

3.7. Zusammenfassung der Struktur

Im Allgemeinen lässt sich am Ende folgendes, in Abbildung 14 ersichtliches Schema des *Model-ModelView + ApiController-View* Aufbaus darstellen. Der *ApiController* ist für die Kommunikation mit der *Model*-Seite zuständig, sprich mit *API/Datenbank* oder im *Offline Modus* auch mit der lokalen Datenbank. Die eingehenden bzw. ausgehenden Daten werden vom *Parser* gewandelt und im ModelView in entsprechenden Klassen gehalten, so dass man diese in den *Activities* verwenden und visualisieren kann. Analog werden Aktivitäten in den Views verarbeitet und evtl. geänderte oder neue Daten gespeichert, geparst und wieder an die Model-Seite gesendet.

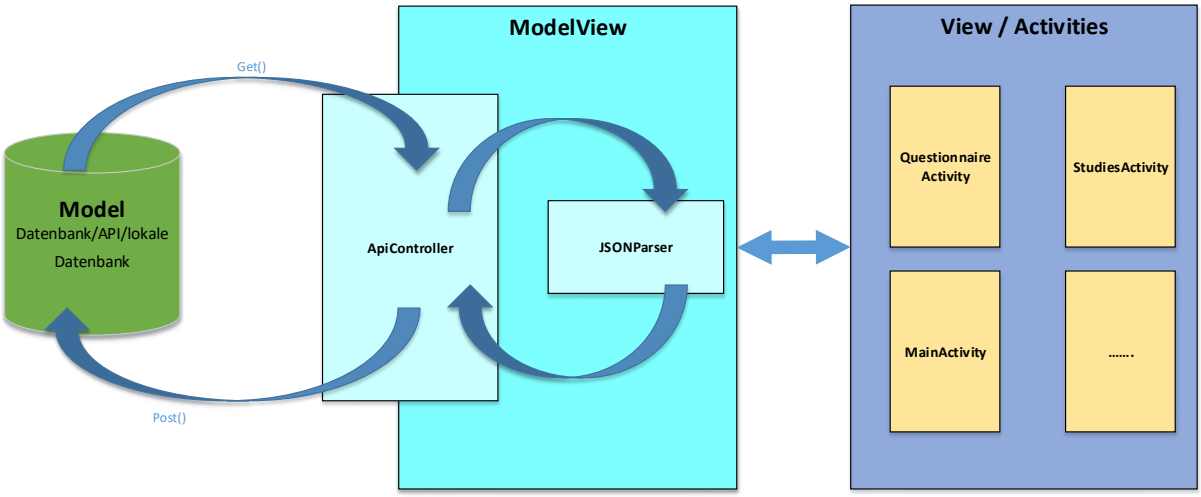

Abbildung 14: Model-ModelView + Controller-View

4. Vorstellung des Track Your Stress Rahmenwerks

Dieses Kapitel stellt das *Track Your Stress* Rahmenwerk aus der Sicht eines Users dar. Dabei werden die einzelnen Funktionen der Android App aufgezeigt.

4.1. Vorstellung der App

Ziel der App ist die individuelle Stresswahrnehmung eines Users zu erfassen. Daher sind die Fragebögen zur Stresswahrnehmung zentraler Funktionsbestandteil der Android App. Unterstützt werden jegliche Android Geräte ab Android Version 4.0.3.

4.1.1. Anmeldung mit Username und Passwort

Um die App nutzen zu können, muss ein User einen gültigen Account besitzen. Die erste Ansicht der App nach dem Öffnen ist daher die Login Ansicht, wie sie in Abbildung 15 zu sehen ist. Der User kann sich nun entweder direkt anmelden, sofern dieser schon einen gültigen Account besitzt. Ist das nicht der Fall, kann er sich über den *Registrieren Button* direkt ein Konto auf dem Server erstellen. Spätestens nach einer erfolgreichen Registrierung kann sich der User mit Name und Passwort, über Klick auf den *Einloggen Button* am Server anmelden. Sollte dies alles auf Grund einer fehlenden Internetverbindung nicht möglich sein oder sollte ein anderer Fehler (wie etwa fehlerhafte Angaben bei Username und/oder Passwort) auftreten, wird eine entsprechende Meldung ausgegeben.

Abbildung 15: Login Ansicht

Nach einer erfolgreichen Anmeldung werden direkt alle Fragebögen zur Studie *Track Your Stress* bzw. die Fragebögen aller beigetretener Studien geladen. Diese Beschreibung beschränkt sich im Weiteren lediglich auf die Studie *Track Your Stress*. Ist es die erstmalige Anmeldung oder hat der User bisher den demographischen Fragebogen zu seiner Person noch nicht beantwortet, werden ihm die kontinuierlich wiederkehrenden Fragebögen zur Stresswahrnehmung noch nicht angezeigt, woraus folgt, dass er diese auch noch nicht bearbeiten kann. Dies wird erst möglich, wenn der demographische Fragebogen erfolgreich beantwortet wurde.

4.1.2. Registrierung

Für die Registrierung bei *Track Your Stress* muss ein Benutzer lediglich einen Benutzernamen, seine E-Mail-Adresse und ein Passwort angeben. Visualisiert wird dies wie in Abbildung 16 zu sehen ist. Um eine falsche Eingabe zu verhindern, muss das Passwort zwei Mal eingegeben werden. Beim Klick auf

den *Button Registrieren* wird überprüft, ob alle Daten eingegeben wurden. Wenn ja, werden diese an die *API*/den Server übertragen. Im Erfolgsfall wird eine Meldung angezeigt, dass eine Bestätigung per E-Mail an den Benutzer gesendet wurde. Bei fehlenden Daten, wenn der Benutzername oder die E-Mail-Adresse bereits registriert sind, bei fehlender Internetverbindung des Smartphones, sowie bei sonstigem Scheitern der Registrierung, wird eine entsprechende Fehlermeldung ausgegeben. Über *Abbrechen* gelangt der Nutzer zurück zur *Login* Ansicht.

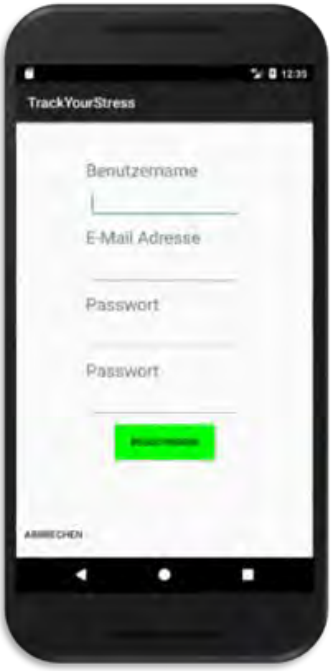

Abbildung 16: Registrierungsansicht

4.1.3. Passwort vergessen / Passwort zurücksetzen

In Abbildung 17 wird der Ablauf aufgezeigt, sollte ein User sein Passwort vergessen haben und in Folge dessen zurücksetzen müssen. In der Login Ansicht betätigt dieser den *Passwort-vergessen-Button* und erreicht dadurch die *ForgotPasswordActivity*. Dort kann man eine E-Mail-Adresse angeben, an welche beim Klick auf den *Send-Reset-Request-Button*, ein *Token* gesendet wird. Mit diesem *Token* kann der User dann auf der *ResetPasswordActivtiy* ein neues Password wählen. Abgeschlossen wird der *Password Reset* durch den *Button Set-New-Password*. Dadurch wird das neue Passwort an den Server gesendet und dort überschrieben. Diese *Passwort Reset* Ansicht (*ResetPasswordActivtiy*) erreicht man, indem man auf der *ForgotPasswordActivity* den *Set-New-Password-Button* klickt. Zur *Login* Ansicht gelangt man über die jeweiligen *Back Buttons*.

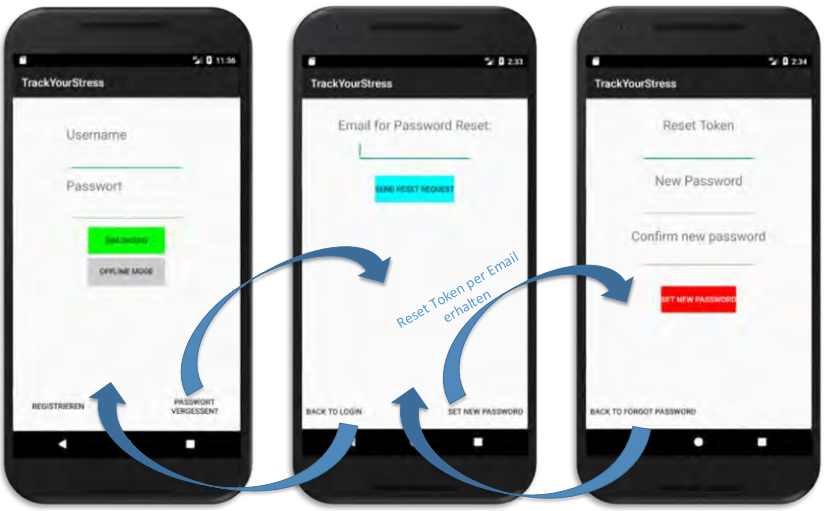

Abbildung 17: Passwort vergessen

4.1.4. Ansicht aller Fragebögen / Die Fragebögen-Liste

Nach erfolgreichem Login gelangt der User direkt zur Oberfläche der Fragebögen-Liste, der *QuestionnaireActivity*. Dort werden die Fragebögen zu den Studien aufgelistet, welchen der User bereits beigetreten ist. Den Usern der *Track Your Stress* Studie bieten sich dabei zwei Möglichkeiten, wie in Abbildung 18 zu sehen. Loggt sich der User zum ersten Mal ein und/oder hat er den einmaligen demographischen Fragebogen zu seiner Person noch nicht ausgefüllt, dann wird ihm momentan auch nur dieser eine Fragebogen zur Bearbeitung angezeigt. Wurde der demographische Fragebogen bereits erfolgreich bearbeitet, so verschwindet dieser in der Liste der Fragebögen (dieser ist nur einmal pro User zugelassen und wird auch nur einmal pro User benötigt) und die kontinuierlich wiederkehrenden Fragebögen, zur täglichen/wöchentlichen/monatlichen Überwachung der Stresswahrnehmung, erscheinen. Nun können diese vom User bearbeitet werden.

Abbildung 18: Fragebögen-Liste

Dass in Abbildung 18 auch Tinnitus-Fragebögen angezeigt sind, liegt daran, dass im Moment jeder neue User automatisch auch der *Track Your Tinnitus* Studie beigetreten ist.

Durch einen Klick auf den entsprechenden Fragebogen wird dieser geöffnet und die Bearbeitung des ausgewählten Fragebogens kann begonnen werden.

4.1.5. Ausfüllen von (statistischen) Fragebögen

Die (statistischen) Fragebögen können in der App ausgefüllt werden. Um die kontinuierlich wiederkehrenden Fragebögen zur Stresswahrnehmung in der App benutzen zu können, müssen zuerst (alle) demographischen Fragebögen ausgefüllt werden. Folgende Fragebögen sind zum Zeitpunkt dieser Arbeit eingepflegt:

- TrackYourStress – Demographischer Fragebogen
- TrackYourStress – Täglicher Fragebogen
- TrackYourStress – Wöchentlicher Fragebogen
- TrackYourStress – Monatlicher Fragebogen

Ein (statistischer) Fragebogen besteht immer aus einem Titel (1), einer optionalen Beschreibung (2) und Fragen (3 (gelb)) mit entsprechenden Antwortmöglichkeiten. Abbildung 19 zeigt verschiedene Fragebögen, die verschiedene Antwortmöglichkeiten bieten. Ein Benutzer hat beim bzw. nach dem Ausfüllen des Fragebogens 2 bzw. 3 verschiedene Möglichkeiten:

- Wenn alle Pflichtfragen beantwortet sind, kann er die Antworten, mittels dem *Senden-Button,* absenden. Dadurch werden im *Online-Modus* seine gegebenen Antworten direkt an die *API* gesendet. Dort werden diese dann weiterverarbeitet und in der Datenbank gespeichert.
- Befindet sich der User auf Grund einer fehlenden Internetverbindung im *Offline-Modus*, dann werden die Antworten beim Klick auf den *Senden-Button* in der lokalen Datenbank gespeichert. Beim nächsten *Online-Login* wird auf jener lokalen Datenbank geprüft, ob noch Antworten vorliegen. Sollte dies der Fall sein, werden diese direkt mit dem *Login* an die *API* gesendet.
- Betätigt der User den *Back-Button* gelangt er zurück zu seiner Fragebögen-Liste. Allerdings werden die zuvor abgegebenen Antworten, welche noch nicht gesendet/gespeichert wurden, gelöscht.

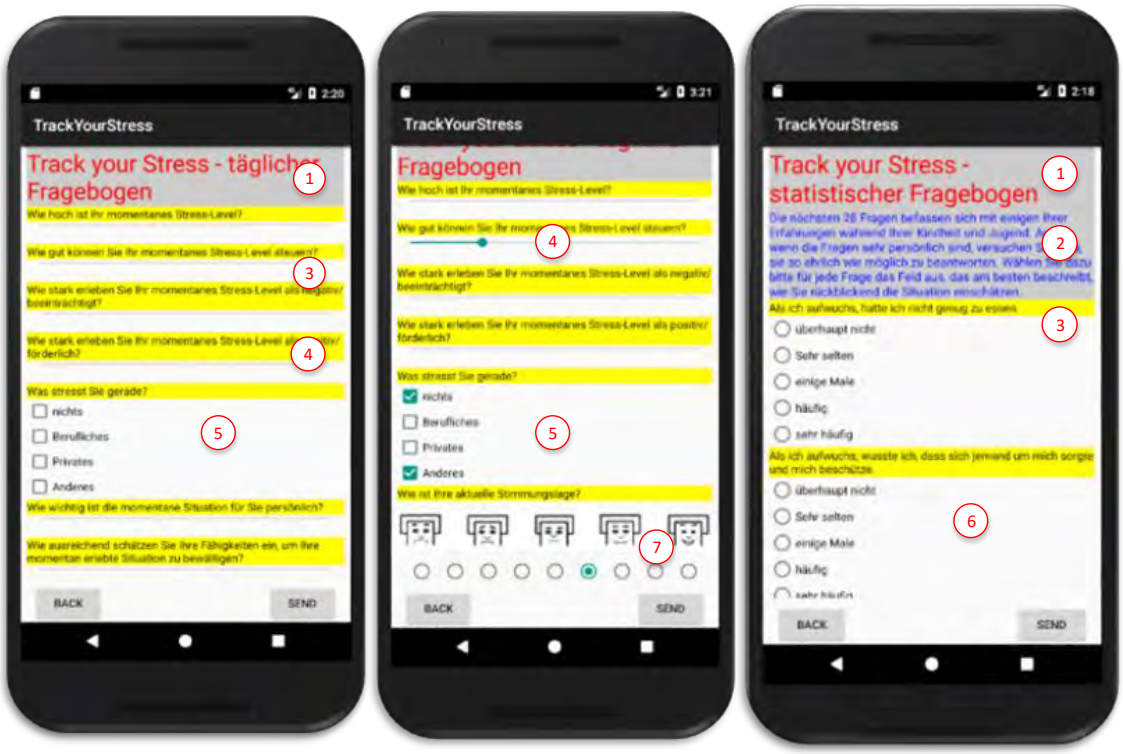

Abbildung 19 : Fragebögen

4.1.6. Antwortmöglichkeiten eines (statistischen) Fragebogens

Ein (statistischer) Fragebogen kann unterschiedliche Antwortmöglichkeiten haben. *Track Your Stress* bietet generell sieben verschiedene Antwortmöglichkeiten:

- Text: Als Antwort kann ein Text in eine Zeile eingegeben werden. Die Eingabe wird dabei nicht eingeschränkt oder validiert. Bei Pflichtfragen wird beim Speichern bzw. Absenden lediglich überprüft, ob ein Text eingegeben wurde.
- Text (mehrzeilig): Unterscheidet sich von dem Antworttyp *Text* nur in der evtl. Darstellung für den User in der App, sofern dieser mehrere Zeilen Text eingibt. Es kann dann ein größeres Textfeld angezeigt werden bzw. Umbrüche und die Eingabe von mehreren Zeilen Text sind erlaubt.
- Radio Buttons: Es gibt mehrere vorgegebenen Antwortmöglichkeiten, von denen allerdings nur eine ausgewählt werden kann. Bei Pflichtfragen wird beim Speichern bzw. Absenden überprüft, ob der User eine Antwort ausgewählt hat. (Abbildung 19 (6))
- Checkboxes: Unterscheidet sich vom Antworttyp *Radio Buttons* nur darin, dass hier auch Mehrfachantworten möglich sind. Bei Pflichtfragen muss mindestens eine Antwortmöglichkeit ausgewählt worden sein, bevor das Speichern/Absenden möglich wird. (Abbildung 19 (5))
- Slider: Ein spezielles Element, um Werte auf einer bestimmten Skala anzugeben, wie z.B. von 0% bis 100% oder auch von 1 bis 10. Allerdings wird die Skala meist anhand von Beschreibungen des besten, bzw. schlechtesten Falls definiert. Wichtig war, dass in der App kein initialer Wert angegeben wird, um den User nicht zu beeinflussen. Das Sliderelement erscheint erst, wenn der User das erste Mal die Skala berührt. Bei Pflichtfragen wird überprüft ob der User einen Wert auf der Skala angegeben hat. Erst dann ist das Speichern/Absenden möglich. (Abbildung 19 (4))

- SAMScaleFace/SAMScaleBody: Diese Antwortmöglichkeit ist eine Mischung aus den Antworttypen *RadioButton* und *Slider*. Es gibt mehrere Antwortmöglichkeiten bzw. Antwortwerte (*Values*). Ziel ist es, wie beim *Slider*, einen Wert auf einer bestimmten Skala anzugeben, allerdings wird die Skala hierbei mit je einem *RadioButton* pro *Value* erzeugt. Die „Werte“ der *RadioButton* sollen dabei durch bestimmte Mannequin (*SAMScaleBody*) oder Smilie-ähnlichen (*SAMScale*)*Faces* assoziiert werden. (Abbildung 19 (7) Bsp.: *SAMScaleFaces*). Bei Pflichtfragen wird beim Speichern bzw. vor dem Senden geprüft, ob der User einen Wert gewählt hat.
- TextDate: Ist generell wie *Text* ein Textfeld. Allerdings kann als Antwort lediglich ein Datum angegeben werden, wie es z.B. bei der Angabe eines Geburtsdatums nötig ist. Diese Antwortmöglichkeit wird immer durch ein Textfeld repräsentiert, wobei wie in Abbildung 20, nur ein entsprechendes Eingabeformat vorhanden ist. Bei Pflichtfragen wird vor dem Speichern/Absenden überprüft, ob ein Wert angegeben wurde.

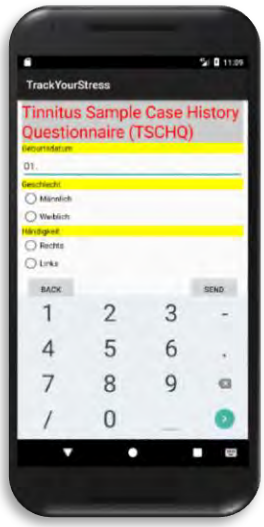

Abbildung 20: TextDate

Weiter wird beim Beantworten des Fragebogens, im Hintergrund, eine gewisse Anzahl an Hintergrundgeräuschen aufgenommen, sofern der User den Zugriff auf das Mikrofon erlaubt hat. Die Anzahl der aufgenommenen Hintergrundgeräusche hängt dabei von den Integerwerten *amount* bzw. *each* in der *API* ab. Ist der *amount*-Wert gesetzt, wird in gewissen, festgelegten Abständen aufgezeichnet, bis die Anzahl der Aufzeichnungen gleich dem amount-Wert ist. Sollte dagegen der *each*-Wert gesetzt sein, dann wird *each* mit einer Sekunde verrechnet und dementsprechend oft aufgezeichnet, d.h. ist *amount* = 5 wird versucht fünfmal aufzuzeichnen, während der Fragebogen bearbeitet wird. Für *each* bedeutet dies bei einem Wert = 5, dass alle 5 Sekunden eine Aufzeichnung der Lautstärke erfolgt. Die Lautstärke wird dabei generell in Dezibel verrechnet und auch in diesem Format an die *API* gesendet. Um ein zu schnelles Beenden der Bearbeitung und dadurch fehlende Lautstärkeaufnahmen abzufangen, erfolgt immer eine Aufzeichnung mit dem Betätigen des *Senden-Buttons*.

Generell wird der Titel eines Fragebogens ganz oben, in roter Schrift auf grauem Hintergrund angezeigt. Die einzelnen Fragen sind gelb hinterlegt und die jeweiligen Antworten stehen auf neutralem weißen Hintergrund. Die Konfiguration, sowie die Erstellung der Fragebögen, d.h. welche Frage und welche möglichen Antwortmöglichkeiten bestehen, erfolgt auf Seiten des Servers. Nach dem Beantworten der Fragen kann der User mit einem Klick auf den *Senden-Button* die Antworten

des Fragebogens abschicken, womit diese auf dem Server, bzw. im *Offline-Modus* auf der lokalen Datenbank, gespeichert werden. Sollten beim Versuch des Speicherns bzw. Absenden der Antworten, noch Pflichtfragen nicht beantwortet sein, so wird dem User, wie in Abbildung 21 zu sehen, mitgeteilt, wie viele Pflichtfragen noch zu beantworten sind. Weiter werden dem User die noch offenen Pflichtfragen entsprechend rot visualisiert (Abbildung 21 (1)).

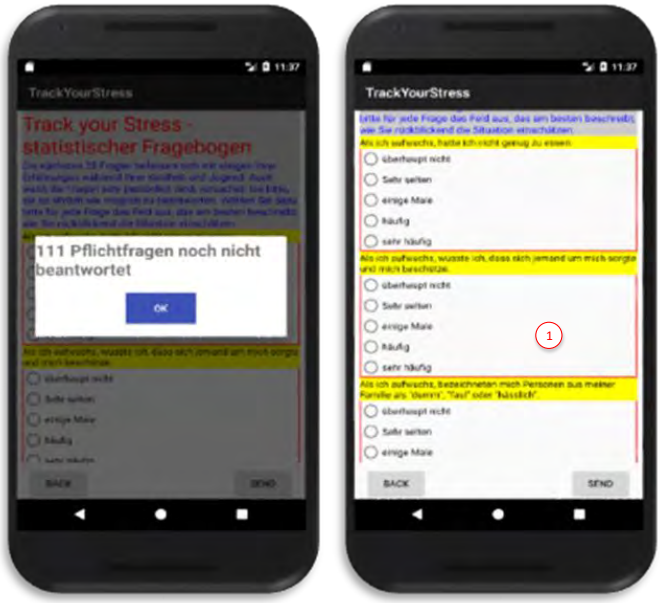

Abbildung 21: Pflichtfragen

Die Ansicht der Fragen erbt, wie bei den Vererbungs-Abbildungen gezeigt, nicht von der *MenuActivity*. Deswegen kann man von dort nur mittels dem *Back-Button* zurück zur Ansicht aller Fragebögen. Wird ein Fragebogen erfolgreich beantwortet und gespeichert bzw. abgesendet, gelangt der User automatisch zur Liste aller Fragebögen zurück. Wie von allen Views, die von der *MenuActivity* erben, kann man von dort über das *Menu-Fenster* rechts oben zu verschiedenen anderen Seiten der Apps navigieren (Abbildung 22(1)). So zum Beispiel auch zur Ansicht aller aktuell verfügbaren Studien (*StudiesActivity*).

4.1.7. Alle Studien

In der *StudiesActivity* werden dem User, wie in Abbildung 22(2), alle aktuell verfügbaren Studien angezeigt. Dem User wird anfangs jeweils nur der Titel der Studie sichtbar gemacht. Bereits beigetretene Studien sind dabei grün hinterlegt. Türkis gefärbte Studien bedeuten, dass der User eine Einladung zur Studie angefragt (*requested*, Zustand = *pending*) hat, ohne die er sonst nicht beitreten kann. Sollte ein User bereits zu einer Studie eingeladen, aber noch nicht beigetreten sein, dann wird diese Studie gelb eingefärbt. Die restlichen Studien werden neutral mit grau visualisiert. Wählt man nun eine Studie aus, indem man lediglich auf einen Titel klickt, dann wird eine kurze Beschreibung der ausgewählten Studie eingeblendet (Abbildung 22 (3)). Bei wiederholtem Klick wird diese Beschreibung wieder ausgeblendet. Dazu werden je nach „Beitritt-Status“ des Users, verschiedene Aktionen zur gewählten Studie angeboten, wie wiederum in Abbildung 22(3) zu sehen ist. Ist der User der gewählten Studie bereits beigetreten, dann kann er diese Studie natürlich wieder verlassen, indem der rote *Studie-verlassen-Button* geklickt wird. Wartet der User auf eine angefragte Einladung (türkis), dann ist es auch möglich, diese Anfrage, mit dem dunkelgrauen *Delete-Pending-Study-Button* wieder zu löschen, falls er doch nicht mehr beitreten möchte. Ist der User der gewählten Studie noch nicht beigetreten, dann kann er dies mit dem grünen *Studie-beitreten-Button* versuchen. Je nach Typ der Studie ist es nun möglich sofort beizutreten (offene Studie, Abbildung

22(6) oder bereits zur Studie eingeladen), eine Anfrage (*Request*) für eine Einladung zu senden (*Invite* Studie, Abbildung 22(5)) oder es wird sogar eine *Code-Eingabe* gefordert, um der Studie beitreten zu können (geschlossene Studie, Abbildung 22(4)). Je nach Aktion und Erfolg wird danach eine entsprechende Meldung angezeigt und die Visualisierung angepasst (Abbildung 22 (5 + 6)).

Abbildung 22: Alle Studien

4.1.8. Benachrichtigungen, Sprache ändern, Ergebnisse, Logout

Über das Menü Fenster rechts oben kann man, neben den Studien und eigenen Fragebögen, auch zu weiteren Funktionalitäten der App navigieren. Dazu zählen unter anderem die Benachrichtigungen, die Sprachauswahl der App und mögliche Feedbacks zu bereits beantworteten und ausgewerteten Fragebögen. Feedbacks sind für das *Track Your Stress Projekt* allerdings bisher noch nicht vorgesehen und daher, für die zugehörigen Fragebögen, auch noch nicht in die Datenbank eingepflegt. Da die App aber ermöglicht weiteren Studien beizutreten, in welchen evtl. bereits Feedbacks vorhanden sind, ist diese Funktionalität trotzdem bereits realisiert. In Abbildung 23 sind die entsprechenden Ansichten dargestellt.

Abbildung 23: Benachrichtigungen, Sprache, Ergebnisse

- Benachrichtigungen werden dem User, wie in Abbildung 23 (1), in der Mitteilungszentrale des Android Geräts mitgeteilt. In diesen Mitteilungen wird er, zu den entsprechenden Benachrichtigungszeiten, aufgefordert, einen neuen Fragebogen seiner Studie auszufüllen. Interagiert der User mit einer dieser Benachrichtigungen, dann wird er direkt auf die *Login* Seite der App geleitet. In der App selber kann man über den Menü-Unterpunkt *Benachrichtigungseinstellungen* auch die Benachrichtigungszeiten bzw. Zeiträume der Benachrichtigungen für seine Fragebögen einsehen (Abbildung 23 (2)).
- Die Sprache der App lässt sich im Unterpunkt *Sprache* (*Language*) verändern (Abbildung 23 (3)). Momentan stehen Englisch und Deutsch zu Verfügung. Die aktuell ausgewählte Sprache ist grün hinterlegt, die anderen sind neutral in Weiß gehalten. Durch Klick auf eine Sprache wird die App sofort auf diese Sprache umgestellt.
- Mögliche Ergebnisse bereits ausgewerteter Fragebögen kann der User im Menü-Unterpunkt *Ergebnisse* einsehen. Visualisiert wird dies wie in (Abbildung 23 (4)). Dort werden alle Ergebnisse, der beantworteten Fragebögen, untereinander aufgelistet. Beschriftet sind die einzelnen Ergebnisse mit der ID und dem genauen Zeitpunkt des Absendens der Antworten zu diesem Fragebogen. Interagiert man mit einem dieser Ergebnisse, dann wird die eigentliche Auswertung zum ausgewählten Ergebnis auf einer neuen Seite angezeigt, sofern eine Auswertung in der Datenbank, zu jenem Fragebogen, vorhanden ist. Dies ist, wie erwähnt, im *Track Your Stress* Projekt bisher allerdings noch nicht vorgesehen.
- Über den Menü-Punkt *Abmelden* (*Logout)* wird der User abgemeldet und gelangt wieder zur *Login* Seite der App. Ist der User zu lange inaktiv, läuft der sogenannte *Login-Token* nach einer gewissen Zeit ab. Ist dies der Fall, wird bei einer Aktion, zu welcher man eingeloggt sein muss, eine Fehlermeldung ausgegeben. Für volle Funktionalität muss sich der User zuerst wieder neu einloggen. Auch in diesem Fall gelangt der User über den *Abmelden*-Menü-Punkt zurück zur *Login*-Seite.

Da die *Views/Activities* Ergebnisse, Benachrichtigungen und Sprache, alle von *MenuActivity* erben, kann man von den 3 verschiedenen Views, jeweils über das Menü, auch jederzeit zu allen anderen Funktionalitäten navigieren.

4.1.9. Offline Modus

Der *Offline-Modus* ist eine Art Lite-Version der App. Ist ein *Login* auf Grund fehlender Internetverbindung nicht möglich, dann kann der User auf der *Login*-Seite über den Button *Offline Mode,* in diesen *Offline-Modus* wechseln. Die grobe Funktionalität wird in Abbildung 24 ersichtlich. Der Unterschied zum *Online-Modus* mit voller Funktionalität besteht darin, dass im Offline-*Modus* lediglich die Hauptfunktion der App, d.h. das Beantworten und Ausfüllen der Fragebögen möglich ist. Dazu müssen die Fragebögen allerdings mindestens einmal zuvor vom Server geladen worden sein, was normalerweise bereits beim ersten (online) *Login* in der App geschehen ist. Zu erwähnen ist auch, dass die Benutzung des *Offline-Modus* lediglich von einem User pro Gerät möglich ist, da die lokale Datenbank immer diejenigen Fragebögen hält, die bei der letzten Onlinenutzung geladen wurden, d.h. es sind lokal immer die Fragebögen des zuletzt (online) angemeldeten Users gespeichert. Da aber heute davon ausgegangen werden kann, dass jede Person auch ein eigenes Gerät besitzt, wird die Hauptfunktion (das Ausfüllen der Fragebögen) auch ohne Internetverbindung ermöglicht.

Nachdem sich der User nun, nach gescheitertem Login, über den *Offline-Button,* in den *Offline-Modus* begeben hat (Abbildung 24 (1 und 2)), werden, die dem User zugehörigen Fragebögen geladen. Dabei wird wieder überprüft, ob der demographische Fragebogen bereits erfolgreich an die *API/Datenbank* gesendet wurde oder nicht (Abbildung 24(3 und 4)). Wählt der User nun einen Fragebogen aus, kann er diesen wie gewohnt bearbeiten. Da der ausgefüllte Fragebogen ohne Internetverbindung nicht abgeschickt werden kann, wird dieser bzw. die Antworten, durch Interaktion mit dem *Send-Button,* auf der lokalen Datenbank gespeichert. Beim nächsten (online) *Login* werden alle offline abgegeben Antworten zu entsprechenden Fragebögen direkt an die *API/Datenbank* geschickt und die lokale Datenbank gelöscht.

Abbildung 24: Offline Modus

5. Implementierung

Dieses Kapitel befasst sich mit bestimmten Bereichen der Implementierung der *Track Your Stress* App, welche unter anderem für die Erfüllung der gestellten Anforderungen nötig sind. Dabei wird beleuchtet, wie erreicht wird, dass ein User, neben der *Stress*-Studie, auch anderen Studien in der App beitreten kann. Zu beachten galt dabei, dass die Zustände der verschiedenen Studien eingehalten werden müssen und sich die App entsprechend verhalten soll.

5.1. Alle Studien

Neben der Hauptfunktion, Fragebögen zur *Stress*-Studie zu beantworten, ist eine weitere Anforderung an die App, dass zusätzlich zur „Standard"-Studie *Stress* die Möglichkeit bestehen sollte, weiteren, anderen Studien beitreten zu können. Der Vorteil besteht darin, dass ein Benutzer der App *Track Your Stress*, auf ein und derselben App verschiedenen Studien parallel beigetreten sein kann. Dadurch wird es natürlich auch möglich, dass der Benutzer die jeweiligen Fragebögen zu den verschiedenen Studien, in einer App beantworten kann. Auch im Hinblick auf die Zukunft und die Weiterentwicklung der App ist dies von Vorteil, da man dadurch eventuelle Zusammenhänge verschiedener Studien oder Gemeinsamkeiten verschiedener Krankheitsbilder erkennen kann. Denkbar wäre dies womöglich gerade für *Track Your Tinnitus* und *Track Your Stress*, da oftmals gerade auch in Stress eine mögliche Ursache für eine Tinnituserkrankung vermutet wird.

Wichtig und eine gleichzeitige Schwierigkeit war, dass die verschiedenen Studien entweder geschlossen (privat) oder offen für jeden Benutzer der App sind. Dabei unterscheiden sich die privaten Studien nochmals untereinander. So kann für eine geschlossene Studie ein Passwort angefordert werden, welches man an gegebener Stelle erhält (z.B. von einem Arzt oder auch Systemadmin) oder es kann eine Studie vom Typ „invite" sein, wofür man für den Beitritt eingeladen werden muss. Offenen Studien kann jeder User jederzeit beitreten. Über den in der *Datenbank/API* gegebenen *AccessType* werden programmtechnisch vier verschiedene mögliche Zustände (in Abbildung 25 pink markiert) für die Studien kreiert. Ein User kann sich demnach, wie in Abbildung 25 zu sehen, pro Studie in je einem der vier Zustände befinden bzw. von einem Zustand, mittels entsprechenden Interaktionen, in den nächsten Zustand wechseln. In Bezug auf die verschiedenen möglichen Zustände der Studie war es nötig, wie in Kapitel 3.3.1 im Datenmodell der App aufgeführt, eine Klasse *StudyEx* mit einem *String State* zu erstellen, um den jeweiligen Status der Studie, in Bezug auf den User, festzuhalten.

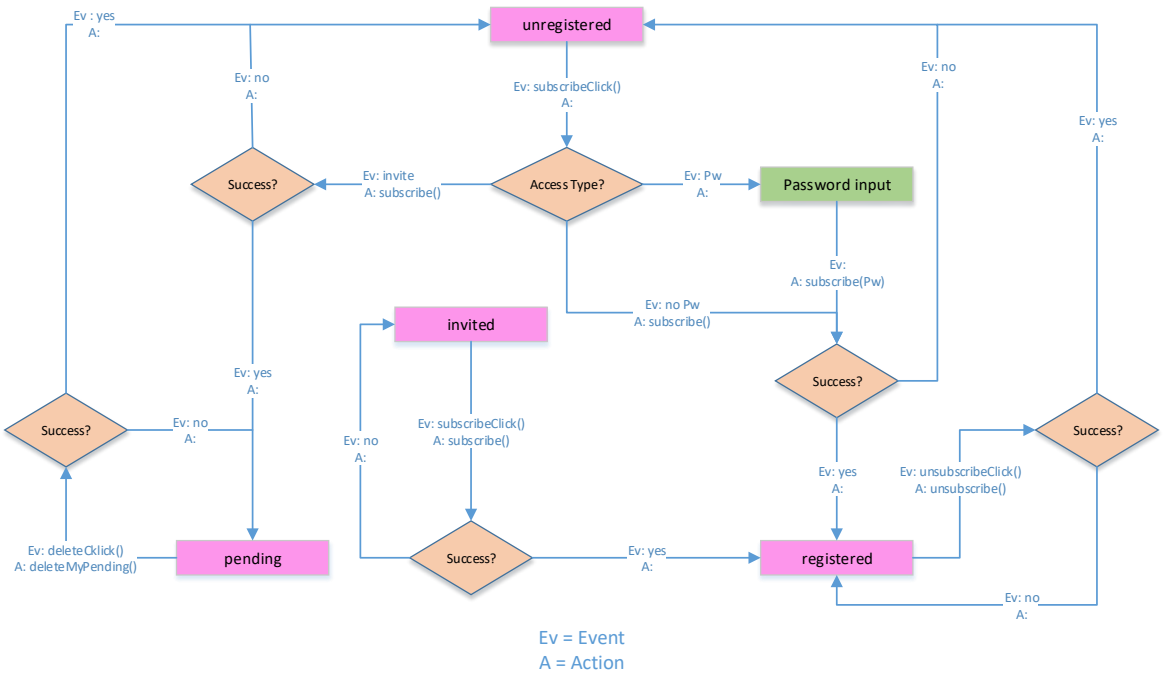

Abbildung 25: Studien: Status und Statuswechsel

Die verschiedenen Studien werden von der *API* geladen und mit Hilfe von *GSON* werden aus den *JSON*-Objekten die einzelnen Studien kreiert. Dafür musste folgende Klasse definiert werden (Listing 1).

Listing 1: Study

```
0 public class Study {
1
2     public String type;
3     public String id;
4     public Attributes attributes;
5
6
7     public String getType() {
8         return type;
9     }
10
11    public void setType(String type) {
12        this.type = type;
13    }
14
15    public String getId() {
16        return id;
17    }
18
19    public void setId(String id) {
20        this.id = id;
21    }
22
23    public Attributes getAttributes() {
24        return attributes;
25    }
26
27    public void setAttributes(Attributes attributes) {
28        this.attributes = attributes;
29    }
30
31    public class Attributes {
32        public String name;
33        public String title;
34        public String description;
35        public String accesstype;
36        public int starts_at;
37        public int ends_at;
38        public int is_private;
39        public boolean is_running;
40        public String text;
41        public String consenttext;
42    }
43 }
```

Zusätzlich dazu wurde eine Klasse StudyEx implementiert, in welcher ein zusätzlicher *String state* (Listing 2, Zeile 1) gehalten wird. Dieser wird je nach User und dessen aktuellen Zustand, bezüglich einer Studie gesetzt. Mögliche Zustände können *none*, *invited*, *pending* oder *joined* sein. Weiter wird hier auch die Variable *isClicked* (Listing 2, Zeile 3) definiert, welche für die spätere Visualisierung der Studienliste und für die Interaktion durch den User von Bedeutung ist.

Listing 2: StudyEx

```
0 public class StudyEx {
1    public String state;
2    public Study study;
3    public boolean isClicked = false;
4 }
```

Wird die *StudiesActivity* (Listing 3), d.h. die Ansicht mit der Liste der Studien vom Benutzer angefordert, dann wird in der *onCreate*-Funktion (Zeile 8) die *APIgetAllStudies*-Funktion des *ApiControllers* aufgerufen, wodurch alle Studien von der Datenbank geladen werden. Dabei wird dem *ApiController* die aktuelle *Activity* direkt als Parameter mitgeben (Zeile 13). Dadurch wird sichergestellt, dass der *Callback* wieder zu dieser *Activity* zurückkommt und die entsprechende *getAllStudiesCallback*-Funktion aufgerufen wird (Zeile 134). In Zeile 15 wird die Funktion *showList* aufgerufen. Diese erstellt die Liste bzw. die Ansicht der Liste der Studien für den User. Dafür wird zuerst die Liste der *activeStudies,* mit Hilfe der Funktion *getActiveStudies* des *ApiControllers,* mit den aktiven Studien aus der Datenbank gefüllt (Zeile 17), da nur diese aktuell relevant sind. Diese Liste wird dem *StudyListAdapter* – einer Klasse, die von *ArrayAdapter*, einem Android-Tool zur Verwaltung von Listen, erbt – übergeben, woraufhin dieser Adapter dem *ListView* übergeben wird, um die Liste an Studien letztendlich zu visualisieren (Zeile 18 und 19). Zwischen Zeile 22 und 49 wird der Code Dialog für Passwort Abfragen zu privaten Studien implementiert. Die anfangs als Parameter „eingehende“ Studie wird in Zeile 42, zusammen mit dem vom User angegeben Code (erhalten in Zeile 38), an die *subscribeStudy*-Funktion im *ApiController* übergeben. Die Funktion *changeStudyState* (Zeile 51-87) löst die benötige, korrekte und zustandsbedingte Reaktion aus, welche ein User für die Studien durch entsprechende Interaktionen auf der Oberfläche anstößt. Unterschieden wird anhand des *myStudyState*, also anhand des jetzigen Zustandes der Studie in Bezug auf den eingeloggten User (Zeile 52). Dabei sind vier verschiedene Zustände (*none*, *joined*, *pending*, *intivted*) möglich, wie schon in Abbildung 25 zu sehen ist, wobei *none unregistered* und *joined* registered entspricht. Ist der User, der ausgewählten Studie noch nicht beigetreten (Status *none*, Zeile 54), wird nochmals unterschieden, ob die gewählte Studie privat oder offen ist. Einer offenen Studie oder einer Studie, für welche dem User bereits eine Einladung vorliegt, kann der User sofort beitreten (*ApiController*.*subscribeStudy*, Zeile 69 und *case*: „*invited*“, Zeilen 80-83). Ist die Studie privat, ergo vom *Accesstype* ungleich null, muss ein weiteres Mal unterschieden werden. Wird bezüglich des *Accesstypes* ein Passwort für die Studie angefordert, wird lediglich der zuvor erwähnte Code Dialog gestartet (Zeile 58). Wird kein Passwort benötigt, wird, mittels der Funktion *subscribePendingStudy*, eine Einladung zur entsprechenden Studie angefordert (Zeile 63). Ist der User bereits Mitglied einer Studie (*joined*) oder hat bereits eine Einladung angefordert (*pending*), dann kann er diese Studie auch wieder verlassen oder die Einladungsanfrage zurückziehen (Zeile 72-79). Zu erwähnen wäre auch, dass die Funktion *subscribeStudy* für private (passwortgeschützte), wie für offene Studien dieselbe ist. Bei offenen Studien wird lediglich der *code* Parameter genullt. Der eigentliche Zustandswechsel bezüglich des Users und seinen Studien erfolgt im *ApiController* und den hierfür implementierten Funktionen. Der Funktion *studyActionCallback* (Zeile 90) wird, über die *ApiController*-Funktionen *ApiCallback* und *studyActionDone*, der Parameter *action* übergeben. Anhand diesem wird nun unterschieden welche Status Nachricht dem User ausgegeben wird. Bei einem zusätzlichen wahren *Boolean*-Wert (Zeile 92-108) wird entsprechend eine erfolgreiche Nachricht ausgegeben, wohingegen

ein falscher *Boolean*-Wert dementsprechend zu einer Fehlermeldung führt (Zeile 113-127). Die letzte Funktion dieser Klasse (*getAllStudiesCallback*, Zeile 134) startet entweder die oben beschriebene Funktion *showList* (*success* gleich *true*) oder gibt eine Fehlermeldung aus, je nach übergebenem *Boolean*-Wert.

Listing 3: StudiesActivity

```
0  public class StudiesActivity extends MenuActivity {
1
2      public ListView lvStudies;
3      public ListAdapter studiesAdapter;
4      public List<StudyEx> activeStudies;
5      public AlertDialog codeDialog;
6      @Override
7      protected void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.activity_studies);
10         myView = this;
11         lvStudies = (ListView) findViewById(R.id.lvStudies);
12         showLoading();
13         new ApiController(this).APIgetAllStudies();
14     }
15     void showList()
16     {
17         activeStudies = new ApiController(this).getActiveStudies();
18         studiesAdapter = new StudyListAdapter(this,activeStudies);
19         lvStudies.setAdapter(studiesAdapter);
20     }
21
22     private void showCodeDialog(final StudyEx myStudy){
23         View codeDialogView = getLayoutInflater().inflate(R.layout.code_dialog,null);
24         AlertDialog.Builder codeDialogBuilder = new AlertDialog.Builder(this);
25         final EditText etCodeDialog = (EditText)codeDialogView.findViewById(R.id.etCodeDialog);
26         Button btnSendCodeDialog = (Button) codeDialogView.findViewById(R.id.btnSendCodeDialog);
27         Button btnCancelCodeDialog = (Button) codeDialogView.findViewById(R.id.btnCancelCodeDialog);
28
29         btnCancelCodeDialog.setOnClickListener(new View.OnClickListener() {
30             @Override
31             public void onClick(View view) {
32                 codeDialog.cancel();
33             }
34         });
35         btnSendCodeDialog.setOnClickListener(new View.OnClickListener() {
36             @Override
37             public void onClick(View view) {
38                 String code = etCodeDialog.getText().toString();
39                 codeDialog.cancel();
40
41                 myView.showLoading();
42                 new ApiController(myView).subscribeStudy(myStudy,code);
43             }
44         });
45         codeDialogBuilder.setView(codeDialogView);
46         codeDialog = codeDialogBuilder.create();
47         codeDialog.setCancelable(false);
48         codeDialog.show();
49     }
50
51     public void changeStudyState(final StudyEx myStudy){
52             switch(myStudy.state)
53             {
54                 case "none":
55                     if  (myStudy.study.attributes.accesstype != null){
56                        if (myStudy.study.attributes.accesstype.equals("password"))
57                         {
58                             showCodeDialog(myStudy);
59                         }
60                         else
61                        {
62                            showLoading();
63                            new ApiController(myView).subscribePendingStudy(myStudy, null);
64                        }
65                     }
66                     else
67                     {
68                         showLoading();
69                         new ApiController(myView).subscribeStudy(myStudy, null);
70                     }
71                     break;
72                 case "joined":
73                     showLoading();
74                     new ApiController(myView).unsubscribeStudy(myStudy);
75                     break;
76                 case "pending":
77                     showLoading();
```

```
78                      new ApiController(myView).deletePendingStudy(myStudy);
79                      break;
80                  case "invited":
81                      showLoading();
82                      new ApiController(myView).subscribeStudy(myStudy, null);
83                      break;
84                  default:
85                      break;
86              }
87      }
88
89      @Override
90      public void studyActionCallback(boolean success, String action) {
91          loadingDialog.cancel();
92          if(success)
93          {
94              switch(action)
95              {
96                  case "subscribe":
97                      showStatus("Successful subscribed");
98                      break;
99                  case "subscribeInvite":
100                     showStatus("Successful requested");
101                     break;
102                 case "unsubscribe":
103                 case "delete":
104                     showStatus("Successful unsubscribed.");
105                     break;
106
107                 default:
108                     break;
109             }
110         }
111         else
112         {
113             switch(action)
114             {
115                 case "subscribe":
116                     showStatus("Subscribing failed");
117                     break;
118                 case "subscribeInvited":
119                     showStatus("Request failed");
120                     break;
121                 case "unsubscribe":
122                 case "delete":
123                     showStatus("Unsubscribing failed.");
124                     break;
125
126                 default:
127                     break;
128             }
129         }
130         showList();
131     }
132
133     @Override
134     public void getAllStudiesCallback(boolean success) {
135         loadingDialog.cancel();
136         if(success)
137         {
138             showList();
139         }
140         else
141         {
142             showStatus("Load of studies failed. Offline mode.");
143         }
144     }
145 }
```

In Listing 4 wird die Implementierung, der zuvor bereits erwähnten Klasse *StudyListAdapter*, aufgezeigt. Diese Klasse erbt von der Android-Klasse Array Adapter (Zeile 0) und dient dazu, Listen zu organisieren und auch zu visualisieren. Im Konstruktor wird zuerst die Layout Ressource, eine einzelne sogenannte *Row* (für Zeile/Reihe) einer Studie definiert. In dieser *Row* wurde die grundlegende Struktur eines einzelnen Listenelements festgelegt. Ein (*Row*-) Element ist dabei zusammengesetzt aus einem *TextView* für die Beschreibung der Studie, einem *Button* für den Studientitel und einem *Button* für die jeweilige Beitritt- oder Austritt-Aktion, bezüglich einer Studie. In der Funktion *getView* (Zeile 11-75) wird ein solches Listenelement, mit Hilfe eines *LayoutInflater*[14]

[14] [42]

und auf Grundlage des *studyRow_Layouts* aufgebaut. Nachdem in Zeile 15 die ausgewählte Studie festgehalten wird, werden zwischen Zeile 16 und 18 die einzelnen Bestandteile eines *Row*-Elements über die jeweilige *Id* der *Layout Ressource* zugewiesen, um den Zugriff auf die *Ressourcen*-Elemente zu bekommen. Im Folgenden wird nun wieder über den *StudyState* entschieden (Zeile 20 - 44), wie die einzelnen Listenelemente dem User visualisiert werden sollen. Dabei werden zusätzlich zu den Texten der *Adapter-Buttons* möglichst intuitiv verständliche Farben für die jeweiligen Studienzustände des Benutzers und dessen möglichen Aktionen darauf, gesetzt (Bsp.: Zeile 22-26, Status *joined*, d.h. beigetreten, worauf der *Adapter-Button* den Text, zum Verlassen der Study, sowie eine rote Färbung erhält. Zusätzlich wird der Titel der Studie (hier grün) eingefärbt, um die beigetretenen Studien in der Liste speziell hervorzuheben. Analog erfolgt dies für die anderen Zustände.). Ab Zeile 46 wird dem *Titel Button* ein *onClickListener*[15] hinzugefügt, um die Beschreibung der Studie, ebenso wie den *Adapter-Button*, bei entsprechender Interaktion, jeweils ein- oder auch wieder auszublenden. Ob diese Elemente ein- bzw. ausgeblendet sind, wird anhand der weiter oben, in Listing 2 erwähnten *Boolean* Variable *isClicked,* entschieden. Dem *Adapter-Button* wird in Zeile 67-73 noch ein *onClickListener* zugwiesen, um den Status einer Studie ändern zu können, wie dies z.B. bei einem Studienbeitritt der Fall ist. Den Titel (*Title*) und die Beschreibung (*Description*) der behandelten Studie erhält man über den Pointer in *StudyEx* (Listing 2). Dem *Titel-Button* wird nun der Titel und dem *Textfeld* die Beschreibung zugewiesen (Zeile 63 und 64). Ganz am Schluss der Klasse, wird der anfangs angelegte (Zeile 14) und nun (aus *Title*, *Description* und *ButtonAdapter*) zusammengesetzte *customView,* zurückgegeben (Zeile 74). So wird eine Studie nach der anderen, Reihe für Reihe, abgearbeitet und erstellt.

Listing 4: StudyListAdapter

```
0 public class StudyListAdapter extends ArrayAdapter {
1     Context context;
2     SharedPreferences sharedTexts ;
3
4     public StudyListAdapter(Context c, List<StudyEx> studies){
5         super(c,R.layout.study_row,studies);
6         context = c;
7         sharedTexts = c.getSharedPreferences("Texts" + ApiController.local, Context.MODE_PRIVATE);
8     }
9     @NonNull
10    @Override
11    public View getView(int position, @Nullable View convertView, @NonNull ViewGroup parent) {
12
13        LayoutInflater inflater = LayoutInflater.from(getContext());
14        View customView = inflater.inflate(R.layout.study_row,parent,false);
15        final StudyEx singleStudy = (StudyEx) getItem(position);
16        final Button btnStudyTitle = (Button) customView.findViewById(R.id.btnStudyTitle);
17        final TextView tvStudyAdapterDescription = (TextView)
              customView.findViewById(R.id.tvStudyAdapterDescription);
18        final Button btnStudyAdapter = (Button) customView.findViewById(R.id.btnStudyAdapter);
19
20        switch(singleStudy.state){
21
22            case "joined":
23                btnStudyAdapter.setText(sharedTexts.getString("app.study.unsubscribe","Unsubscribe
                  study"));
24                btnStudyAdapter.setBackgroundColor(Color.RED);
25                btnStudyTitle.setBackgroundColor(Color.GREEN);
26                break;
27            case "pending":
28                btnStudyAdapter.setText("Delete Pending Study");
29                btnStudyAdapter.setBackgroundColor(Color.GRAY);
30                btnStudyTitle.setBackgroundColor(Color.CYAN);
31                break;
32            case "invited":
33                btnStudyAdapter.setText("Accept Invitation.");
34                btnStudyAdapter.setBackgroundColor(Color.GREEN);
35                btnStudyTitle.setBackgroundColor(Color.YELLOW);
36                break;
37            case "none":
38                btnStudyAdapter.setText(sharedTexts.getString("app.study.subscribe","Subscribe study"));
39                btnStudyAdapter.setBackgroundColor(Color.GREEN);
40                btnStudyTitle.setBackgroundColor(Color.LTGRAY);
41                break;
```

[15] [10]

```
42            default:
43                break;
44        }
45
46        btnStudyTitle.setOnClickListener(new View.OnClickListener() {
47            @Override
48            public void onClick(View view) {
49                if(singleStudy.isClicked == false) {
50                    tvStudyAdapterDescription.setVisibility(View.VISIBLE);
51                    btnStudyAdapter.setVisibility(View.VISIBLE);
52                    singleStudy.isClicked = true;
53                }
54                else
55                {
56                    tvStudyAdapterDescription.setVisibility(View.GONE);
57                    btnStudyAdapter.setVisibility(View.GONE);
58                    singleStudy.isClicked = false;
59                }
60            }
61        });
62        singleStudy.isClicked = false;
63        tvStudyAdapterDescription.setText(singleStudy.study.attributes.description);
64        btnStudyTitle.setText(singleStudy.study.attributes.title);
65
66
67        btnStudyAdapter.setOnClickListener(new View.OnClickListener() {
68            @Override
69            public void onClick(View view) {
70                StudiesActivity myActivity = (StudiesActivity) context;
71                myActivity.changeStudyState(singleStudy);
72            }
73        });
74        return customView;
75    }
76 }
```

In der Klasse *ApiController* sind, wie in den nächsten Abschnitten aufgelistet, mehrere Funktionen implementiert, die jeweils von außen bzw. aus anderen Klassen aufgerufen werden. So auch bei Vorgängen, die die verfügbaren Studien betreffen (Listing 5, Listing 6), wie z.B. die *APIgetAllStudies* (Listing 5, Zeile 0), welche in der *onCreate*-Funktion der *StudiesActivity* aufgerufen werden. Die Vorgehensweise ist generell bei all den Funktionen in Listing 5 dieselbe, einzige Ausnahme ist *getMyStudiesState* in Zeile 48, wobei lediglich drei Funktionen aufgerufen werden (hier werden im Übrigen auch 3 *Asynctasks* parallel gestartet, weswegen die Kontrollvariable *count* eingeführt wurde). Bei den Anderen wird zuerst ein *ApiJob* kreiert, eine eigens definierte Klasse, zu verstehen als eine Art Container für die Request an – und das erhaltene *Result* von der *API* (Zeile 1 und 2, 13 und 14, 25 und 26 usw.). Weiter wird dem *Request* jeweils der entsprechende URL-Pfad mitgegeben(Zeile 4, 16, 28, usw.). Im nächsten Schritt wird bei der *Request,* anhand der http-Methode unterschieden, ob etwas geladen, gesendet oder gelöscht werden soll (*GET* Zeile 5, 17, usw.; *POST* Zeile 72 und 85; *DELETE* Zeile 60 und 98). *SetJsonData* wird nur bei einer *POST*-http-Methode benötigt, da nur dann Daten an die *API* versendet werden müssen, ansonsten wird hier lediglich ein Leerstring übergeben (Bsp. Leerstring: Zeile 6, Bsp. Daten: Zeile 73). Muss etwas an die *API* versendet werden, dann wird im *setJsonData,* die hierfür, im *JsonParser* implementierte Funktion aufgerufen und die zu sendenden Daten werden dort weiter verarbeitet. Zusätzlich wird bei jeder Funktion eine sogenannte *doneMethode* gesetzt. Über diese *doneMethode* (Zeile 7, 19, usw.) wird – wie in Listing 6 (Zeile 5) zu sehen – entschieden, welche (*Callback*-) Funktion letztendlich wieder aufgerufen wird. Die *ID* wiederrum, wird für die Studien nur bei *POST*- und *DELETE*-http-Methoden benötigt (Bsp. Zeile 63 und 75), da dann die entsprechende Studie referenziert werden muss. Im *GET*-Fall wird ebenfalls nur ein Leerstring mitgegeben (Bsp. Zeile 8). Am Ende dieser ganzen einzelnen Funktionen wird jeweils eine *ApiConnectionTask*, sprich ein *Asynctask* gestartet, wobei dem *ApiConnectionTask* der Controller, und der *execute*-Funktion des *Asynctasks* der komplette *Job* übergeben werden. (Zeile 9, 21, 33, usw.).

Listing 5: ApiController Studies Funktionen

```
0  public void APIgetAllStudies(){
1          ApiJop job = new ApiJop();
2          ApiRequest request = job.getRequest();
3
4          request.setUrltitle("/api/v1/studies?token="+token);
5          request.setHttpMethod("GET");
6          request.setJsonData("");
7          request.setDoneMethod("allStudies");
8          request.setId("");
9          new ApiConnectionTask(this).execute(job);
10     }
11
12     private void APImyStudies(){
13         ApiJop job = new ApiJop();
14         ApiRequest request = job.getRequest();
15
16         request.setUrltitle("/api/v1/my/studies?token="+token);
17         request.setHttpMethod("GET");
18         request.setJsonData("");
19         request.setDoneMethod("myStudies");
20         request.setId("");
21         new ApiConnectionTask(this).execute(job);
22     }
23
24     private void APImyPendingStudies(){
25         ApiJop job = new ApiJop();
26         ApiRequest request = job.getRequest();
27
28         request.setUrltitle("/api/v1/my/studies/pending?token="+token);
29         request.setHttpMethod("GET");
30         request.setJsonData("");
31         request.setDoneMethod("myPendingStudies");
32         request.setId("");
33         new ApiConnectionTask(this).execute(job);
34     }
35
36     private void APImyInvitedStudies(){
37         ApiJop job = new ApiJop();
38         ApiRequest request = job.getRequest();
39
40         request.setUrltitle("/api/v1/my/invitations?token="+token);
41         request.setHttpMethod("GET");
42         request.setJsonData("");
43         request.setDoneMethod("myInvitedStudies");
44         request.setId("");
45         new ApiConnectionTask(this).execute(job);
46     }
47
48     public  void getMyStudiesState(){
49         count = 0;
50         APImyStudies();
51         APImyPendingStudies();
52         APImyInvitedStudies ();
53     }
54
55     public void unsubscribeStudy(StudyEx myStudy){
56         ApiJop job = new ApiJop();
57         ApiRequest request = job.getRequest();
58
59         request.setUrltitle("/api/v1/studies/"+myStudy.study.id+"/unsubscribe?token="+token);
60         request.setHttpMethod("DELETE");
61         request.setJsonData("");
62         request.setDoneMethod("unsubscribeStudy");
63         request.setId(myStudy.study.id);
64         new ApiConnectionTask(this).execute(job);
65     }
66
67     public void subscribeStudy(StudyEx myStudy, String code){
68         ApiJop job = new ApiJop();
69         ApiRequest request = job.getRequest();
70
71         request.setUrltitle("/api/v1/studies/"+myStudy.study.id+"/subscribe?token="+token);
72         request.setHttpMethod("POST");
73         request.setJsonData(JsonParser.setSubscribeStudy(code));
74         request.setDoneMethod("subscribeStudy");
75         request.setId(myStudy.study.id);
76         request.addHeader("Content-Type","application/json");
77         new ApiConnectionTask(this).execute(job);
78     }
79
80     public void subscribePendingStudy(StudyEx myStudy, String code){
81         ApiJop job = new ApiJop();
82         ApiRequest request = job.getRequest();
83
84         request.setUrltitle("/api/v1/studies/"+myStudy.study.id+"/subscribe?token="+token);
85         request.setHttpMethod("POST");
```

```
86          request.setJsonData(JsonParser.setSubscribeStudy(code));
87          request.setDoneMethod("subscribePendingStudy");
88          request.setId(myStudy.study.id);
89          request.addHeader("Content-Type","application/json");
90          new ApiConnectionTask(this).execute(job);
91      }
92
93      public void deletePendingStudy(StudyEx myStudy){
94          ApiJop job = new ApiJop();
95          ApiRequest request = job.getRequest();
96
97          request.setUrltitle("/api/v1/my/studies/pending/"+myStudy.study.id+"?token="+token);
98          request.setHttpMethod("DELETE");
99          request.setJsonData("");
100         request.setDoneMethod("deletePendingStudy");
101         request.setId(myStudy.study.id);
102         new ApiConnectionTask(this).execute(job);
103     }
```

Wie schon erwähnt wurde im *ApiController* eine *Callback*-Funktion implementiert. In dieser wird, anhand der zuvor (wie in Listing 5 zu sehen) gesetzten *doneMethode,* der weitere Ablauf entschieden. Hier in Listing 6 werden lediglich die für die Studien-Funktionalität relevanten Fälle verdeutlicht. Das Symbol in Zeile 7 soll verdeutlichen, dass dort eigentlich noch mehrere Funktionen implementiert wurden. Man kann hier die Fälle *myStudies* (Zeile 15), *myPendingStudies* (Zeile 19) und *myInvitedStudies* (Zeile 23) zusammenfassen, weil diese jeweils die klasseninterne Funktion *myStudiesDone* aufrufen (Bsp. Zeile 16). Dabei wird schlicht das *Result* aus dem *ApiJob* und ein neuer Status bezüglich der Studie übergeben. Ebenso kann man die Fälle *unsubscribeStudy* (Zeile 28), *subscribeStudy* (Zeile 32), *subscribePendingStudy* (Zeile 36) und *deletePendingStudy* (Zeile 40) gruppieren, welche lediglich die klasseninterne Funktion *studyActionDone* (Bsp. Zeile 29) aufrufen. Hierbei wird die *Request* und das *Result* aus dem *ApiJob*, ebenfalls ein neuer Status, sowie ein *callback*-String übergeben. Ausnahme in dieser Funktion ist der Fall *allStudies* (Zeile 9). Liegt kein Fehler vor, dann wird die Liste *allStudies* (liegt statisch im *ApiController* vor), mit Hilfe des *JsonParser,* mit allen Studien aus dem *JSON-Paket* gefüllt. Direkt darauf wird dann die oben beschriebene Funktion *getMyStudiesState* aufgerufen (Zeile 12 und Listing 5, Zeile 48).

Listing 6: ApiCallback Studies

```
0 public void APIcallback(ApiJop job){
1
2         ApiRequest request = job.getRequest();
3         ApiResult result = job.getResult();
4
5         switch(request.getDoneMethod()){
6
7           [.....]
8
9             case "allStudies":
10                if( StatusFehlerOrEx(result) == true){ myActivity.getAllStudiesCallback(false);
                 return;};
11                allStudies = JsonParser.getAllStudies(result.getResultString());
12                getMyStudiesState();
13                break;
14
15            case "myStudies":
16                myStudiesDone(result,"joined");
17                break;
18
19            case "myPendingStudies":
20                myStudiesDone(result,"pending");
21                break;
22
23            case "myInvitedStudies":
24                myStudiesDone(result,"invited");
25                break;
26
27
28            case "unsubscribeStudy":
29                studyActionDone(request,result,"none","unsubscribe");
30                break;
31
32            case "subscribeStudy":
33                studyActionDone(request,result,"joined","subscribe");
34                break;
35
```

```
36            case "subscribePendingStudy":
37                studyActionDone(request,result,"pending","subscribeInvite");
38                break;
39
40            case "deletePendingStudy":
41                studyActionDone(request,result,"none","delete");
42                break;
43
44            default:
45                break;
46        }
47
48    }
```

Listing 7 zeigt die Implementierung der in Listing 6 aufgerufenen Funktionen. *MyStudiesDone* bekommt, wie erwähnt, ein *ApiResult* aus dem *ApiJob* und einen neuen Status übergeben. Zuerst wird überprüft, ob kein Fehler vorlag (Zeile 2). Als nächstes wird die Funktion *fillMyStudyState*, mit dem im *ApiResult* enthaltenen *ResultString* und dem neuen Status der Studie aufgerufen. Danach wird die Kontrollvariable *count* inkrementiert, womit letztendlich überprüft wird, ob auch alle 3, in Listing 5 (Zeile 48) gestarteten *Asynctasks*, per *Callback* zurückgekommen sind. Nur dann wird dem *getAllStudiesCallback* ein wahrer *Boolean*-Wert übergeben (Zeile 4-7). Hat sich ein User zu einer neuen Studie angemeldet oder eine Studie verlassen etc., dann wird im Programmverlauf die Funktion *studyActionDone* aufgerufen. Wiederum wird zuerst eine Abfrage bezüglich eines Fehlers durchlaufen. Ist alles korrekt, werden die Funktionen *changeMyStudyState* – mit der ID aus der *Request* und einem neuen Status (Zeile 13) –, sowie die *studyActionCallback,* mit einem wahren *boolschen* Wert (Zeile 14) aufgerufen. *FillMyStudyState* dient lediglich dazu, die aktuellen Zustände der geladenen Studien, in Bezug auf den User, zu setzen. Dies geschieht über zwei geschachtelte Schleifen, wobei jeweils über die *ID* verglichen wird (Zeile 19-25). Ist die *ID* einer Studie in der *StudyListe* (*myStudies*) gleich der *ID* einer *StudyEx* in der *StudyExListe* (*allStudies*), dann wird der Status der *StudyEx*, auf den übergeben bekommenen Status (*setState*) gesetzt (Zeile 24). *ChangeMyStudyState* verändert lediglich den Status einer Studie, je nachdem wie der User mit dieser Studie interagiert hat. Programmtechnisch gelöst wurde dies, indem man die Liste der Studien (*StudyExListe*) eines Users durchläuft (Zeile 32) und über die *ID* der Studie abgleicht. Ist der Vergleich korrekt, so wird der bisherige Status vom eingehenden, neuen Status überschrieben. *GetActiveStudies* dient lediglich dazu, nur die aktiven Studien aus allen Studien zu laden, d.h. nur diejenigen bei denen das Attribut *running* wahr ist (Zeile 46).

Listing 7: Weitere ApiController Studies Funktionen

```
0  public void myStudiesDone(ApiResult result, String newState){
1
2          if( StatusFehlerOrEx(result) == true){ myActivity.getAllStudiesCallback(false);  return;};
3          fillMyStudyState(result.getResultString(),newState);
4          count++;
5          if(count == 3)
6          {
7              myActivity.getAllStudiesCallback(true);
8          }
9      }
10
11     public void studyActionDone(ApiRequest request, ApiResult result, String newState, String callback){
12         if( StatusFehlerOrEx(result) == true){  myActivity.studyActionCallback(false,callback); return;};
13         changeMyStudyState(request.getId(),newState);
14         myActivity.studyActionCallback(true,callback);
15     }
16
17    public void fillMyStudyState(String result, String setState){
18         List<Study> myStudies = JsonParser.getStudies(result);
19         for (Study study: myStudies)
20         {
21             for (StudyEx myStudy:allStudies)
22             {
23                 if(study.id.equals(myStudy.study.id)){
24                     myStudy.state = setState;
25                     break;
26                 }
27             }
28         }
```

```
29     }
30
31     public void changeMyStudyState(String id, String newState){
32         for (StudyEx myStudy:allStudies)
33         {
34             if(id.equals(myStudy.study.id)){
35                 myStudy.state = newState;
36                 break;
37             }
38         }
39     }
40
41    public List<StudyEx> getActiveStudies()
42     {
43         List<StudyEx> returnList = new ArrayList<>();
44         for (StudyEx myStudy:allStudies)
45         {
46             if(myStudy.study.attributes.is_running){
47                 returnList.add(myStudy);
48             }
49         }
50         return returnList;
51     }
52
53}
```

6. Anforderungsabgleich

In diesem Kapitel werden die an die App gestellten Anforderungen, die in Kapitel 2 definiert wurden, mit der Implementierung und den Funktionen der App abgeglichen. Wie schon bei der Definition der Anforderungen, wird wiederum in funktionale und nichtfunktionale Anforderungen unterteilt.

6.1. Funktionale Anforderungen

Der folgende Abschnitt zeigt den Abgleich der definierten funktionalen Anforderungen an die App und überprüft, ob in der momentanen Implementierung alle funktionalen Anforderungen erfüllt worden sind.

Nummer	Beschreibung	Problembeschreibung
1	Kontinuierliche Fragebögen zur Überwachung des Stresslevels	Anforderung erfüllt siehe 4.1.4 und 4.1.5
2	(Demographischer) Fragebogen zur Überwachung des Stresslevels	Anforderungen erfüllt siehe 4.1.4 und 4.1.5
3	Anzeige der möglichen Ergebnisse	Anforderungen erfüllt siehe 4.1.8 (Ergebnisse)
4	Registrierung in der App	Anforderungen erfüllt siehe 4.1.2
5	Sliderfragen ohne initialen Wert	Anforderungen erfüllt siehe 4.1.6 (Slider)
6	Ausfüllen der statistischen Fragebögen in der App	Anforderungen erfüllt siehe 4.1.5 und 4.1.6
7	Synchronisierung der Ergebnisse	Anforderungen erfüllt siehe 4.1.5 und 4.1.6. Die ausgefüllten Fragebögen werden an die *API*/Datenbank gesendet.
8	Meldung von Benachrichtigungen	Anforderungen erfüllt siehe 4.1.8 (Benachrichtigungen)
9	Messung des Geräuschpegels	Anforderungen erfüllt, siehe 4.1.6. Während des Ausfüllens eines Fragebogens werden unterschiedlich oft der Lautstärkepegel in Dezibel gemessen und beim Speichern/Senden mitgeschickt. Die Anzahl der Messung richtet sich dabei je nach gesetztem *amount*-Wert oder *each*-Wert in der *API*
10	App auch ohne Internetverbindung nutzbar machen	Anforderungen erfüllt siehe 4.1.9. Die App besitzt einen Offline-Modus. Dieser ermöglicht auch ohne bestehende Internetverbindung weiterhin das Bearbeiten von Fragebögen, was die Hauptfunktion der App darstellt.
Studienteilnahme		
11	An verschiedenen Studien teilnehmen	Anforderungen erfüllt siehe 4.1.7
12	Studieneinladung annehmen	Anforderungen erfüllt siehe 4.1.7
13	An privater Studie teilnehmen	Anforderungen erfüllt siehe 4.1.7

6.2. Nichtfunktionale Anforderungen

Im folgenden Abschnitt werden die oben definierten nichtfunktionalen Anforderungen an die App abgeglichen.

Nummer	Beschreibung	Problembeschreibung
1	Ausfüllen des Fragebogens zur Überwachung der Stresswahrnehmung sollte in möglichst kurzer Zeit gelingen	Anforderung größtenteils erfüllt. Einzig der demographische Fragebogen benötigt eine etwas längere Bearbeitungszeit. Die täglich/wöchentlich/monatlich zu bearbeitenden Fragebögen sind in recht kurzer Zeit zu bewältigen.
2	Benutzbarkeit	Bestmöglich erfüllt. Es wurde darauf geachtet, dass die App intuitiv und leicht verständlich ist. Falsche oder unerwünschte Eingaben sind bestmöglich ausgeschlossen. Entsprechende Fehlermeldungen werden bei unerwünschtem Verhalten, seitens des Users oder der App, ausgegeben.

7. Vergleichbare Projekte

Das folgende Kapitel befasst sich mit ähnlichen Projekten wie das *Track Your Stress* Projekt. Die einzelnen Projekte werden kurz erläutert und es wird auf etwaige Gemeinsamkeiten eingegangen.

7.1. Track Your Tinnitus (TYT)

Das Projekt *Track Your Tinnitus*, ist ein Projekt zur Überwachung der Schwankungen der Tinnituswahrnehmung. Es existiert bereits seit 2014 und soll dazu dienen, die Forschung, sowie betroffene Patienten dieser schwierig zu behandelnden und oftmals stark belastenden Krankheit, die in schwerwiegenden Fällen sogar bis zum Selbstmord führen kann, zu unterstützen. *„Tinnitus bezeichnet ein Symptom, bei dem der Betroffene, Töne oder Geräusche wahrnimmt, ohne dass in der unmittelbaren Umgebung eine physikalische Ursache für das akustische Signal zu finden ist"*[16]. Die Überwachung soll vor allem dazu dienen, genauere Angaben über die, bei 60% der Patienten, von Tag zu Tag unterschiedlich starke Tinnituswahrnehmung zu ermitteln, sowie mögliche Ursachen dafür zu identifizieren. Das Projekt ist darauf ausgerichtet, dass der Benutzer seine persönliche Tinnituswahrnehmung, zu unterschiedlichen Zeitpunkten, mittels den verfügbaren Apps dokumentiert. Bisher wurden Schwankungen der Tinnituswahrnehmung einzelner Patienten nur mit sogenannten Tinnitus-Tagebüchern festgehalten, womit eine Erfassung des exakten, zeitlichen Verlaufs nicht wirklich möglich ist. Die Apps haben den Vorteil, dass zum einen eine möglichst exakte, detaillierte Aufzeichnung des zeitlichen Verlaufs der wahrgenommenen Lautstärke des Tinnitus ermöglicht wird und zum anderen, dass auch teils stressige Alltagssituationen sowie Hintergrundgeräusche bzw. die Hintergrundlautstärke, die eventuell mit der Tinnituswahrnehmung korrelieren, mit einbezogen werden können. Weiter wird dem Benutzer eine Auswertung seiner Wahrnehmungen bereitgestellt und entsprechend visualisiert, womit auch ihm eine gewisse Reflektion seiner Tinnituswahrnehmung ermöglicht wird. Das Projekt entstand 2014 im Rahmen einer Diplomarbeit von Jochen Herrmann an der Universität Ulm und wird, auch mit Hilfe weiterer Abschlussarbeiten, kontinuierlich weiterentwickelt. Die Apps sind derzeit im GooglePlay Store und im Apple AppStore erhältlich.[17] [18] [19]

7.2. Assess Your Stress (AYS)

Das *Assess Your Stress* Projekt baut im Grunde auf dem *Track Your Tinnitus* Projekt und dessen Erfolg auf. Ebenso ist es ein Projekt an der Uni Ulm, wobei das TYT Projekt grundlegend überarbeitet und neu aufgebaut werden soll. Grund für das Projekt ist ein neues Gesetz des Bundestags, das sogenannte Präventionsgesetz.[20] Ein Gesetz, welches Prävention von Krankheiten, Gesundheitsförderung, sowie Früherkennung von Krankheiten verbessern soll. Unter anderem will man gerade Unternehmen dazu bewegen, entsprechende Maßnahmen zur Erhaltung und Verbesserung der Gesundheit ihrer Arbeitnehmer zu treffen. Da Stress heutzutage oftmals als ein großer Faktor, in Bezug auf Krankheiten oder auch als Auslöser von Krankheiten, vermutet wird, wurde das Projekt AYS, in Kooperation der Universitäten Ulm und Regensburg gegründet. So ist das „große" AYS Projekt Auslöser für das, in dieser Arbeit vorgestellte *Track Your Stress* Projekt, welches einen ersten Schritt zur Erfassung des Stresslevels auf Android Geräten darstellt. Dabei sollen Benutzer der App kontinuierlich wiederkehrende Fragebögen, in Bezug auf ihren (momentan) gefühlten Stress, beantworten. Durch die Erfassung des empfundenen Stresses in alltäglichen Situationen können in Zukunft womöglich Stressursachen besser erkannt und im Sinne einer

[16] [41]
[17] [34]
[18] [21]
[19] [41]
[20] [9]

gesundheitsfördernden Idee, verringert werden. Weitere Komponenten des AYS Projekts werden bisweilen in weiteren Abschlussarbeiten der Universität Ulm entwickelt. Da sich das Projekt noch in der Entwicklungsphase befindet, sind momentan auch noch keine Apps in den gängigen Stores verfügbar.[21] [22]

7.3. MyKind

MyKind ist eine App, die zur Zeit im GooglePlay Store und im Apple AppStore erhältlich ist. Ziel war es eine Frau in ihrer Schwangerschaft, mit Hilfe dieser App zu begleiten und entsprechend zu beraten. Dabei soll die App nicht die medizinischen Vorsorgeuntersuchungen ersetzen, sondern vielmehr unterstützen. Vor allem in Bereichen für die, neben den oft rein medizinisch-körperlichen Untersuchungen, nicht immer genügend Zeit vorhanden ist. Deshalb wird mit Hilfe von wiederkehrenden Fragebögen auf die Veränderungen der Befindlichkeit, die Stimmungslage sowie auf das Umfeld der Schwangeren eingegangen. Durch Auswerten der erfassten Antworten zu den Fragebögen können mögliche Risikofaktoren aufgezeigt und bestenfalls neutralisiert werden. Weiter soll den Schwangeren hilfreiche Tipps für eine positive Entwicklung ihrer Schwangerschaft gegeben werden. MyKind ist in Kooperation der Universität Ulm und Universität Konstanz entstanden und wird, unter anderem, mit Hilfe weiterer Abschlussarbeiten, gerade an der Universität Ulm weiterentwickelt.[23] [24] [25] [26]

[21] [23]
[22] [35]
[23] [18]
[24] [20]
[25] [19]
[26] [17]

8. Fazit

Die erste Phase der Entwicklung des *Track Your Stress* Projektes ist zur Zeit dieser Arbeit abgeschlossen. Die Funktionalität der App ist so umgesetzt, dass die definierten Anforderungen an die App weitestgehend umgesetzt wurden. Ein User kann sich in der App registrieren, sich anmelden, Fragebögen seiner Studien betreffend ausfüllen und speichern/absenden, sowie zusätzlich selbst auswählen, welchen weiteren Studien er beitreten, oder welche beigetretenen Studien er wieder verlassen möchte. Das erste der folgenden Kapitel (8.1) fasst die Erkenntnisse dieser Arbeit zusammen. Das zweite und letzte Kapitel (8.2) behandelt mögliche Ideen für die App, um deren Funktionsweise noch weiter zu verbessern.

8.1. Zusammenfassung

Im Rahmen dieser Bachelor Arbeit ist eine mobile Android Anwendung zur Überwachung des Stresslevels entstanden. Die Anforderungen hierfür wurden in mehreren Meetings definiert. Während der Arbeit aufgetretene neue Anforderungen und Ideen wurden ebenfalls bestmöglich berücksichtigt und umgesetzt. Es wurde gezeigt, wie verschiedene Fragebögen mit unterschiedlichen Antwortmöglichkeiten (*Slider*, *RadioButtons*, …) visualisiert und beantwortet werden können. Dabei wurde gerade beim *Slider* darauf eingegangen, dass die Antwort möglichst unbeeinflusst gegeben wird. Erreicht wurde dies dadurch, dass die *SeekBar*[27] aus Android so angepasst wurde, dass kein initialer Wert angegeben wurde. Ebenso erhielt die App eine Funktionalität, um Hintergrundgeräusche beim Beantworten der Fragebögen aufzuzeichnen, womit evtl. Zusammenhänge von Stress und Lautstärke, z.B. im Forschungsbereich untersucht werden können. Weiter wurde auch erklärt wie es möglich ist, zusätzlich zur *Track Your Stress* Studie, parallel an weiteren Studien teilzunehmen bzw. teilgenommene Studien auch wieder zu verlassen.

8.2. Ausblick

Dieser Abschnitt beschreibt mögliche Ideen, die während der Entwicklung der App für das *Track Your Stress* Projekt aufgekommen sind. Ziel sollte es sein, die mobile Anwendung weiterhin zu verbessern. Infrage kommen dabei Verbesserungen am User-Inferface oder auch der Ausbau des *Offline-Modus*, zu einer, wie im *Online-Modus* möglichst vollständig nutzbaren Funktionalität der App. Weiter könnte man auch überlegen, ob der User die Benachrichtigungszeiten selbst einstellen kann, sofern das vom Fragebogen erlaubt ist.

8.2.1. Verbesserung am User-Interface und Hilfe

Generell standen im Rahmen dieser Bachelorarbeit die rein funktionalen Anforderungen im Vordergrund, wodurch verschiedene *GUI's* womöglich noch nicht ganz ausgereift sind und dadurch verbessert werden könnten, um die Bedienung und Interaktion des Users mit der App noch besser zu gestalten. So könnten z.B. die Pflichtfragen der verschiedenen Fragebögen von vornherein entsprechend markiert sein und nicht erst beim Absenden. Eventuell kann auch das Layout der Fragebögen an sich noch verbessert und ansehnlicher gestaltet werden. Weiter könnten auch Studien in der Anzeige aller Studien direkt entsprechend als privat oder offen markiert sein, womit dem User ein deutlicheres Bild über die verschiedenen Studien erschaffen wird. Nicht zu vergessen ist auch, dass es momentan keinerlei Hilfs- oder Infotexte gibt, was einem User bei eventuellen Verständnisproblemen sicherlich deutlich helfen würde.

[27] [36]

8.2.2. Verbesserter Offline-Modus

Der aktuelle *Offline-Modus* beinhaltet schlicht die Hauptfunktionen der App, d.h. das Beantworten von Fragebögen zu beigetretenen Studien ist auch ohne Internetverbindung möglich. Allerdings ist dies im *Offline-Modus* bisweilen nur von einem User bzw. vom zuletzt (online) angemeldeten User möglich, da immer dann die Fragebögen vom angemeldeten User geladen und auf der lokalen Datenbank gespeichert werden. Da in den meisten Fällen davon ausgegangen werden kann, dass heutzutage sowieso jeder ein eigenes Gerät besitzt, ist dies zu diesem Zeitpunkt noch nicht essentiell. Allerdings wäre es doch wünschenswert, wenn die App auch offline von mehreren Usern auf einem Gerät, mit möglichst allen Funktionen genutzt werden kann, um eine gewisse Unabhängigkeit bezüglich der Internetverbindung erreichen zu können.

8.2.3. Benachrichtigungen

Die App benachrichtigt den User erfolgreich über auszufüllende Fragebögen bezüglich seiner beigetretenen Studien, sofern diese Fragebögen auch Benachrichtigungen „versenden“. Der User kann die Zeiträume bzw. die Zeitpunkte der Benachrichtigungen der Fragebögen in der App einsehen, allerdings sind diese aktuell noch nicht veränderbar. Daher wäre eine mögliche Verbesserung, dem User eine Bearbeitung der Benachrichtigungen zu ermöglichen, um festlegen zu können, wann er Benachrichtigungen zu den entsprechenden Fragebögen erhalten will. Dadurch wird die App besser an jeden User angepasst. Im Gegenzug dazu stellt sich jedoch die Frage, ob man dem User aus wissenschaftlichen Gründen diese Funktion überhaupt ermöglichen sollte, da dieser dann Fragebögen, womöglich in seiner Freizeit oder in Ruhephasen ausfüllt, was zu einer eventuellen Verfälschung der Ergebnisse führen kann, gerade bei einer Überwachung des Stresslevels.

Literaturverzeichnis

[1] E. Budna. Stress als Krankeitsauslöser. Das Diathese-Stress-Modell bei Traumatisierungen in der Kindheit, Katholische Hochschule für Sozialwesen Berlin, 2017.

[2] „Die Techniker," 2016. [Online]. Available: https://www.tk.de/centaurus/servlet/contentblob/921466/Datei/3654/TK-Stressstudie_2016_PDF_barrierefrei.pdf. [Zugriff am September 2017].

[3] „Geist und Körper trainieren," *Galler Tagblatt,* 1999.

[4] „Gesetze im Internet," [Online]. Available: http://www.gesetze-im-internet.de/arbschg/__5.html. [Zugriff am 3 Oktober 2017].

[5] „healthindex.de," [Online]. Available: https://www.healthindex.de/krankheitsausloeser-stress.php. [Zugriff am 3 Oktober 2017].

[6] „Sprichwort-Plattform.org," [Online]. Available: http://www.sprichwort-plattform.org/sp/In%20einem%20gesunden%20Körper%20wohnt%20ein%20gesunder%20Geist#ref-In einem gesunden Körper wohnt ein gesunder Geist-8. [Zugriff am 3 Oktober 2017].

[7] „Vital.de," [Online]. Available: https://www.vital.de/wohlbefinden/entspannung/artikel/auswirkungen-stress. [Zugriff am 3 Oktober 2017].

[8] F. Markert. „minicles.de," 2015. [Online]. Available: http://www.minicles.de/8im-seele%20und%20koerper%20eudis.htm. [Zugriff am 3 Oktober 2017].

[9] „Bundesministerium für Gesundheit," 29 Februar 2016. [Online]. Available: http://www.bundesgesundheitsministerium.de/themen/praevention/praeventionsgesetz.html. [Zugriff am 2 September 2017].

[10] „developer.android.com," [Online]. Available: https://developer.android.com/reference/android/view/View.OnClickListener.html. [Zugriff am 21 Juli 2017].

[11] „developer.android.com," [Online]. Available: https://developer.android.com/reference/android/app/AlarmManager.html. [Zugriff am 25 September 2017].

[12] „developer.android.com," [Online]. Available: https://developer.android.com/index.html. [Zugriff am 5 Juli 2017].

[13] „developer.andoid.com," [Online]. Available: https://developer.android.com/reference/android/support/v7/app/AppCompatActivity.html. [Zugriff am 4 Juli 2017].

[14] „developer.android.com," [Online]. Available: https://developer.android.com/reference/android/widget/ArrayAdapter.html. [Zugriff am 21 Juli 2017].

[15] „developer.android.com," [Online]. Available: https://developer.android.com/reference/android/os/AsyncTask.html. [Zugriff am 28 Juli 2017].

[16] „duden.de," [Online]. Available: http://www.duden.de/rechtschreibung/Stress. [Zugriff am 24 September 2017].

[17] „GooglePlayStore," 3 November 2015. [Online]. Available: https://play.google.com/store/apps/details?id=de.uni_ulm.bachmeier.mykind. [Zugriff am 20 September 2017].

[18] „MyKind," [Online]. Available: https://www.mykind.info/about. [Zugriff am 20 September 2017].

[19] „MyKind," [Online]. Available: https://www.mykind.info/home. [Zugriff am 20 September 2017].

[20] „uni-ulm.de," [Online]. Available: https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/mykind/. [Zugriff am 20 September 2017].

[21] „uni-ulm.de," [Online]. Available: https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/trackyourtinnitus/. [Zugriff am 22 September 2017].

[22] T. Groll. „Zeit Online," 11 August 2014. [Online]. Available: http://www.zeit.de/karriere/2014-08/anti-stress-gesetz-chancen. [Zugriff am 28 Mai 2017].

[23] B. Klecina. „dbis.eprints.uni-ulm.de," 22 Juli 2016. [Online]. Available: http://dbis.eprints.uni-ulm.de/1395/1/MA_KL_2016.pdf. [Zugriff am 22 September 2017].

[24] „developer.android.com," [Online]. Available: https://developer.android.com/reference/java/io/BufferedReader.html. [Zugriff am 24 Juli 2017].

[25] „developer.android.com," [Online]. Available: https://developer.android.com/reference/java/io/BufferedOutputStream.html. [Zugriff am 25 Juli 2017].

[26] „developer.android.com," [Online]. Available: https://developer.android.com/reference/java/io/InputStreamReader.html. [Zugriff am 24 Juli 2017].

[27] „developer.android.com," [Online]. Available: https://developer.android.com/reference/android/app/AlertDialog.Builder.html. [Zugriff am 15 Juli 2017].

[28] R. Pryss, T. Probst, W. Schlee, J. Schobel, B. Langguth, P. Neff, M. Spiliopoulou und M. Reichert. „Mobile Crowdsensing for the Juxtaposition of Realtime Assessments and Retrospective Reporting for Neuropsychiatric Symptoms," in *30th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2017)*, I. C. S. Press, Hrsg., 2017.

[29] T. Probst, R. Pryss, B. Langguth und W. Schlee. „Emotional states as mediators between tinnitus loudness and tinnitus distress in daily life: Results from the TrackYourTinnitus application," *Scientific Reports,* 6 Februar 2016.

[30] R. Pryss, M. Reichert, B. Langguth und W. Schlee. „Mobile Crowd Sensing Services for Tinnitus Assessment, Therapy and Research," in *IEEE 4th International Conference on Mobile Services (MS 2015)*, I. C. S. Press, Hrsg., 2015, pp. 352-359.

[31] M. Ruf-Leuschner, N. Brunnemann, M. Schauer, R. Pryss, E. Barnewitz, M. Liebrecht, M. Reichert und T. Elbert. „Die KINDEX-App - ein Instrument zur Erfassung und unmittelbaren Auswertung von psychosozialen Belastungen bei Schwangeren in der täglichen Praxis bei Gynäkologinnen, Hebammen und in Frauenkliniken," *Verhaltenstherapie,* August 2016.

[32] M. Schickler, M. Reichert, R. Pryss, J. Schobel, W. Schlee und B. Langguth. Entwicklung mobiler Apps: Konzepte, Anwendungsbausteine und Werkzeuge im Business und E-Health, Bd. eXamen.press, S. Vieweg, Hrsg., 2015.

[33] I. Berres. „Spiegel Online," 30 Oktober 2013. [Online]. Available: http://www.spiegel.de/gesundheit/diagnose/so-gestresst-sind-die-deutschen-umfrage-der-techniker-krankenkasse-a-930696.html. [Zugriff am 28 Mai 2017].

[34] J. Herrmann. *Konzeption und technische Realisierung eines mobilen Frameworks zur Unterstützung tinnitusgeschädigter Patienten,* Universität Ulm, 2014.

[35] B. Klecina. *Assess Your Stress: Conceptual re-design of the TrackYourTinnitus system for measuring stress at the workplace,* Universität Ulm, 2016.

[36] „developer.android.com," [Online]. Available: https://developer.android.com/reference/android/widget/SeekBar.html. [Zugriff am 17 august 2017].

[37] T. Probst, R. Pryss, B. Langguth, M. Spiliopoulou, J. Schobel, M. Reichert und W. Schlee. „Does tinnitus depend on time-of-day? An ecological momentary assessment study with the TrackYourTinnitus application," *Frontiers in Aging Neuroscience,* Nr. 9, pp. 253-253, 2017.

[38] T. Probst, R. Pryss, B. Langguth, M. Spiliopoulou, M. Landgrebe, M. Vesala, S. Harrison, J. Schobel, M. Reichert, M. Stach und W. Schlee. „Outpatient Tinnitus Clinic, Self-Help Web Platform, or Mobile Application to Recruit Tinnitus Study Samples?," *Frontiers in Aging Neuroscience,* Nr. 9, pp. 113-113, April 2017.

[39] R. Pryss, M. Reichert, J. Herrmann, B. Langguth und W. Schlee. „Mobile Crowd Sensing in Clinical and Psychological Trials - A Case Study," in *28th IEEE Int'l Symposium on Computer-Based Medical Systems*, I. C. S. Press, Hrsg., 2015, pp. 23-24.

[40] W. Schlee, P. Pryss, T. Probst, J. Schobel, A. Bachmeier, M. Reichert und B. Langguth. „Measuring the Moment-to-Moment Variability of Tinnitus: The TrackYourTinnitus Smart Phone App," *Frontiers in Aging Neuroscience,* Nr. 8, pp. 294-294, Dezember 2016.

[41] J. Herrmann. „dbis.eprints.uni-ulm.de," 19 März 2014. [Online]. Available: http://dbis.eprints.uni-ulm.de/1037/1/DA_Herrmann2014.pdf. [Zugriff am 4 September 2017].

[42] „developer.android.com," [Online]. Available: https://developer.android.com/reference/android/view/LayoutInflater.html. [Zugriff am 22 Juli 2017].

[43] R. Pryss, W. Schlee, B. Langguth und M. Reichert. „Mobile Crowdsensing Services for Tinnitus Assessment and Patient Feedback," in *6th IEEE International Conference on AI & Mobile Services (IEEE AIMS 2017)*, I. C. S. Press, Hrsg., 2017.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sinngemäße Übernahmen aus anderen Werken sind als solche kenntlich gemacht und mit genauer Quellenangabe (auch aus elektronischen Medien) versehen.

Ulm, den_____ _____

Julian Haug