ulm university universität ulm

29.10.2020

Konzeption und Realisierung eines Moduls für auditorische Stimulationen für eine multizentrische und multinationale mHealth-App für eine paneuropäische Tinnitus-Studie

Masterarbeit

im Studiengang Informatik

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie

Institut für Datenbanken und Informationssysteme

Verfasser: Julian Haug
Matrikelnummer: 843443
1.Prüfer: Prof. Dr. Manfred Reichert
2.Prüfer: Prof. Dr. Rüdiger Pryss
Betreuer: Carsten Vogel M.Sc.

Fassung 29. Oktober 2020

Kurzfassung

Pfeifende, klingelnde oder auch als rauschende beschriebene Töne, subjektiv oder nachweislich vom Patienten wahrgenommen, werden als Tinnitus bezeichnet. Tinnitus ist zu einer Volkskrankheit geworden. Wie viele Personen tatsächlich unter Tinnitus leiden, lässt sich nur schätzen. Betroffene leiden infolge eines Tinnitus unter Schlafstörungen oder erleiden weitere psychische Erkrankungen. Die Wissenschaft und Medizin arbeiten ständig daran Tinnitus-Patienten besser behandeln und unterstützen zu können. Bestenfalls sollen die Geräusche nicht mehr störend oder gar nicht mehr wahrgenommen werden und verschwinden. Am stärksten nehmen Betroffene ihren Tinnitus in ruhigen Umgebungen wahr. Deshalb ist eine Form der Therapie eine auditorische Stimulation. Hierbei hört sich ein Patient unterschiedliche Geräuschkulissen, Melodien und Klänge an. Neben einer Vielzahl an möglichen Effekten, soll diese Art der Therapie bei Tinnitus Patienten vor allem darauf abzielen eine Hörüberempfindlichkeit, bzw. das Hören eines Dauertons weitestgehend zu reduzieren.

Im Rahmen des europäischen *UNITI*-Projekts soll eine multizentrische und multinationale mHealth-Applikation entwickelt und realisiert werden, wobei ein Modul für eben jene auditorische Stimulation implementiert und integriert wird. Diese Arbeit zeigt, wie sich Personen bzw. Patienten selbstständig auf mobilen Android Geräten, innerhalb jener *UNITI*-Applikation, jederzeit auditiv stimulieren können. Ferner soll diese Arbeit dazu beitragen Betroffene mit ihrer Erkrankung zu unterstützen und zu einem besseren Wohlbefinden führen, indem die wahrgenommenen Geräusche reduziert werden. Zudem erhofft sich die Forschung im Bereich Tinnitus für die Zukunft, durch Feedbacks und Befragungen der Patienten, bessere Therapieverfahren und Heilungsmöglichkeiten zu erhalten. Die Gesamtfunktionalität dieser Android mHealth-Applikation und vor allem die Funktionalität des Moduls für auditorische Stimulation werden detailliert dargestellt, sowie aus technischer Sicht betrachtet. Weiter wird aufgezeigt, wie ein solches Projekt architektonisch zu realisieren ist. Zusätzlich werden in dieser Arbeit spezielle Implementierungsthemen hervorgehoben. Am Ende steht eine voll funktionsfähige mobile Applikation zur Unterstützung von Tinnitus Patienten. Vermutet wird, dass eine auditive Stimulation viele positive Effekte mit sich bringt, wodurch das Projekt in dieser Form auch auf andere Zwecke übertragen werden kann. Beispiele hierfür sind die Unterstützung bei psychischen Erkrankungen, Schlafstörungen, Hörproblemen oder die Verbesserung der Aufmerksamkeit und Konzentration, sowie eine Hilfe beim Entspannen.

Inhalt

Abbildungsverzeichnis

Listingverzeichnis

1. Einleitung

Dieses Kapitel definiert zuerst kurz den Begriff Tinnitus, bzw. die Erkrankung an sich, sowie mögliche Therapieverfahren, wie etwa eine auditive Stimulation. Danach wird erläutert, warum die mobile Applikation *UNITI* [1], bzw. das enthaltene Modul der auditorischen Stimulation, diesbezüglich von Bedeutung ist. Zuletzt wird ein kurzer Überblick über den Aufbau dieser Arbeit gegeben. Weitere implementierte Funktionalitäten der *UNITI*-Applikation werden detailliert in einer ähnlichen Arbeit betrachtet [2].

1.1. Tinnitus und Therapie (auditorische Simulation)

„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“ [3]. Eine weitere Definition beschreibt Tinnitus als *„subjektiv wahrgenommenes Rauschen, Klingeln oder Pfeifen in den Ohren“* [4]. In Deutschland leiden ca. drei Millionen an einem zeitweisen oder chronischen Tinnitus [5]. Trotz bedeutender Fortschritte der Forschung bleibt Tinnitus als Krankheit weiterhin ein medizinisches und wissenschaftliches Rätsel, welche beinahe 15% der Bevölkerung betrifft [6]. Die möglichen, bisher angenommenen Ursachen sind vielseitig und oftmals schwer zu bestimmen. Dazu gehören psychische Probleme, Stress, Lärm, ein Hörsturz aber auch Ohrerkrankungen, wie Entzündungen oder Morbus Menière, bis hin zu Fremdkörpern und übermäßigem Ohrenschmalz. Unterscheiden muss man zusätzlich zwischen subjektiv und objektiv wahrgenommenen Tönen. *„90 Prozent der Patienten, die unter einem Tinnitus leiden, beklagen einen subjektiven Tinnitus. Nur sie hören das Ohrgeräusch, das ihnen zu schaffen macht“* [7]. Von einem objektiven Tinnitus spricht man, wenn die Quelle des Geräusches messbar ist und nachgewiesen werden kann. Meist liegt eine Erkrankung des Innenohrs vor, allerdings kann dies auch durch Atemgeräusche oder fließendes Blut in den Arterien hervorgerufen werden. Millionen Personen hatten oder haben eine Tinnitus-Erkrankung. *„Während ein Teil der Menschen gut mit ihrem Tinnitus auskommt, fühlen sich jedoch 8 % der Bevölkerung in Industriestaaten durch ihr Ohrgeräusch im Alltagsleben belästigt oder weisen Schlafstörungen auf. Bei etwa 0,5 % hat das Ohrgeräusch den Stellenwert einer eigenständigen Krankheit*“ [8]. Dabei wird wiederrum zwischen einem akuten und einem chronischen Tinnitus differenziert. Während bei einem akuten Tinnitus, wie er eventuell nach einem Hörsturz oder Tauchunfall auftritt, oftmals organisch, mithilfe von Infusionen oder Injektionen, wie bei einer Intratympanalen Kortikoidtherapie[1], sehr gute Ergebnisse erzielt werden können, sind diese Maßnahmen bei einem fortgeschrittenen, chronischen Tinnitus meist nicht von Erfolg gekrönt. Ein Tinnitus wird als chronisch definiert, wenn die Beschwerden länger als drei Monate anhalten. Die auditorische Stimulation ist demnach ein Therapieverfahren für chronisch Leidende. Die Patienten werden hierfür aufgefordert, sich regelmäßig, für eine gewisse Dauer, Klänge, Geräusche und Melodien anzuhören. Ebenfalls soll das bewusste Hören gefördert werden. Die Fähigkeit einzelne Instrumente einer Melodie oder dem Tinnitus-Geräusch ähnlich laute und frequente Töne erkennen zu können, soll aktiviert und gefestigt werden. Erstrebenswert ist eine Linderung der Beschwerden, sei es durch eine bewusstere Selbstwahrnehmung der Patienten, durch einfache Unterdrückung des Geräusches oder schlichtweg durch die Entspannung und den Stressabbau der beruhigenden Melodien und Klänge.

[1] *„Bei der Intratympanalen Kortikoidtherapie wird das Kortison durch eine Injektion in das Mittelohr verabreicht. Daher auch die Bezeichnung als „lokale Kortisontherapie“. Die Behandlung kann zusätzlich zur Infusionstherapie oder auch als gesonderte Therapieform erfolgen“* [9]. [20.10.2020]

1.2. Motivation

Dieses Projekt wird im Rahmen meiner Masterarbeit an der Universität Ulm und des Europa-Projekts *UNITI* entwickelt. Auf Grund der enormen Anzahl an Betroffenen, den unterschiedlichen Ursachen, sowie verschiedenen Therapieformen, ist es für Forschung und Medizin ein bedeutendes Thema. Steigende Anzahl an Krankentagen in Unternehmen und Organisationen, sowie steigende Zahlen an Erkrankungen psychisch bedingter Krankheiten, zeichnen ein eindeutiges Bild. Da eine Tinnitus-Erkrankung eine mögliche Ursache für psychische Probleme darstellt, unterstreicht dies die Wichtigkeit auf diese Erkrankung aufmerksam zu machen. Wünschenswert wäre es, wenn das Projekt *UNITI,* in Kombination mit der mobilen Applikation dazu beiträgt, die Krankheit besser erforschen und besser behandeln zu können. Da ich selbst bereits zwei Hörstürze mit leichten Tinnitus-Symptomen hatte, ist mir persönlich bewusst, wie schnell solche Probleme auftreten und wie belastend diese sein können.

Unter diesen Aspekten wird mit dieser Arbeit eine mobile Android Applikation entwickelt, die Tinnitus Patienten, unter anderem durch auditorische Stimulation, assistieren soll, besser mit ihrem Tinnitus umzugehen und eventuell sogar dabei unterstützt die Symptome zu reduzieren. Gleichermaßen wird durch Feedback und Befragung der Patienten die Forschung und Medizin vorangetrieben, die Erkrankung besser zu verstehen und diese dadurch effektiver behandeln zu können.

1.2.1. Vision

Im Idealfall ist das System der mobilen Applikation unkompliziert aufgebaut und benutzerfreundlich zu bedienen. Als optimales Ziel sollen mit dieser mobilen Applikation, Personen mit Tinnitus Symptomen dahingehend unterstützt werden, dass die störenden Geräusche nicht mehr wahrgenommen werden. Zusätzlich besteht die Hoffnung, die Krankheit hierdurch weiter zu erforschen und festzustellen, wie erfolgreich generell eine auditorische Stimulation als Therapiemaßnahme für Tinnitus-Patienten ist. Ferner soll durch das Feedback, bzw. durch das Bewertungssystem für die einzelnen *Sounds*, gezielt herausgearbeitet werden, wie effektiv, welche *Sounds,* subjektiv empfunden werden.

1.2.2. Projektkontext

Aufgrund der zahlreichen betroffenen Patienten, sowie um weiterführenden psychischen Erkrankungen vorzubeugen, besteht die Nachfrage an Instrumenten, mithilfe derer Patienten unterstützt und die Forschung dieser Krankheit vorangetrieben werden kann. Ähnlich den Absichten der *European School for Interdisciplinary Tinnitus Research (ESIT)*[2]*,* soll auch die *UNITI*-Applikation, durch erhobene Daten, dazu beitragen, neue Behandlungsansätze zu entwickeln, bestehende Behandlungsmöglichkeiten zu optimieren und den Horizont bezüglicher Risikofaktoren zu erweitern [10]. Weiter existiert der Wunsch möglichst vielen Personen zu helfen und eine breite Menge an verschiedenen, individuellen Symptomatiken zu erfassen. Aufgrund der Kombination mehrerer Tinnitus-Studien in der *UNITI*-Applikation, soll dieses Projekt einen Meilenstein in Bezug auf Forschung und Behandlung von Tinnitus-Erkrankungen repräsentieren. Da eine auditorische Stimulation eine Vielzahl an positiven Effekten mit sich bringt, ist diese mobile Applikation zusätzlich eine hervorragende Maßnahme eine solche Therapie immer und überall zu ermöglichen. Gerade im digitalen Zeitalter ist es mehr als sinnvoll eine mobile Applikation zu entwickeln, um Betroffene

[2] Webseite: https://esit.tinnitusresearch.net/ [28.10.2020]

unabhängig und jederzeit unterstützen zu können, sowie die Personen, auch in schwierigen Situationen, wie dem diesjährigen Lockdown, mit ihren Problemen nicht allein zu lassen.

1.3. Aufbau der Arbeit

Dieser Abschnitt stellt eine Übersicht der Struktur der Arbeit dar. Sie ist in acht Kapitel unterteilt. Nach der eben erfolgten Einleitung folgen zuerst ein kleiner Überblick und Vergleich bezüglich ähnlicher Projekte (Kapitel 2). Die Erläuterung der Anforderungen an die Applikation wird in Kapitel 3 angeführt. In Kapitel 4 erfolgt die Darstellung der Architektur des Projekts, wobei eine allgemeine Übersicht der Architektur, der Datenstruktur und des Aufbaus der mobilen Applikation gegeben wird. Im darauffolgenden Kapitel wird auf die Implementierung des Projektes eingegangen, sowie auf etwaige, spezielle Besonderheiten und Schwierigkeiten (Kapitel 5). Im 6. Kapitel wird das Projekt an sich vorgestellt und die Applikation beschrieben. Es wird darauf eingegangen, welche Funktionen dem Benutzer zur Verfügung stehen und wie er diese nutzen kann. Anschließend werden die gesetzten Anforderungen mit dem Stand der Entwicklungen abgeglichen (Kapitel 7). Schlussendlich wird im letzten Kapitel eine kurze Zusammenfassung, sowie ein Ausblick auf mögliche Weiterentwicklungen des Projekts gegeben (Kapitel 8).

2. Vergleichbare Projekte

Das folgende Kapitel befasst sich mit Projekten, die dem *UNITI*-Projekt ähnlich sind, unter anderem auch in Bezug auf die auditorische Stimulation. Die einzelnen Projekte werden kurz erläutert und etwaige Gemeinsamkeiten oder Unterschiede beleuchtet.

2.1. Track Your Tinnitus (TYT)

Das Projekt *Track Your Tinnitus* [11], ist ein Projekt zur Überwachung der Schwankungen einer individuellen Tinnituswahrnehmung. Es existiert bereits seit 2014 und dient dazu, die Forschung, sowie die Behandlung betroffener Patienten dieser Krankheit, die in schwerwiegenden Fällen sogar bis zum Suizid führen kann, zu unterstützen. 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 soll sicherstellen, dass der Nutzer seine persönliche Tinnituswahrnehmung, zu unterschiedlichen Zeiten, mithilfe der verfügbaren mobilen Applikationen dokumentiert. Bisher wurden Schwankungen der Tinnituswahrnehmung einzelner Patienten mit Tinnitus-Tagebüchern festgehalten, womit eine Erfassung des exakten zeitlichen Verlaufs nicht wirklich möglich ist. Die mobilen Applikationen haben den Vorteil, dass zum einen eine möglichst exakte, detaillierte Aufzeichnung des zeitlichen Verlaufs der wahrgenommenen Lautstärke des Tinnitus ermöglicht wird. Zum anderen besteht der Vorteil, 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 Applikationen sind derzeit im *GooglePlay*[3] Store und im Apple *AppStore* erhältlich. Die *UNITI*-Applikation bietet, zusätzlich zu einer auditiven Stimulation, ebenfalls ein Modul zur Befragung der aktuellen Tinnitus-Beschwerden, welches mit jenem des *Track Your Tinnitus* Projektes vergleichbar ist. Detailliert erläutert ist dieses Modul unter anderem in einer analogen Arbeit [2]. [3; 11–20]

2.2. Assess Your Stress (AYS)

Das *Assess Your Stress* Projekt [21] baut auf dem *Track Your Tinnitus* Projekt auf, wobei jenes grundlegend überarbeitet und neu aufgebaut werden sollte. Ein Grund für die Entwicklung ist das Präventionsgesetz [22] des Bundestags. Ein Gesetz, welches Prävention von Krankheiten, Gesundheitsförderung, sowie Früherkennung von Krankheiten verbessern soll. Unter anderem sollen gerade Unternehmen dazu bewegt werden, 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. Das AYS Projekt ist somit auch Auslöser für das, in Kapitel 2.3 beschriebene Projekt *Track Your Stress*, welches einen ersten Schritt zur Erfassung des Stresslevels auf Android Geräten darstellt. Dabei sollen Benutzer der Applikation kontinuierlich wiederkehrende Fragebögen, in Bezug auf ihren aktuell empfundenen Stress, beantworten. Durch die Erfassung des empfundenen Stresses in alltäglichen Situationen, sollen

[3] Google Play Store: https://play.google.com/store/apps/details?id=com.jochenherrmann.trackyourtinnitus&hl=de&gl=US [05.10.2020]

in Zukunft Stressursachen besser erkannt und im Sinne einer gesundheitsfördernden Idee, verringert werden. Weitere Komponenten des AYS Projekts wurden bisweilen in weiteren Abschlussarbeiten der Universität Ulm entwickelt. Da sich das Projekt noch in der Entwicklungsphase befindet, sind momentan noch keine Apps in den bekannten Stores verfügbar. [11; 23; 24]

2.3. TrackYourStress

Laut dem deutschen Duden wird Stress als eine „*erhöhte Beanspruchung, Belastung physischer oder psychischer Art*“ beschrieben [25]. Unter dauerhaftem Stress lässt die Leistung nach und im schlimmsten Fall schadet jener der Gesundheit. Dies kann so weit führen, dass Betroffene Burnout-Symptome oder Ähnliches durchleiden müssen. 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 empfinden ca. sechs von zehn Personen ihr Leben als stressig und jeder Fünfte steht unter andauerndem Druck [26]. Um der Entwicklung eines Anstiegs der Zahlen an Krankentagen und psychischen Erkrankungen entgegenzuwirken, 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 [22]. 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“* [24]. Da übermäßiger Stress als Ursache von psychischen Erkrankungen vermutet wird oder einen großen Teil der Ursache ausmacht, ist es für die Unternehmen von großer Bedeutung Stressursachen auf den Grund zu gehen.

In Folge von *Assess Your Stress* wurde daher das Projekt *Track Your Stress* in Kooperation mehrerer Universitäten, unter anderem durch die Universität Ulm und die Universität Regensburg, entwickelt. Um solchen stressbedingten Erkrankungen vorzubeugen oder auch um eine genaue Gefährdungsanalyse durchzuführen, war es zuallererst nötig das Stresslevel von Arbeitnehmern, bzw. Personen unter alltäglichen Bedingungen und Gegebenheiten festzustellen und zu untersuchen. Das Projekt *Track Your Stress* soll ein erster Schritt in diese Richtung sein, um das Stresslevel eines jeden Einzelnen zu bestimmen.

Hierzu wurde ähnlich wie in dieser Arbeit eine mobile Android Applikation, allerdings zur Messung des Stresslevels erstellt. Das Stresslevel von Nutzern soll, durch kontinuierliches, jedoch zeitlich zufälliges Beantworten von Fragen und Tests, erfasst werden. Die Applikation ist ein Versuch, den Stresslevel von Personen unter alltäglichen Bedingungen zu bestimmen, zu messen und zu untersuchen. Darüber hinaus soll die Applikation eventuelle Ursachen und Hintergründe eines hohen Stresslevels, anhand der beantworteten Fragebögen und den erhobenen Daten, identifizieren, um diese in Zukunft möglichst zu verringern oder vollständig zu egalisieren.

Technisch und strukturell ähnelt diese mobile Applikation der *UNITI*-Applikation sehr. Ein bedeutender Grund hierfür ist die gleiche Struktur der *API*. Auch hier werden Fragebogen vom Server geliefert, die der Nutzer beantworten und wieder zurück an den Server senden kann. Weitere Studien könnten theoretisch hinzugefügt werden, dies ist bisher jedoch nicht vorgesehen. Ebenso werden Nutzer über Benachrichtigungen dazu aufgefordert erneut mit der Applikation zu interagieren. Nutzbar ist die *TrackYourStress* Applikation, genau wie die des *UNITI*-Projekts, offline wie online. Ähnlich der *UNITI*-

Applikation soll die Applikation *TrackYourStress* möglichst bald in den bekannten Stores von Apple und Google verfügbar sein.

2.4. Tinnitracks

Das Projekt bzw. die gleichnamige mobile Applikation *Tinnitracks*[4] bietet generell zwei verschiedene Therapieverfahren an. Zum einen eine *Basis-Therapie*[5] und zum anderen eine *Neuro-Therapie*[6] [27]. Die Applikation ist ein Angebot der *Sonormed GmbH*[7], wobei die Therapie laut Internetauftritt von einem HNO-Arzt verordnet wird und die Kosten von den meisten Krankenkassen übernommen werden [27].

Die *Basis-Therapie* beruht generell auf dem Prinzip des *Counselling*. „*Im engeren Sinne wird unter dem Counselling die ärztliche Beratung des Tinnituspatienten verstanden*“ [28]. Hier hat der Patient, mittels der mobilen Applikation, die Möglichkeit einer ständigen Beratung bezüglich seiner Tinnitus-Erkrankung. Die Therapie sei ein „*umfangreiches Lern- und Übungsprogramm*“[8] [27] und soll so das Leben mit Tinnitus erleichtern. Durch diese Beratung soll dem Patienten der Umgang und die Akzeptanz des störenden Tons im Ohr erleichtert werden – bestenfalls soweit, dass dieser vom Patienten als nicht mehr störend wahrgenommen wird. Dadurch soll eine „*Höhere Lebensqualität durch Gelassenheit im Umgang mit Tinnitus*“[5] [27] erreicht werden. Das Anwendungsgebiet ist die subjektive Empfindung eines Tinnitus und die Anwendungsdauer beläuft sich auf circa sechs Wochen. Vergleichbar ist diese Art der Therapie mit dem Edukations-Modul der *UNITI*-Applikation, in welcher die Nutzer ebenfalls durch mehrere Kapitel und Lektionen geführt werden. Auch hier steht die Erlernung von eventuellen neuen Techniken zum Umgang mit der Erkrankung im Vordergrund [2]. Der große Vorteil des Moduls der *UNITI*-Applikation besteht darin, dass die *UNITI*-Applikation kostenfrei und ohne ärztliche Verordnung genutzt werden kann. Zusätzlich haben die Nutzer die Möglichkeit ihr Erlerntes über verfügbare Quizze zu überprüfen.

Die *neuro-Therapie „basiert auf dem Tailor-Made Notched Music Training (TMNMT), dessen Wirksamkeit […] placebokontrolliert belegt wurde und die statt lediglich Symptome zu verwalten, die Ursachen des Tinnitus adressiert*“[8] [27].

Die Funktionsweise des *TMNMT* basiert, ebenfalls wie das Modul der *UNITI*-Applikation, auf einer auditorischen Stimulation. Allerdings bekommt der Patient hierbei Musik zu hören „*die im Bereich seiner individuellen Tinnitus-Frequenz so gefiltert ist, dass sie keine Signalanteile mehr enthält*“ [29]. *Tinnitracks* benutzt hierzu die eigene Musik des Nutzers auf dem Gerät. „*Die Musik wird dabei von der Tinnitracks Neuro-Therapie speziell für die Tinnitus-Frequenz des Patienten auf Eignung geprüft, gefiltert und optimiert*“[8] [27]. Laut Angabe soll das Therapieverfahren für einen, subjektiven, tonalen[9], chronischen Tinnitus angewendet werden können, wobei eine stabile Frequenz gegeben und der Hörverlust „*geringer als 60dB HL*“[6] [27] sein sollte. Die Dauer der Anwendung wird mit ca. 12 Monaten à jeweils ca. 90 Minuten pro Tag angegeben[6] [27]. Die Neuro-Therapie ist also eine „*ursächliche Therapie zur Senkung der empfundenen Tinnitus-Laustärke*“[6] [27]. Sie soll eine „*angenehme und kurzweilige*“[8] Therapie darstellen und dazu beitragen die Lautstärke, durch die (Nicht-)Stimulierung,

[4] Google Play Store: https://play.google.com/store/apps/details?id=de.sonormed.tinnitracks&hl=de [07.10.2020]

[5] Basis-Therapie: https://www.tinnitracks.com/de/basistherapie [07.10.2020]

[6] Neuro-Therapie: https://www.tinnitracks.com/de/neurotherapie [07.10.2020]

[7] Webseite: https://www.sonormed.com/de/ [19.10.2020]

[8] Tinnitracks-Fachinformationen: https://www.tinnitracks.com/de/fachinformationen [07.10.2020]

[9] „So lassen sich die Geräusche in vielen Fällen eindeutig als Ton beschreiben, ähnlich einem Piepen oder Pfeifen. Ist eine solche Zuordnung ohne Weiteres möglich, spricht man von einem tonalen Tinnitus.“ [27]
Tinnitus-Symptome: https://www.tinnitracks.com/de/tinnitus/symptome [19.10.2020]

nachhaltig zu minimieren [27]. Vergleichbar ist die *Tinnitracks* neuro-Therapie mit dem Modul der auditorischen Stimulation der *UNITI*-Applikation. Vorteil des *UNITI*-Moduls ist wiederrum die Möglichkeit der unentgeltlichen und freien Nutzung. Allerdings ist es dem Nutzer der *UNITI*-Applikation bisher nur möglich eine bestimmte Anzahl an mitgelieferten Sounds zu hören. Zum Zeitpunkt der Erstellung dieser Arbeit ist es dem *UNITI*-Nutzer weder möglich seine eigene Musik anzuhören, noch ist es möglich, dass diese auf bestimmte Frequenzen gefiltert und optimiert werden. Allerdings besteht für den Nutzer des Moduls der auditorischen Stimulation die Möglichkeit, die einzelnen Sounds, bezüglich einer Verbesserung seines Empfindens, zu bewerten, sowie diese entsprechend als Favoriten zu sortieren. Ein weiterer Vorteil der *UNITI*-Applikation gegenüber *Tinnitracks* ist die Möglichkeit der vollständigen anonymen Nutzung, bzw. die Möglichkeit einer Registrierung ohne Angabe persönlicher Daten.

Beide *Tinnitracks* Therapien werden dabei als Medizinprodukte bezeichnet. Dem steht die *UNITI*-Applikation nichts nach, da diese im weiteren Verlauf die Medizinproduktegesetz-Prüfung durchlaufen wird. Der Ablauf der *Tinnitracks* Therapien ist bei beiden generell derselbe. Nachdem ein Tinnitus von einem HNO-Arzt diagnostiziert wurde, bekommt der Patient, bei einer Eignung für eine der Therapien, einen Aktivierungsscode. Mit diesem verifiziert sich der Patient über die *Tinnitracks*-Webseite. Danach kann sich der Nutzer in der mobilen Applikation mit seinem Nutzerkonto anmelden und wird anschließend durch die Therapie geführt[4].

2.5. Phonak Tinnitus Balance

Die Phonak Tinnitus Balance[10] Applikation ist Teil einer Klangtherapie *„und ermöglicht die Auswahl von Klängen aus einer Standardliste oder die Kombination von Klängen mit Ihrer persönlichen Smartphone-Musikbibliothek“*[11] [30]. Da bei *„mehr als 50 Prozent der Menschen mit einem Tinnitus zugleich auch ein*[…] *Hörverlust“*[12] [31] einhergeht, ist diese mit sämtlichen Phonak (Tinnitus-)Hörgeräten der *Sonova Deutschland GmbH*[13] kompatibel. Tinnitus-Hörgeräte sind bereits mit einem sogenannten Tinnitus-Noiser ausgestattet. Ein Noiser verfolgt ebenfalls das Prinzip einer Klangtherapie, indem, von den Patienten als angenehm empfundene, Klänge ins Ohr übertragen werden und somit vom ansonsten wahrgenommenen Tinnitus-Ton ablenkt[14] [31]. Zusätzlich wird zu diesem Tinnitus-Noiser als weiterer Baustein die Applikation *Tinnitus Balance* angeboten. Mittels des *ComPilot*[15] [30], einem universellen Streamer, ist es dem Nutzer möglich, sämtliche Sounds direkt über ihr Hörgerät anzuhören. Nichtsdestotrotz soll auch eine Nutzung ohne Hörgerät möglich sein, gerade für Patienten auf die ein *„nicht-versorgbarer Hörverlust“*[11] [30] zutrifft. Ähnlich wie in der auditorischen Stimulation der *UNITI*-Applikation ist es dem Nutzer der Tinnitus Balance App möglich die einzelnen Klänge, bezüglich ihrer Effektivität, zu bewerten. Im Falle der *Tinnitus Balance*-Applikation dient das dazu den individuellen Klangplan zu optimieren[10], wobei die *UNITI*-Applikation neben der Unterstützung der Patienten auch dem Forschungszweck dienen soll. Aus diesem Grund unterscheiden sich die mobilen Applikationen auch in Hinsicht der „Dringlichkeit“ der Bewertung. Im Gegensatz zur *UNITI*-Applikation, bei welcher der Nutzer erst nach dem Beenden eines Sounds zu einer Bewertung aufgefordert wird, ist es in *Tinnitus Balance* möglich die Sounds jederzeit, auch parallel zur Wiedergabe, zu validieren. Entscheidender Unterschied hierbei ist, dass *UNITI* den Nutzer offensiv darauf aufmerksam macht eine

[10] Google Play Store: https://play.google.com/store/apps/details?id=com.phonak.tinnitus&hl=de&gl=US [08.10.2020]

[11] Tinnitus-Portfolio: https://www.phonak.com/de/de/hoergeraete/tinnitus-behandlung/tinnitus-hoergeraete-und-loesungen-portfolio.html [08.10.2020]

[12] Audibene Tinnitus: https://www.audibene.de/tinnitus/ [08.10.2020]

[13] Webseite: https://www.sonova.com/germany/de-de [21.10.2020]

[14] Audibene Tinnitus-Noiser: https://www.audibene.de/tinnitus/noiser/ [08.10.2020]

[15] ComPilot: https://www.phonak.com/de/de/hoergeraete/zubehoer/phonak-com-pilot-II.html [08.10.2020]

Validierung abzugeben, während dies bei *Tinnitus Balance* nahezu beiläufig geschieht. *Tinnitus Balance* ermöglicht es Nutzern die Sounds in einzelnen Listen zu sortieren oder neue Listen für spezielle Momente anzulegen, wohingegen es dem Nutzer des Moduls der *UNITI*-Applikation möglich ist die einzelnen Sounds in der Liste zu favorisieren oder nicht. Ansonsten besitzen beide Applikationen eine Timer- bzw. Looping-Funktion und sind weitestgehend anonym, ohne manuelles Anlegen eines Nutzerkontos, benutzbar. Zusammenfassend lässt sich sagen, dass sich beide Apps ähneln und kostenlos in den Stores erhältlich sind.

2.6. Tinnitus Help

Tinnitus Help[16] ist eine weitere Applikation, die versucht Tinnitus-Patienten, basierend auf einer Klangtherapie, zu unterstützen. Im *GooglePlay* Store wird diese wie folgt beworben:

> *Tinnitus Help – 8 Programme in Einem. Die ultimative Anwendung für Tinnitus-Betroffene. Nur natürliche Klänge – kein Syntheziser! Entwickelt von einem Softwarespezialisten für Frequenz-Analyse und einer führenden Musikpsychologin, Musiktherapeutin und Hörtherapeutin mit mehr als 20 Jahren Erfahrung in der der Therapie mit Tinnitus-Patienten. Bereits mit großem Erfolg in der Praxis angewendet.*[16]

Im Gegensatz zu *Tinnitracks* soll hier die individuelle Tinnitus-Frequenz selbst ermittelt und am Gerät über einen Schieberegler angegeben werden[17] [32]. Dies ist laut Store Auftritt bis zu einer Frequenz von 20.000Hz möglich. Dieses Vorgehen ist für den Nutzer einerseits komfortabel, birgt aber auch Kritikpunkte: „*Hier wird der Tinnitus vom Patienten selbst bestimmt, was angeblich bis 20 kHz möglich sein soll, allerdings per Internet oder Smartphone schlicht nicht denkbar ist*“ [33]. Die zugehörige Webseite zur Applikation weist zudem darauf hin, dass *Tinnitus Help „nicht der ‚primären medizinischen Zweckbestimmung', sondern der Entspannung“*[18] [32] diene. Sie verfolgt das Prinzip der *Tinnituszentrierten Musiktherapie (TIM)*[19], d.h. die Applikation „*ist ein neurophysiologisches Hör- und Entspannungstraining, das auf der Grundlage der Tinnituszentrierten Musiktherapien (TIM) nach Dr. Annette Cramer*[20] *entwickelt wurde“*[18] [32]. Die Applikation ersetze keine Therapie, sondern soll vielmehr als ergänzendes Mittel der *TIM* und für ein individualisiertes Selbsttraining zum Umgang mit der eigenen Tinnitus-Erkrankung betrachtet werden. Die erste der beworbenen acht Funktionen war die Selbstbestimmung der Tinnitus-Frequenz. In einer weiteren Funktion kann der Nutzer ein Geräusch auswählen, das zu seinem Tinnitus-Profil addiert wird. Dadurch entsteht eine Art Noiser der die Wahrnehmung des Tinnitus-Tons verringern soll. Zu diesem Tinnitus-Profil kann der Nutzer nun verschiedene Musikstücke hinzumischen, welche ähnlich wie bei *UNITI* mit der Installation der Applikation ausgeliefert werden, um den Tinnitus weiter zu überdecken. Dadurch soll, mithilfe dieser kreierten Mischung, jene Hirnareale stimuliert werden, in welchen, nach dem Stand der Forschung, die Tinnitus-Störung entsteht und somit für eine Linderung der Symptome und Entspannung des Patienten sorgen. Eine Recherche über eine genauere, technische Beschreibung der Applikation war ebenso wenig möglich, wie die Ausführung einer persönlichen Testung, da die Applikation wiederrum nicht kostenlos verfügbar war. Im Endeffekt handelt es sich um eine auditorische Stimulation, ähnlich dem Modul der *UNITI*-Applikation. Allerdings ist die *UNITI*-Applikation kostenlos und verbraucht deutlich weniger Speicherplatz. Dies liegt womöglich an der größeren Mediathek an Sounddateien, welche

[16] Google Play Store: https://play.google.com/store/apps/details?id=com.embarcadero.TinnitusHelp&hl=de [08.10.2020]

[17] Interessanter Vergleich zu Tinnitracks: http://www.tinnitus-help.eu/wirkungsweise/abgrenzung-zu-tinnitracks-der-fa-sonormed-gmbh/index.php [08.10.2020]

[18] Webseite: http://www.tinnitus-help.eu/wirkungsweise/wirkungsweise/index.php [08.10.2020]

[19] TIM: https://www.musiktherapie.de/wp-content/uploads/2019/03/tinnitus-tim.pdf [08.10.2020]

[20] http://www.musiktherapeutikum.de/ [08.10.2020]

Tinnitus Help im Vergleich zur *UNITI*-Applikation ausliefert, was wiederrum eine Ähnlichkeit darstellt, da beide Applikationen die Sounds lokal auf dem Gerät halten.

2.7. Zusammenfassung der Projekte

Zusammenfassend lässt sich sagen, dass mittlerweile eine enorme Anzahl an Projekten vorhanden ist, die dem *UNITI*-Projekt, bzw. dessen Applikation ähneln, wenngleich die Applikationen verschiedene Vorgehensweisen präsentieren und die einzelnen Funktionalitäten unterschiedlich umgesetzt wurden. Mit Ausnahme von *Assess Your Stress* und *Track Your Stress* befassen sich alle, in diesem Kapitel vorgestellten Projekte, mit dem Thema Tinnitus. Nichtsdestotrotz verfolgen diese unterschiedliche Therapieansätze, mitunter sogar in ein und derselben Applikation. Die *UNITI*-Applikation ist daher eine sinnvolle und vor allem kostenfreie Alternative, da versucht wird mehrere Therapien, in Form verschiedener Studien, in einer Applikation zu vereinen.

Ein interessanter Einblick in die Applikationen und deren tatsächlichen Wirkungen, einschließlich kritischer Betrachtungsweisen, auch in Hinblick auf die Applikation *Tinnitracks*, ist auf der Webseite der deutschen Tinnitus-Liga nachzulesen [33].

3. Anforderungen

Dieses Kapitel definiert die Anforderungen an die *UNITI*-Applikation und das darin enthaltene Modul der auditorischen Stimulation. Dabei sind diese Anforderungen jeweils für die gesamte Applikation und das integrierte Modul, sowie in funktionale und nichtfunktionale Anforderungen unterteilt.

3.1. Funktionale Anforderungen

Nummer	Beschreibung	Problembeschreibung
***UNITI*-Applikation Gesamtprojekt**		
1	Registrierung in der Applikation	Für nicht (serverseitig) vorab angelegte Nutzer sollte es möglich sein, direkt auf einem Android Gerät ein Benutzerkonto zu erstellen. Dafür wird eine existierende E-Mail-Adresse benötigt, welche der Nutzer, in Kombination mit einem Passwort, beim Registrierungsvorgang angibt. Die Verifizierung des Accounts soll durch eine automatisiert verschickte E-Mail erfolgen.
2	Anonyme Registrierung	Es soll möglich sein, die Applikation anonym zu starten und zu nutzen. Dabei wird für den Benutzer je eine Zufalls-ID als Benutzername bzw. E-Mail-Adresse, sowie eine Zufalls-ID als Zufalls-Passwort generiert und dem Nutzer zugeteilt. Infolgedessen erfolgt ein automatischer Login in die Applikation.
3	Login registrierte Nutzer	Benutzer können sich mit den, zuvor bei der Registrierung, angegebenen Daten für E-Mail-Adresse und Passwort, nach erfolgreicher Verifizierung, direkt am Gerät anmelden.
4	Login angelegte Nutzer	Benutzer bekommen einen Benutzernamen und ein Passwort zugeteilt (per PDF etc.). Mit diesen Daten kann sich der Nutzer nun am Gerät in die Applikation einloggen.
5	Automatischer Re-Login	Hat sich ein Benutzer einmal erfolgreich in der Applikation angemeldet, bleibt er dauerhaft angemeldet, bzw. es erfolgt beim Start der Applikation eine erneute automatische Anmeldung mit den zuvor eingeloggten Daten.
6	Passwort ändern	Benutzer können innerhalb der Applikation deren Passwort ändern, sofern eine aktive Internetverbindung besteht.
7	Passwort vergessen	Für selbständig registrierte Nutzer (E-Mail-Adresse + Passwort) ist es möglich ein neues Passwort per E-Mail zu erhalten, sollte es vergessen werden. Für vorab angelegte Nutzer soll dies über den Kundensupport möglich sein.
8	Profilinformationen	Der Benutzer soll seine Profilinformationen innerhalb der Applikation einsehen können. Ebenso besteht hier die Möglichkeit das

		Passwort zurückzusetzen und zu ändern. Weiter ist hier die Funktionalität des Logouts angesiedelt.
9	Logout	Es besteht für die Nutzer die Möglichkeit sich auszuloggen. Ein erneuter Login auf den vorherigen Account ist dann nur mit Benutzername und Passwort möglich! Für Anonyme Nutzer wird ein erneuter, manueller Login kompliziert, da der zugeteilte Nutzername sehr komplex aufgebaut ist.
10	Impressum	Der Nutzer kann hier sämtliche Informationen über das Projekt einsehen.
11	Seite aktualisieren	Über einen „refresh-Button“ soll es dem Nutzer möglich sein die aktuelle Seite neu zu laden.
12	Ergebnisse	Die Applikation soll eine Registerkarte für Feedbacks enthalten. Hier werden dem Nutzer sämtliche möglichen Rückmeldungen zu seinen Aktivitäten in der Applikation aufgelistet. Dazu gehören auch Feedbacks zu den ebenfalls integrierten Funktionalitäten zur Tinnitus-Befragung und eines Edukations-Moduls.
13	An Studien teilnehmen	Ein Benutzer sollte sich für verschiedene Studien einschreiben und dadurch an diesen teilnehmen können. Ebenfalls besteht für diejenigen Nutzer die Möglichkeit einzelne Studien wieder zu verlassen. Diese Funktionalitäten sind **nicht** verfügbar für vorab serverseitig angelegte Nutzer, ausgenommen ist reine Ansicht und Informationen zu den einzelnen Studien.
14	Menüstruktur (Navigation Bar)	Die Menüstruktur, die Bottom Navigation Bar, wird dynamisch erzeugt, bedingt durch die Studien, an welchen der Nutzer teilgenommen hat.
15	Navigation	Der Nutzer kann sich jederzeit mithilfe der Navigation Bar durch die einzelnen Module der Applikation bewegen (auditorische Stimulation, Edukations-Modul, Tinnitus-Fragebögen), sofern er an den entsprechenden Studien beigetreten ist.
16	Zurück Button (in der Applikation)	Lässt den Nutzer auf die vorherige Seite zurückkehren. Ist dann vorhanden, wenn keine Navigation Bar verfügbar ist.
17	Zurück Taste (Android Gerät)	Lässt den Nutzer auf die vorherige Seite zurückkehren. Bei zweimaligem Klicken soll die Applikation geschlossen werden.
18	Netzwerkunabhängigkeit	Eine funktionierende Internetverbindung auf dem Smartphone sollte keine Voraussetzung für das Benutzen der Applikation sein, da ein Benutzer evtl. nur schlechten oder gar keinen Empfang haben kann. Voraussetzung ist eine

		erfolgreiche erste Anmeldung mit funktionierender Internetverbindung.
19	Benachrichtigungen	Die Applikation benachrichtigt den Nutzer über neue Aktivitäten, die der Nutzer ausführen soll, sofern diese von den einzelnen Modulen der Applikation getriggert werden.
Modul für auditorische Stimulation		
20	Sounds visualisieren	Dem Nutzer soll eine komplette Liste an abspielbaren Sounds angezeigt werden.
21	Sortierung der Sounds	Sortiert sind diese Sounds in drei Gruppen: an erster Stelle erscheint ein vorgeschlagener Sound, der zweite Teil bildet die Favoriten des Nutzers ab und die restlichen Sounds sind in der dritten Gruppe gebündelt. Ebenfalls symbolhaft visualisiert wird, welchen Sound der Nutzer in der aktuellen Sitzung bereits angehört hat.
22	Interaktion mit einem Sound	Benutzer können durch einen Klick auf einen Sound der Liste den Audio-Player starten. Sobald der Player geöffnet ist, kann der Sound abgespielt werden.
23	Playerfunktionalitäten	Ist der Player geöffnet und der Sound wird abgespielt, hat der Nutzer mehrere Möglichkeiten: - Loop: Durch Klicken wird der RepeatDialog geöffnet. Der Nutzer kann auswählen, wie lange bzw. wie oft er den ausgewählten Sound wiedergeben lassen möchte. Zur Auswahl stehen zwei Minuten, fünf Minuten, zehn Minuten oder auf unbegrenzte Zeit. Wurde eine Zeitspanne ausgewählt, ist die Wiederholungsfunktion solange aktiviert, wie vom Nutzer zuvor gewählt. Die gewählte Zeitspanne wird entsprechend über das Symbol visualisiert. Wurde der Sound vor der Interaktion mit dem Loop-Button bereits abgespielt, wird die Wiedergabe pausiert. - Pause: durch Klicken wird das Abspielen des Sounds pausiert - Stopp: Durch Klicken wird die Wiedergabe beendet und der Bewertungsdialog geöffnet, sofern bereits eine Wiedergabe gestartet wurde. - Play: o Ist der Audio-Player geöffnet kann der Nutzer die Wiedergabe des Sounds durch Klicken starten.

		o Wurde die Wiedergabe zuvor pausiert, kann durch Klicken das Abspielen des Sounds fortgesetzt werden. - Schließen: Audio-Player wird geschlossen und Wiedergabe des Sounds beendet. Wurde zuvor eine Wiedergabe gestartet und noch keine Bewertung abgefragt, erscheint der Bewertungsdialog. Danach soll der Nutzer zur Liste aller Sounds zurückgeleitet werden. - Favoriten: Durch Klicken wird dieser Sound den Favoriten hinzugefügt, erneutes Klicken entfernt diesen wieder aus der Favoriten-Gruppe. Dies wird entsprechend durch ein Symbol visualisiert. Jede Interaktion mit dem Audio-Player soll in die „Key-Value“ Liste mit Zeitstempel oder Wert eingetragen werden, damit diese Daten später an den Server gesendet werden kann.
24	Bewertungsdialog	Wird die Wiedergabe des Sounds, gestoppt oder der Audio-Player geschlossen, erscheint ein Pop-Up mit einem Bewertungsdialog. Der Nutzer kann in diesem angeben wie sich der Sound auf seine Beschwerden ausgewirkt hat.
25	Synchronisierung der Ergebnisse	Hat der Nutzer einen oder mehrere Sounds wiedergegeben und bewertet, soll das Ergebnis eines jeden Sounds festgehalten und nach Beenden einer Sitzung gebündelt an den Server geschickt werden. Vervollständigt werden diese Daten durch Aufzeichnen des Abspielverhaltens eines jeden einzelnen, wiedergegebenen Sounds.

3.2. 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 die *UNITI*-Applikation und das integrierte Modul für die auditorische Stimulation.

Nummer	Beschreibung	Problembeschreibung
1	Design	Die Applikation und das Modul sollten freundlich und ansprechend sein, sodass sich die Nutzer von Anfang an wohl fühlt.
2	Benutzbarkeit	Die Applikation 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, sodass dieser reagieren kann.
3	Transparenz	Dem Nutzer sollte offen gelegt werden wofür diese Applikation entwickelt wurde, wofür die Daten genutzt, sowie wo und wann diese erhoben und gespeichert werden.
4	Medizinproduktegesetz	Die Applikation und die einzelnen Module durchlaufen die Prüfung für das Medizinproduktegesetz und werden nach Bestehen entsprechend verifiziert.
5	Mehrsprachigkeit	Da die Applikation weltweit verfügbar sein soll, muss eine Anpassung der dargestellten Sprache erfolgen.
6	Verfügbarkeit	Die Applikation sollte auf möglichst vielen Android Geräten verfügbar sein, um eine Vielzahl an Benutzern zu erreichen.

4. Architektur

Dieses Kapitel beschreibt die Architektur des Projekts. Zunächst wird eine kurze Übersicht der Gesamtarchitektur gegeben (Kapitel 4.1). Im Anschluss werden zwei typische Abläufe angeführt (Kapitel 4.2), gefolgt von der Beschreibung der Datenstruktur (Kapitel 4.3). Des Weiteren werden die Controller-, Manager-, und Handlerklassen, wie z.B. der *MainManager* kurz erläutert (Kapitel 4.4). Im vorletzten Kapitel (4.3.2) wird die lokale Datenbank betrachtet, welche eine Offline-Nutzung der Applikation ermöglicht. Am Ende dieses Kapitels werden noch die *Views* (*Activities*) der Applikation erklärt (Kapitel 4.5).

4.1. Architekturübersicht

Die *UNITI*-Applikation besteht generell aus zwei Komponenten: Server und mobiler Android Applikation. Der Server bzw. das gesamte Backend (*API* inkludiert) war bereits von Seiten der Universität Ulm und Würzburg gestellt. Die Applikation wurde nativ mit Java auf Android entwickelt. Abbildung 1 zeigt die Verknüpfungspunkte der einzelnen Komponenten. Ersichtlich ist, dass die Applikation nicht direkt auf die Daten des Servers zugreift, sondern jegliche Kommunikation zwischen Applikation und Server über eine *JSON-API*[21] läuft.

Die Applikation und die Module sind generell nach dem Model-View-Controller Prinzip aufgebaut. Eine Besonderheit stellt die Verbindung via *API-Connection-Tasks* zur *API* dar. Zum einen, weil in dieser Funktionalität keine View enthalten ist, zum anderen, da diese größtenteils bereits als Library implementiert und somit ausgegliedert wurde, um diese Funktionalität weiteren Projekten komfortabel zur Verfügung zu stellen. Die restlichen Verbindungen zum Server, vor allem jene die nicht sofort ausgeführt werden müssen oder aufgrund fehlender Internetverbindung nicht sofort ausgeführt werden können, werden als Services im Hintergrund getätigt. Dies ermöglicht ein erneutes Triggern, einer bestimmten Funktionalität, zu einem späteren Zeitpunkt, wie etwa den Sendevorgang von Antwortsätzen an die *API*.

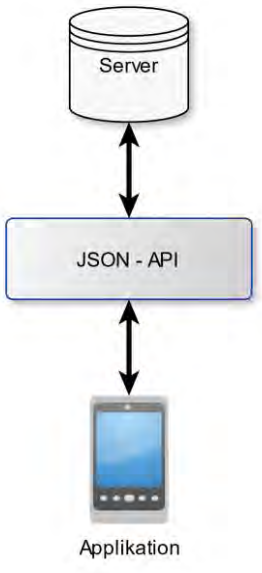


Abbildung 1: Grundsätzliche Komponenten

[21] Restful-Api: https://restfulapi.net/ [27.10.2020]

4.2. Genereller Ablauf

Die Benutzung der *UNITI*-Applikation setzt standardmäßig ein Benutzerkonto voraus. Es gibt hierfür drei Möglichkeiten ein solches Benutzerkonto zu erhalten:

1. Im Rahmen des Europaprojekts *UNITI* wird es vordefinierte Nutzer geben. Diese erhalten vom Administrator einen Benutzernamen sowie ein Passwort. Das heißt diese Benutzernamen sind vorab im System registriert und die Benutzer können sich mit den erhaltenen Daten direkt am Gerät einloggen.
2. Nutzer der Applikation können sich direkt am Gerät registrieren. Hierfür muss der Nutzer seine Daten, Benutzername, bzw. E-Mail-Adresse und Passwort, angeben. Nach einer erfolgreichen Verifizierung kann sich der Nutzer sofort in die Applikation einloggen. Die Aktivierung des Nutzerkontos erfolgt über einen Bestätigungs-Link in einer automatisiert zugestellten E-Mail.
3. Nutzer können die Applikation auch absolut anonym benutzen. Hierfür folgen sie dem Weg der anonymen Registrierung auf dem Startbildschirm. Dem Nutzer wird mit Hilfe zweier UIDs (unique identifier) ein Benutzername und ein Passwort generiert und zugeteilt. Dadurch wird der Nutzer automatisch eingeloggt und kann die Applikation sofort benutzen.

Ist ein Nutzer erfolgreich registriert kann sich der Nutzer in die Applikation einloggen. Im Benutzerbereich landet der Nutzer nach erfolgreichem Login zuallererst auf der Ansicht der verfügbaren Studien. Je nachdem an welchen Studien der Nutzer teilnimmt baut sich das Menü am unteren Rand dynamisch zusammen. Nun ist es möglich durch die gesamte Applikation zu navigieren, unter anderem auch zur auditorischen Stimulation. Innerhalb des Moduls für auditorische Simulation ist es dem Nutzer möglich die einzelnen Sounds abzuspielen, Sounds zu favorisieren und diese zu bewerten. Abbildung 2 zeigt den soeben beschriebenen, generellen Ablauf der Applikation und Abbildung 3 jenen Ablauf des Moduls der auditorischen Stimulation[22].

[22] Genauere Details und Visualisierung der Abläufe siehe in Kapitel 6

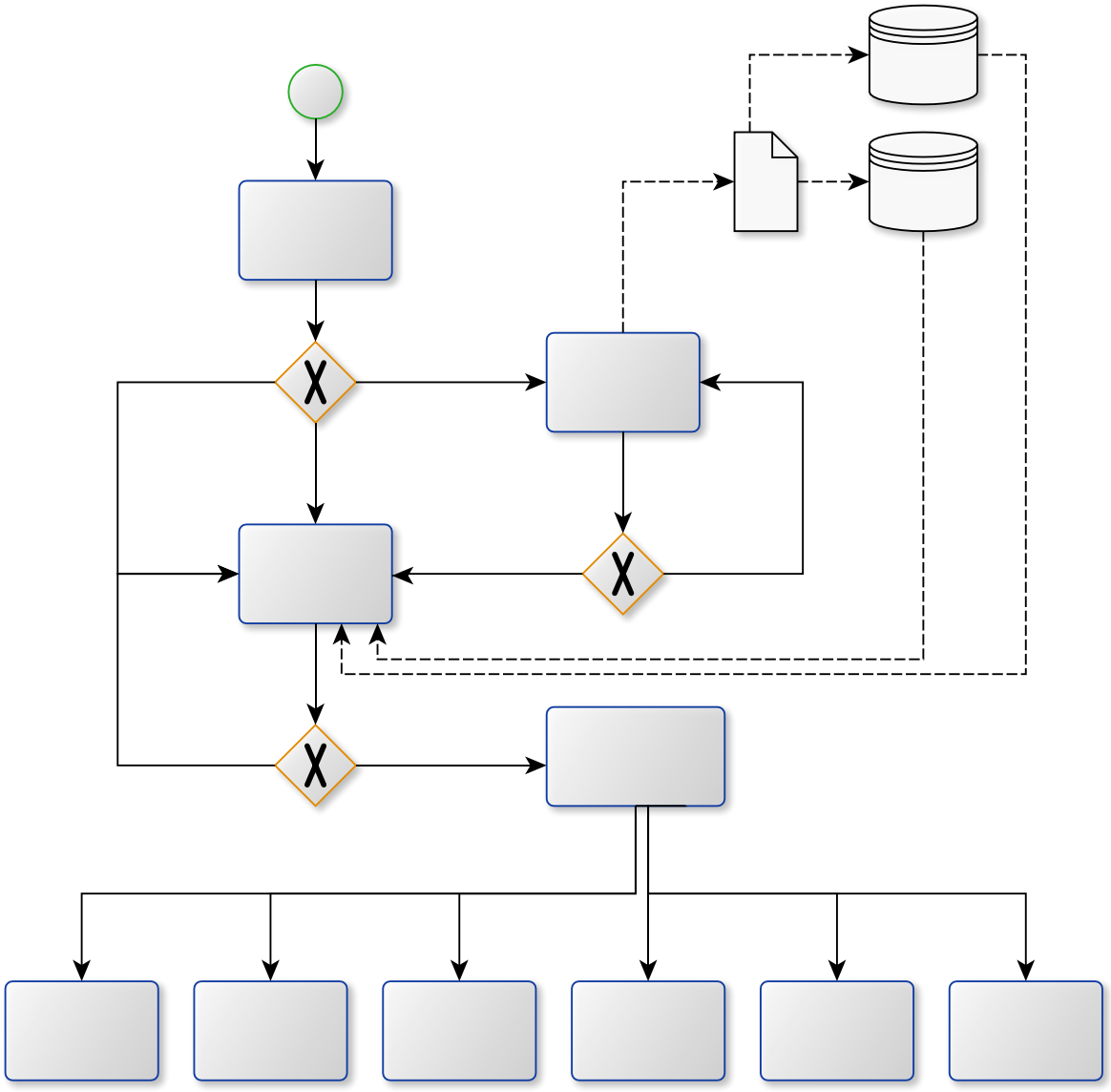


Abbildung 2: Prozessablaufdiagramm: Genereller Ablauf UNITI Gesamt-Applikation

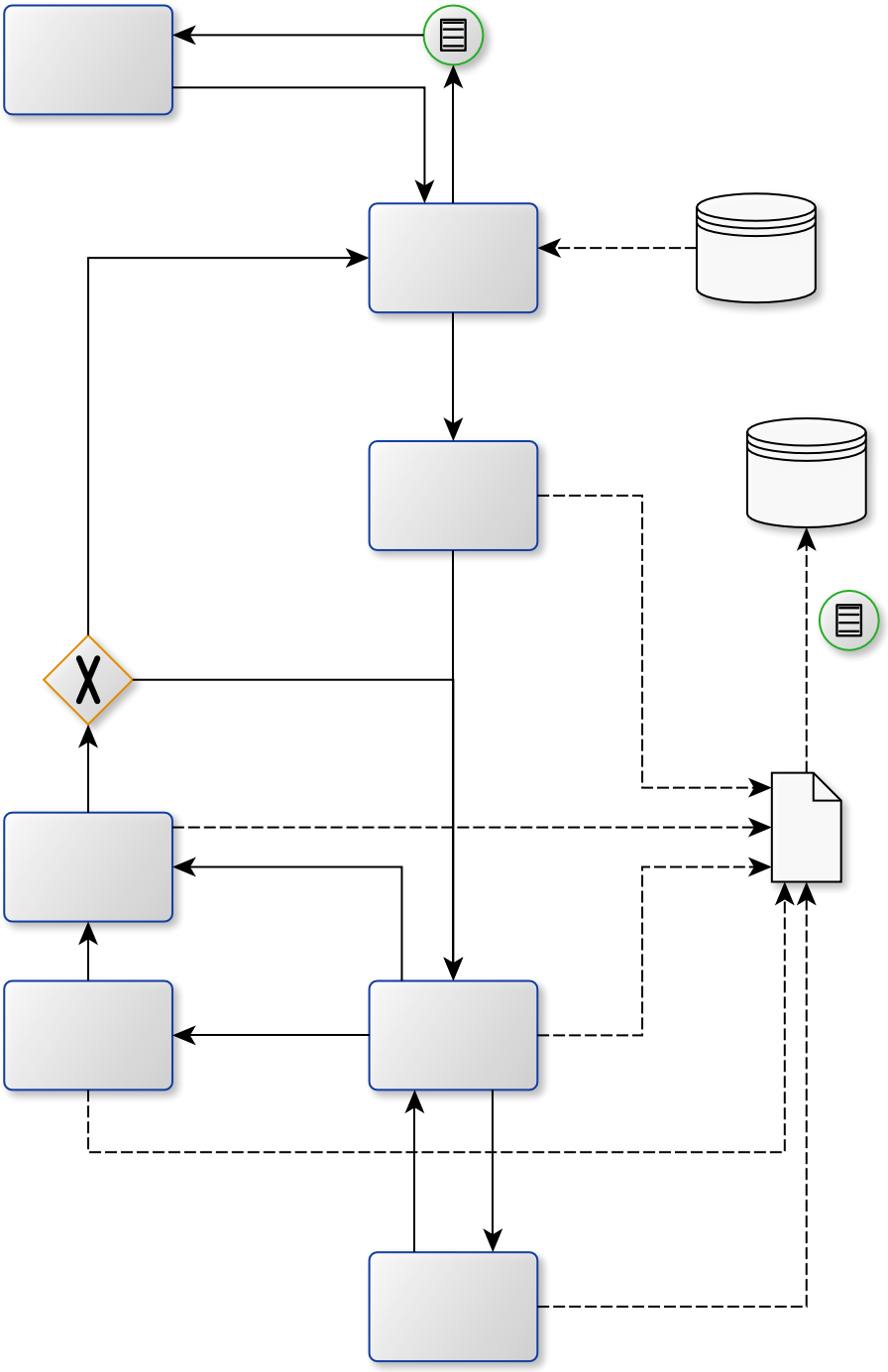


Abbildung 3: Prozessablaufdiagramm: Ablauf Modul auditorische Stimulation

Sofern eine Internetverbindung besteht, laufen die Datentransfers und Server-Verbindungen generell immer nach dem gleichen Schema ab. In Abbildung 4 wird dies anhand des *Logins* beispielhaft, strukturell dargestellt.

Durch eine entsprechende Aktion des Nutzers, hier am Beispiel des *Logins*, wird zuerst die *login*-Funktion im *MainManager* aufgerufen. Übergeben werden dabei die Parameter für Name und Passwort. In der *login*-Funktion des *MainManagers* wird eine neue Instanz der *ApiConnectionTask*-Klasse erzeugt und ein *AsyncTask*[23] [34] gestartet. Nach der Funktion *doInBackground()*, in welcher die eigentliche Kommunikation mit der *API* erfolgt, wird in der *onPostExecute()*-Funktion die *done()*-Methode des *MainManagers* angestoßen. Dort wird anhand des *doneMethod*-Parameters unterschieden, welche *CallbackFunktion*, hier die *loginCallback,* aufgerufen werden muss. Ebenfalls wird auch ein Status-Code, der für einen erfolgreichen oder fehlgeschlagenen Versuch steht, als Antwort mitgeliefert. Anhand der im Konstruktor übergebenen *Activity* im *MainManager,* kann auch nur zur richtigen *Activity* zurückgesprungen werden. Generell können mehrere *Tasks* parallel nebeneinander laufen. Durch den Pointer auf die entsprechende *Activity,* beim Anlegen des *MainManagers* im Konstruktor, ist immer klar zu welcher *Activity* das Ergebnis des *AsyncTasks,* über die *Callback*-Funktion, zurückkehren muss.

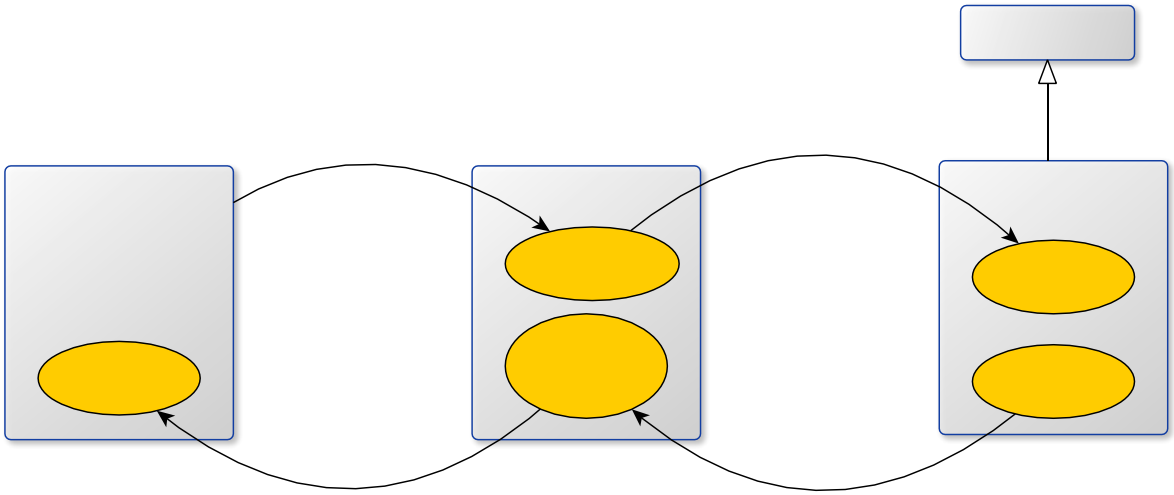


Abbildung 4: Programmablauf: Kommunikation von Activity- MainManager- ApiConnectionTask

[23] AsyncTask: https://developer.android.com/reference/android/os/AsyncTask [25.09.2020]

Wird etwas über die *API* geladen oder gesendet und sind womöglich mehrere *AsyncTasks* parallel gestartet, wird dies dem Nutzer durch entsprechende Dialogfenster ebenso mitgeteilt, wie erfolgreiche Lade- bzw. Sendevorgänge oder auch gescheiterte Kommunikationen. Dem Nutzer wird dies ähnlich visualisiert, wie es in Abbildung 5 auf dem Smartphone zu sehen ist. Ersichtlich wird auch, dass mehrere *AsyncTasks* echt parallel ablaufen können. Der *LoadingDialog* bleibt hier, beim Laden der Fragebögen und der zugehörigen Fragen, solange bestehen, bis im *MainManager* alle gestarteten *AsyncTasks* auch wieder zurückgekommen sind. Erst dann erfolgt, vom *MainManager* aus, der Callback zur *Activity*. Analog oder nur unwesentlich unterschiedlich ist jegliche Kommunikation mit der *API* aufgebaut. Dabei spielt es keine Rolle, ob es sich um einen Sende- oder Ladevorgang handelt.

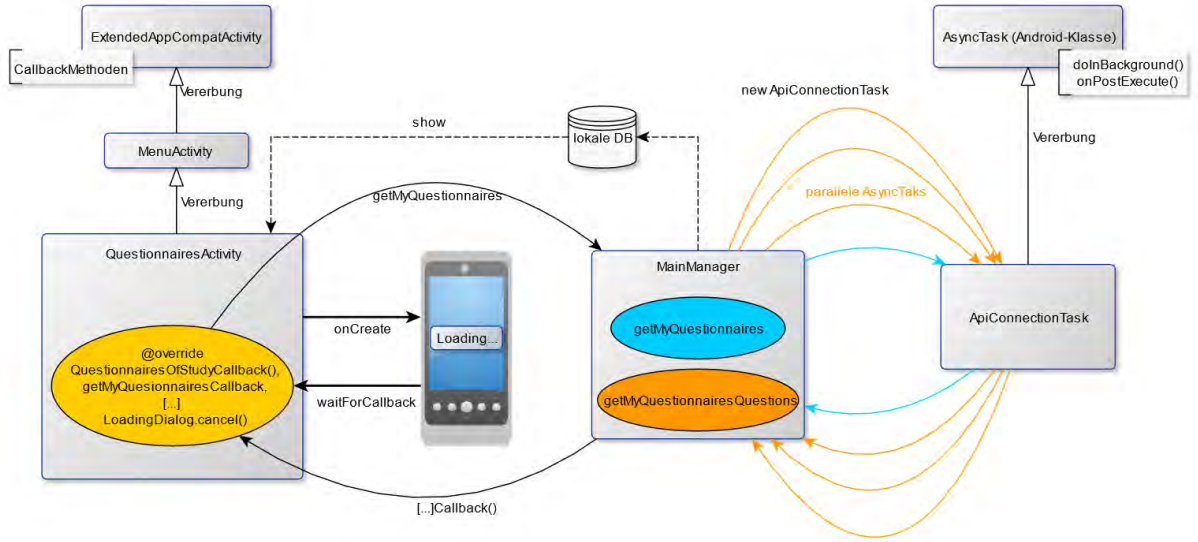


Abbildung 5: Programmablauf: Lade-/Sendevorgänge und parallele AsyncTasks

Eine Ausnahme stellt neben den Sendevorgängen der Antworten der einzelnen Befragungen in den einzelnen Modulen, das Versenden der Aufzeichnungen des Wiedergabeverhaltens des Audio-Players für die auditorische Stimulation, dar. Hierfür wird ein *Upload Service* verwendet. Vorteil ist, dass dieser vollständig im Hintergrund abläuft. Das heißt, dass es auch möglich ist Daten zu einem späteren Zeitpunkt abzuschicken, sollte aktuell keine Internetverbindung bestehen. Der *Upload Service* wird beim Start der Applikation, sowie bei einer funktionierenden Internetverbindung, erneut getriggert und sendet alle noch nicht gesendeten Daten an den Server. Gespeichert werden die noch nicht verschickten Daten lokal auf dem Gerät, bis diese gesendet werden konnten.

4.3. Datenstruktur/Datenmodell

Die generelle Datenstruktur wird von der *API* und der zugehörigen Datenbank vorgegeben. Die Applikation ist nach *dem Model-View-Controller* [35] Prinzip aufgebaut. Genauer wäre es den Aufbau als *Modell-ViewModel (mit Manager-Klasse)-View* zu bezeichnen. Die Daten zu Fragebögen, Studien etc. werden von der *API* als *JSON*-Objekte bereitgestellt. Aus diesen *JSON*-Paketen wird mittels der *GSON*-Klasse das Datenmodell typisiert für die Applikation generiert. Strukturell anschaulicher erklärt wird dies in Abbildung 8. Für die Generierung, mittels *GSON*, aus der *JSON-API* und eventuell wieder zurück, wurden auf Grundlage der gegebenen *JSON*-Objekte, folgende Klassen wie folgt definiert (vgl. Abbildung 6 und Abbildung 7):

4.3.1. Datenmodell der Applikation / ViewModel + Manager

Ausgenommen der *NotificationTime*-, der *Question*- und der *Answer*-Klasse sind alle in Abbildung 6 bzw. Abbildung 7 ersichtlichen Klassen direkt aus der *API*-Datenbank generierte Objekte und müssen somit strukturgleich sein. Das Datenmodell der Applikation bzw. die mittels *GSON* generierbaren Klassen-Objekte werden durch mehrere Klassen ergänzt. Darunter befinden sich die Klassen *MyStudy*, *MyQuestionnaire, Answer* und *Question.* 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 Klassen dienen weiter dazu, um schnellere und einfachere Zugriffe auf öfters verwendete Objekte und Attribute zu gewährleisten, ohne zusätzliche *Parsing*-Aufrufe zu benötigen. Ebenso enthält die *Answer*-Klasse den *JSON*-String, in welchem die Antworten zu den Fragen der Fragebögen zusammengebaut werden. Dieser wird dann an die *API* übermittelt. Die restlichen Klassen sind trivial erklärbar. *Questionnaire* enthält Informationen zu den Fragebögen und *QuestionSound* ist ein *„element“* eines Fragebogens. Die *UNITI*-Applikation enthält mehrere dieser *element*-Klassen, jedoch sind diese für die weiteren Module der *UNITI*-Applikation von Bedeutung und werden hier nicht weiter erläutert. Anhand dieser Klassen wird später entschieden, wie die einzelnen Fragen der Fragebögen aufgebaut sind. Genauere Informationen zu den anderen Klassen und Modulen sind hier [2] beschrieben. Analog zur *Questionnaire*-Klasse enthält *Study* die Informationen zu einer Studie, sowie die *Profile*-Klasse die Informationen über die Profildaten des Nutzers. Die *Feedback*-Klasse wird für Rückmeldungen an die Nutzer verwendet. Hierbei wird die *Answersheet*-Klasse dazu benutzt, um abzugleichen, ob und welche Antwortsätze für eine entsprechende *questionnarie-id* bereits vorhanden sind. Die *NotificationTime*-Klasse wird für Benachrichtigungen vom Gerät an den Nutzer benötigt. Ein kleiner Einblick in spezielle Implementierungsdetails, explizit auch für das Modul der auditorischen Stimulation ist in Kapitel 5 zu finden.

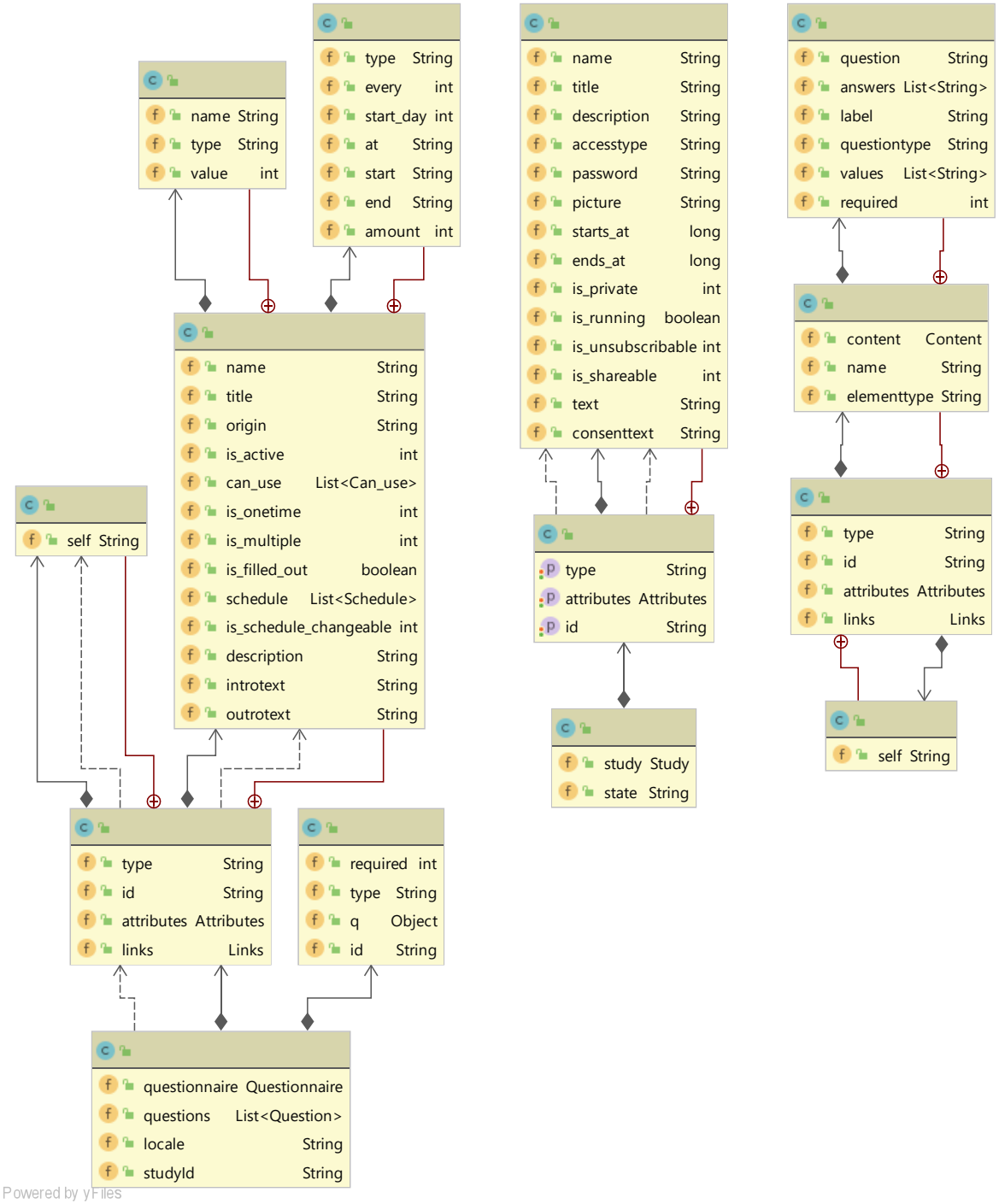


Abbildung 6: generiertes UML Klassendiagramm: ModelView der Applikation Teil 1

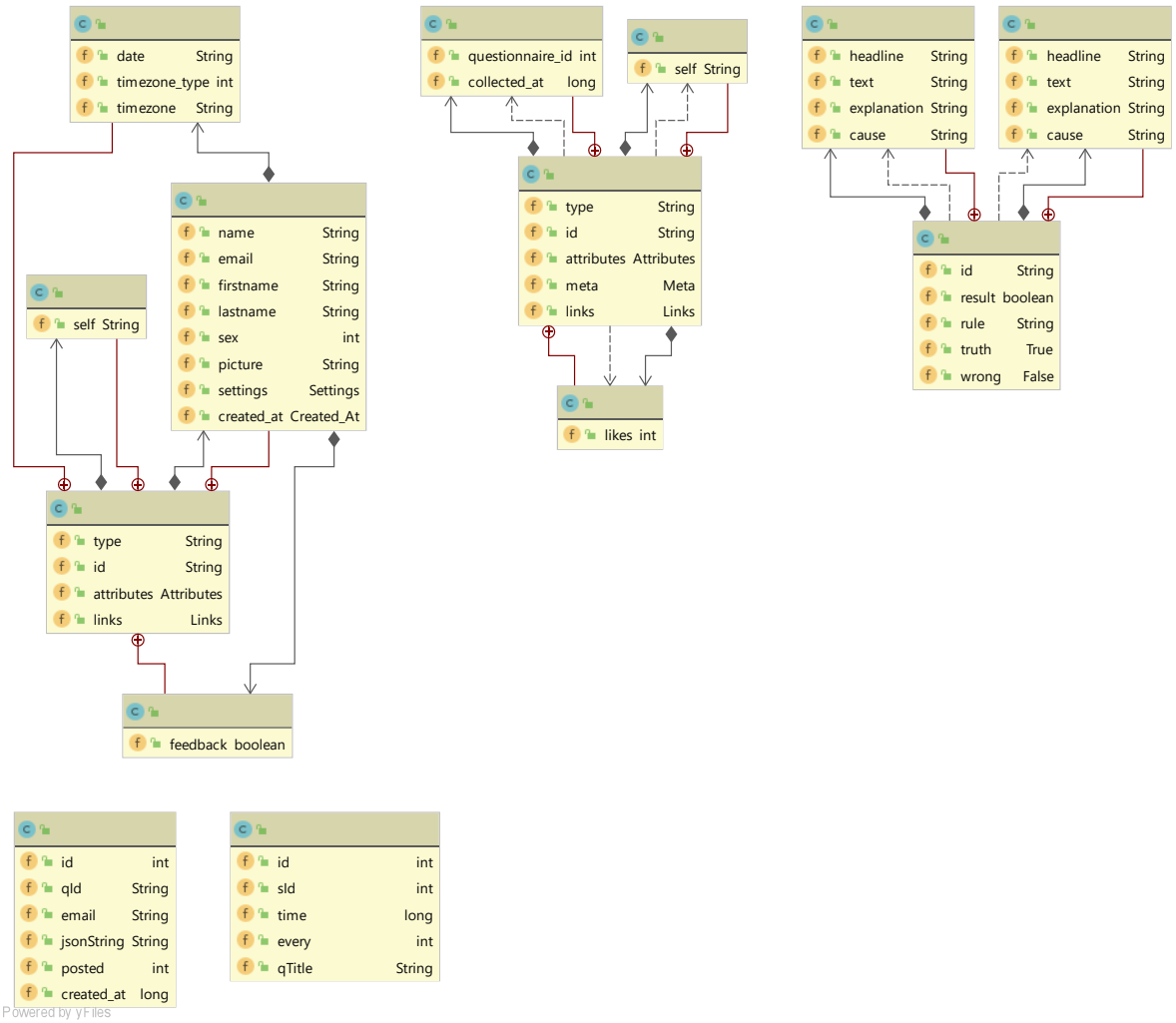


Abbildung 7: generiertes UML Klassendiagramm: ModelView der Applikation Teil 2

In Abbildung 8 wird ersichtlich, wie das Datenmodell des *ViewModel*, welches von der Applikation 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,* mithilfe der *put()*-Funktion, 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 [36], sofern die Struktur der *Models* genau mit derer in der *API/Datenbank* übereinstimmt.

Die Daten zu Studien, Fragebögen, etc. liegen in der Datenbank auf dem Server und werden als *JSON*-Objekte über die *API* bereitgestellt. Im *MainManager* wird eine neue Instanz des *ApiConnectionTasks* erzeugt und dadurch ein neuer *AsyncTask* gestartet. Die benötigten Parameter wie die HTTP-Methode (*GET*, *POST, DELETE,* etc.) werden direkt beim Aufruf übergeben. Anhand dieser HTTP-Methoden werden Daten vom *ApiConnectionTask* angefordert, gesendet, aktualisiert oder gelöscht. Der *ApiConnectionTask* ruft wiederrum mithilfe der *done()*-Funktion den *MainManger* auf. In dieser wird dann anhand der übergebenen *Callback*-Methode der weitere Programmablauf getriggert. Müssen Daten geparst werden, wird der *JsonParser* angestoßen, damit die Daten in der Applikation benutzbar sind oder um diese, im entsprechend richtigen Format, an den Server schicken zu können. Daten in dieser Kommunikation werden stets als *JSON*-Objekte übermittelt. Die *JSON*-Objekte werden mittels *GSON* automatisch zu den, in der Applikation, benötigten Daten-Objekten geparst. In diesem Projekt werden die benötigten Datenmodelle stets über *GSON* generiert. Lediglich die *POST*-Methoden, also die zu sendenden Antworten an die *API,* wurden größtenteils manuell implementiert, was implementierungstechnische Gründe hatte, da unterschiedliche *QuestionElemente* existieren, deren Antworten alle in das gleiche Format gebracht werden müssen.

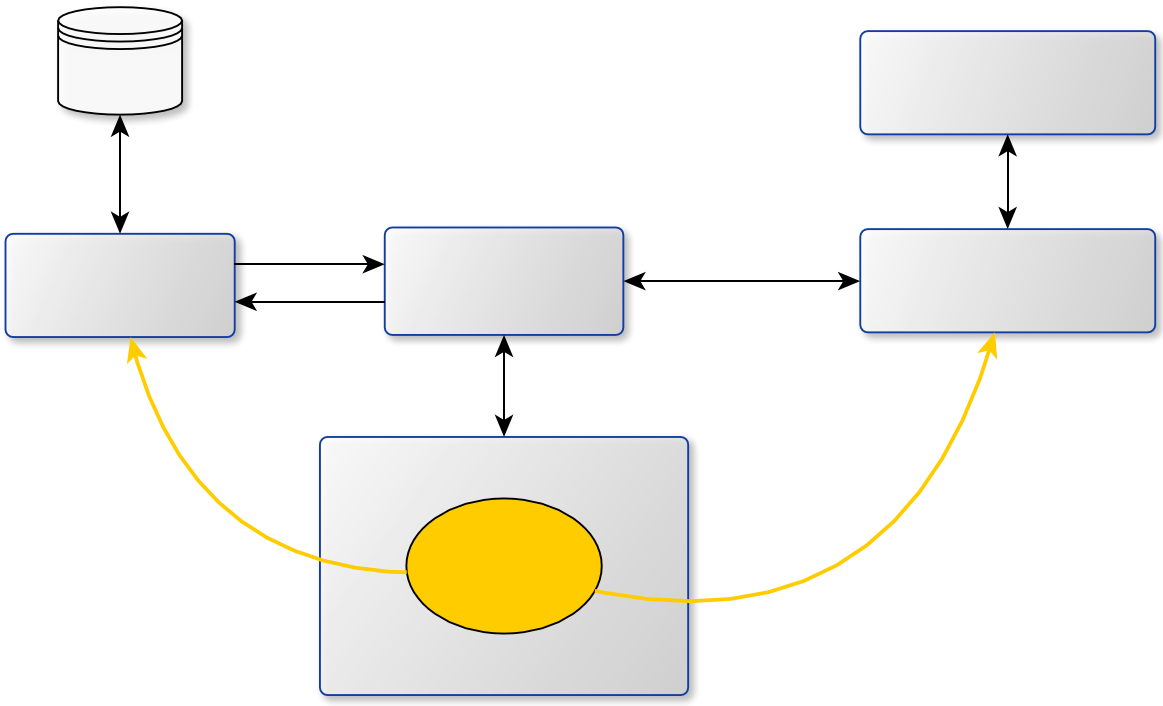


Abbildung 8: Programmablauf: Datenmodel/ViewModel Generierung mittels GSON

4.3.2. Lokale Datenbank

Die lokale Datenbank[24] umfasst die in Abbildung 9 dargestellten Tabellen und wurde so simpel wie möglich gehalten. Die in mehreren Tabellen enthaltene Spalte *Email* referenziert überall den bzw. die Nutzer, um einen Mehrbenutzerbetrieb, auf ein und demselben Gerät, auch offline zu ermöglichen. Die vom Server geladenen Studien werden alle in der Tabelle Studies gehalten. Dabei ist die Tabelle um die Spalte *State* erweitert. Diese Spalte ermöglicht die lokale Speicherung eines Studienstatus zu einem Nutzer, wodurch unterschieden werden kann, ob ein Nutzer einer Studie bereits beigetreten ist oder nicht. Weitere mögliche Zustände wären die Anfrage zur Teilnahme an einer Studie oder, ob der Nutzer zu einer Studie eingeladen wurde. Allerdings werden diese Zustände aktuell und im Rahmen dieser Arbeit serverseitig nicht genutzt. In der Tabelle *Questionnaires* sind alle benötigten Informationen zu den einzelnen Fragebögen der Module enthalten, die vom Server geladen wurden. Gespeichert werden jedoch nur die Fragebögen, die zu den Studien gehören, an welchen der Nutzer auch teilgenommen hat. Die Umsetzung in eine klassenähnliche Struktur, anstelle von *JSON*-Strings, ermöglicht es beim Austritt aus einer Studie, ohne weiteren Serverzugriff, nur die zugehörigen Fragebögen zu löschen und diese aktuell zu halten. Dadurch bleiben die Daten auch im Offline-Modus konsistent. In der Spalte *Local* ist hinterlegt in welcher Sprache die Fragebögen für den Nutzer bereitgestellt werden. Für die *Notifications* werden, die in den Fragebögen enthaltenen *Schedules*, in die Tabelle *Schedules* ausgelagert, da diese lokal auf dem Gerät gehalten werden und es dadurch ermöglicht wird, die *Schedules* verändern zu können. *Schedules* sind zu verstehen als eine Art Zeitpläne für die einzelnen Benachrichtigungen der Fragebögen an den Nutzer. Die Benachrichtigungszeiten, welche mithilfe der *Schedules* erstellt wurden, sind in der Tabelle *Notificationtimes* gesichert. Mit abgespeichert wird auch der Titel des Fragebogens (*QTitle*), um dem Nutzer, zum einen komfortabel und detailliert zu visualisieren zu welchem Fragebogen die angezeigte Benachrichtigung gehört, zum andern aber auch um, infolge eines eventuellen Neustarts des Gerätes, Zugriffszeiten für die erneute Initialisierung der *Notifications* zu verkürzen. Die Struktur eines Fragebogens wird in der *QuestionnaireStructure* Tabelle gespeichert. Eine Besonderheit ist, dass der zum Fragebogen gehöhrende JSON-String als solcher erhalten bleibt. Eine Umsetzung auf eine andere Struktur wie bei *Questionnaires* wird bewusst unterlassen, da ein Fragebogen unzählige verschiedene Elemente enthält und die entstehende Komplexität in der Datenbank keinen Mehrwert erzielen würde. Gleichermaßen enthält die *Answers* Tabelle einen solchen *JSON*-String. Jedoch beinhaltet dieser String einen Antwortsatz eines Nutzers zu einem Fragebogen. Für das korrekte Senden an den Server werden zusätzlich die ID des Fragebogens sowie die ID des Benutzers gespeichert. *Posted* wird lediglich dazu verwendet, um zu überprüfen, ob ein Antwortsatz bereits erfolgreich an den Server übermittelt wurde oder nicht. Für das Feedback werden die Tabellen *Answersheets* und *Evaluation* benötigt. Über die Spalte *Evaluated* wird festgehalten und geprüft, ob ein Antwortsatz bereits bewertet wurde oder nicht.

Zusätzlich zur lokalen Datenbank werden einige benötigte Informationen, hauptsächlich Key-Value Tupel, wie z.B. für den Login (Benutzername und Passwort), in den *SharedPreferences*[25] [34] der Applikation gehalten. Vorteil sind die deutlich einfacheren und schnelleren Zugriffe.

Generell bedeutet dies, dass eine Studie mehrere Fragebögen haben kann. Weiter hat ein Fragebogen jeweils eine Struktur und besitzt Referenzen zu eventuellen *Answers* bzw. *Answersheets*, mit zugehöriger Evaluation, referenziert über die ID des *Answersheets,* sowie zu den *Schedules*. Ein *Schedule* wiederrum hat eine oder mehrere *Notificationtimes*. Um die Applikation sowie das Modul der auditorischen Stimulation offline nutzen zu können, muss der Nutzer mindestens einmal erfolgreich online eingeloggt gewesen sein, da zu Beginn, einmal die Fragebögen plus deren Strukturen

[24] SQLite: https://www.sqlite.org/index.html

[25] SharedPreferences: https://developer.android.com/reference/android/content/SharedPreferences [29.09.2020]

von der *API* geladen werden müssen. Im *Offline-Modus* ist es in dieser Applikation daher lediglich möglich, die zuletzt, mit funktionierender Internetverbindung, geladenen Fragebögen zu bearbeiten. Die auditorische Stimulation nutzt eine Abwandlung der Fragebogenstruktur, um die Sounds zu visualisieren und abspielen zu können. Die Sounddateien sind dabei lokal gehalten und werden mit der Installation der Applikation auf dem Gerät abgelegt. Die Applikation ist auch offline voll funktionsfähig. Dies gilt auch für den Mehrbenutzerbetrieb auf demselben Gerät. Voraussetzung ist, dass sich jeder Nutzer des Gerätes mindestens einmal erfolgreich, online am mobilen Gerät angemeldet hat. Sollte keine Internetverbindung bestehen und ein Nutzer beendet seine Sitzung im auditorischen Stimulations-Modul, werden die Ergebnisse lokal gespeichert. Durch den *Upload-Service* wird zu einem späteren Zeitpunkt ein erneuter Versuch gestartet diese Antworten an den Server zu senden. Um keine redundanten Daten zu verschicken, wird mithilfe von *posted* vermerkt, welche *Answers* bereits erfolgreich versendet wurden.

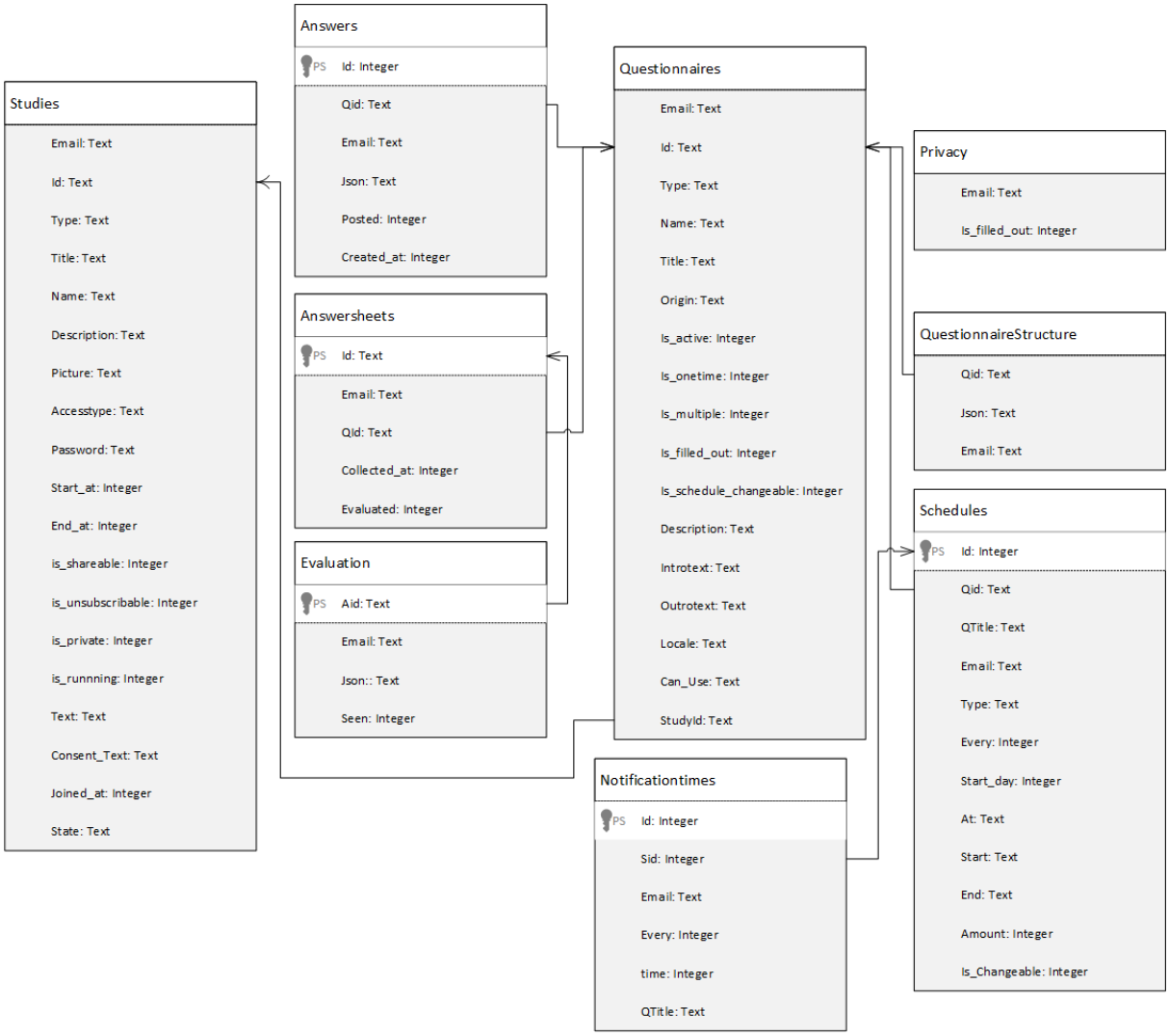


Abbildung 9: UML Datenbankdiagramm: Struktur lokale Datenbank

Der *Offline-Modus* funktioniert generell identisch wie eine Nutzung der Applikation mit Internetverbindung. Der Unterschied besteht darin, dass beim Fehlschlagen der Serverkommunikation, etwa bei einer fehlenden Internetverbindung, ein anderer *result-code* zurückgeben wird. Es konnten also z.B. keine Daten geladen werden, ergo wird auf der lokalen Datenbank auch nichts neues gespeichert oder überschrieben. Ab diesem Zeitpunkt verhält sich die Applikation wieder wie im Online-Modus, da die Daten, die visualisiert werden, immer von der lokalen Datenbank bereitgestellt werden. Zu Beginn ist die lokale Datenbank leer, was erklärt warum beim Erststart eine funktionierende Internetverbindung Voraussetzung ist. Fehlende Daten oder ähnliche Probleme werden dem Nutzer entsprechend mitgeteilt. Das grobe Ablaufschema, sowie Online-, Offline-Unterschied wurde versucht in Abbildung 10 bildlich darzustellen.

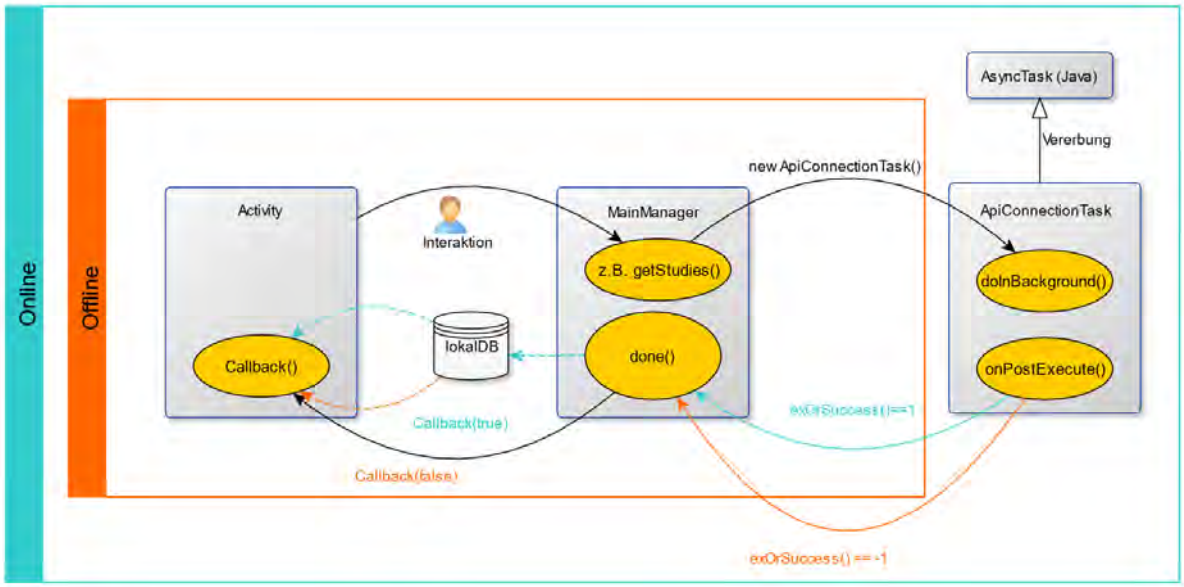


Abbildung 10: Programmablauf: Unterschied Kommunikation Offline vs. Online

4.4. Controller-, Manager- und Handlerklassen

Komplettiert wird das *Model* der Applikation durch die Klasse *ApiConnectionTask*. Diese Klasse stellt zusammen mit dem *MainManager* das Bindeglied zwischen *API* und dem intern angepassten Datenmodell der Präsentationsschicht (*ViewModel*) dar. Die Klasse *ApiConnectionTask* erbt von der Android-Klasse *AsyncTask* und überschreibt dessen Funktionen *doInBackground* und *onPostExecute*. Im Wesentlichen beinhaltet diese Klasse die beiden genannten Methoden. In Ersterer finden die eigentlichen Zugriffe auf die *API* statt. Sind diese beendet, wird Zweitere angestoßen. In dieser wird lediglich die *done()*-Methode des *MainManagers* aufgerufen. Zum anderen findet man im *ApiConnectionTask* noch eine interne Klasse *Result*, welche als eine Art Container angesehen werden kann und die Daten der Kommunikation vereint, die gebündelt benötigt werden. Weitere implementierte Klassen für die definierten Anforderungen sind: die bereits in Kapitel 4.3.1 genannten *MainManager* und *JsonParser*, sowie die Klassen *BootCompleteReceiver, MyNotificationManager, NotificationPublisher, NotificationPublisherFB, SetNotificationService, PlaybackStateListener, SoundsHelper, ImageHelper, SeekDisabler, ResultListAdapter, StudyListAdapter, UploadService* und *MyDBHandler*.

Wie bereits in Abbildung 8 zu sehen, wird im *ApiConnectionTask* die eigentliche Verbindung zum Server bzw. zur *API* aufgebaut, Daten übertragen und die Verbindung wieder geschlossen. Die Daten werden dabei über *BufferedOutputStreams*[26], *BufferedReader*[27] sowie *InputStreamReader*[28] verarbeitet [34]. Es können hierbei mehrere *ApiConnectionTasks* bzw. *AsyncTasks* parallel ablaufen. In der *onPostExecute*-Methode wird dann die *done()-Methode* des *MainManagers* aufgerufen, wobei das *Result* übergeben wird. Somit kann, anhand der im *Result* enthaltenen *doneMethode*, der weitere Programmverlauf getriggert werden. Dies können weitere Funktionsaufrufe, neue *API*-Verbindungen oder eine lokale Speicherung von Daten sein. Diese Art der Kommunikation funktioniert auch dann, wenn mehrere *AsyncTasks* parallel laufen.

Der *MainManager* ist die zentrale Klasse und „Steuereinheit“ der gesamten Applikation. Inhalt sind sämtliche statische Variablen und Funktionen für (lokale) Lade-, Speicher- und Visualisierungsvorgänge. Auf bedeutende Implementierungsdetails und Beteiligungen in Bezug auf die Funktionalität der auditorischen Stimulation wird in Kapitel 5 eingegangen.

Der *JsonParser* dient dazu die Daten und Dateien, die von der *API* geladen oder umgekehrt an die *API* gesendet werden sollen, entsprechend zu parsen. Die *JSON*-Objekte, wie sie die *API* bereitstellt, müssen derart verarbeitet werden, dass diese von der Applikation verwendet und abgearbeitet werden können. Auf der anderen Seite wird der *JsonParser* gleichermaßen dazu verwendet, von der Applikation verarbeitete, geänderte oder neue Daten dementsprechend zu parsen, damit diese dann an den Server gesendet und in der Datenbank abgelegt werden können. Die *Parser-Funktionen* können dabei, wie bereits in Kapitel 4.3.1 erläutert, mithilfe der Klasse *GSON* oder auch manuell implementiert werden[29].

Die Klassen *BootCompleteReceiver*, *MyNotification*Manager*, NotificationPublisher, NotificationPublisherFB, SetNotificationService* sind für die Benachrichtigungen der Applikation an den Nutzer von Nöten. Die Benachrichtigungszeiten werden zu Beginn vom Server geladen und lokal gespeichert, um den Nutzer auch offline benachrichtigen zu können. Benutzt wird dafür in Android ein

[26] BufferedOutputStream: https://developer.android.com/reference/java/io/BufferedOutputStream.html [28.09.2020]

[27] BufferedReader: https://developer.android.com/reference/java/io/BufferedReader.html [28.09.2020]

[28] InputStreamReader: https://developer.android.com/reference/java/io/InputStreamReader.html [28.09.2020]

[29] grober Ablauf in Abbildung 8 zu sehen

AlarmManager[30] [34]. Dieser sendet zu den bestimmten Zeitpunkten einen *Broadcast*, welcher von einem *BroadcastReceiver*[31] [34], hier die erbende Klasse *NotificationPublisher*, empfangen wid. Dort wird dann die eigentliche Benachrichtigung für den Nutzer 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 den *BootCompleteReceiver* gelöst, welcher ebenfalls von der Klasse *BroadcastReceiver* erbt und beim Neustart des Android Gerätes aktiviert wird. Dieser startet in seiner *onReceive*-Funktion die *SetNotificationService*-Klasse, welche wiederum den *MyNotificationManager* aufruft, um dort erneut die Zeitpunkte für die Benachrichtigungen im *AlarmManager* zu registrieren. *MyNotificationManager* ist für die Verwaltung, d.h. für das Speichern, Löschen und Ändern der Benachrichtigungszeiten verantwortlich. Das *FB* in der Klasse *NotificationPublisherFB* steht für Feedback. Dieser Publisher ist demnach für das Erstellen der Feedback-Benachrichtigungen an den Nutzer zuständig. Genauere Details zu den Benachrichtigungen, sowie Implementierungstechniken sind hier [2] nachzulesen. *MyDbHandler* dient zur Verwaltung der lokalen Datenbank, um einen weitestgehend normalen Betrieb der Applikation auch ohne funktionierende Internetverbindung zu ermöglichen. Weitere Infos in Kapitel 4.3.2.

[30] AlarmManager: https://developer.android.com/reference/android/app/AlarmManager.html [28.09.2020]

[31] BroadcastReceiver: https://developer.android.com/reference/android/content/BroadcastReceiver [28.09.2020]

PlaybackStateListener, SoundsHelper, SeekDiabler und *ImageHelper* (zu sehen in Abbildung 11) werden alle dafür benutzt, um die auditorische Stimulation bzw. den Player zu realisieren. Zu sehen ist, dass mithilfe des *SoundsHelper* die einzelnen, lokal gespeicherten, Sounddateien über ihren Namen bzw. Label zugeordnet werden. Hierdurch werden diese in der Applikation korrekt positioniert, sodass die korrekte Datei mit dem zugehörigen Namen des Sounds verknüpft wird. Dieses *Matching* über den Soundname musste implementiert werden, da es zu viel Datentransfer mit sich gebracht hätte, würden die Sounds über den Server ausgeliefert werden. Der Server liefert nun nur eine Art „Fragebogen“ mit den Informationen, wie dem Namen, plus Befragung zum Befinden bezüglich des Sounds. Der *ImageHelper* funktioniert analog zum *SoundHelper*, allerdings dient dieser dazu die einzelnen Bilddateien der Sounds, die angezeigt werden sollen, richtig zuzuordnen. Der *SeekDisabler* ist lediglich dafür da, das vor- und zurückspulen im Audio-Player zu verhindern. Die Klasse *PlaybackStateListener* dient dazu das Wiedergabeverhalten eines Nutzers, zu einem Sound, festzuhalten. Hierfür wird der Inhalt des *Enum Actionstate* als Schlüsselworte genutzt. Zu diesen Schlüsselworten werden ergänzend die Zeitstempel zum Zeitpunkt der Aktion addiert, wodurch eine Menge an Key-Value Paaren entsteht. Genauere Implementierungstechniken und Details werden in Kapitel 5, respektive Kapitel 5.2 beschrieben.

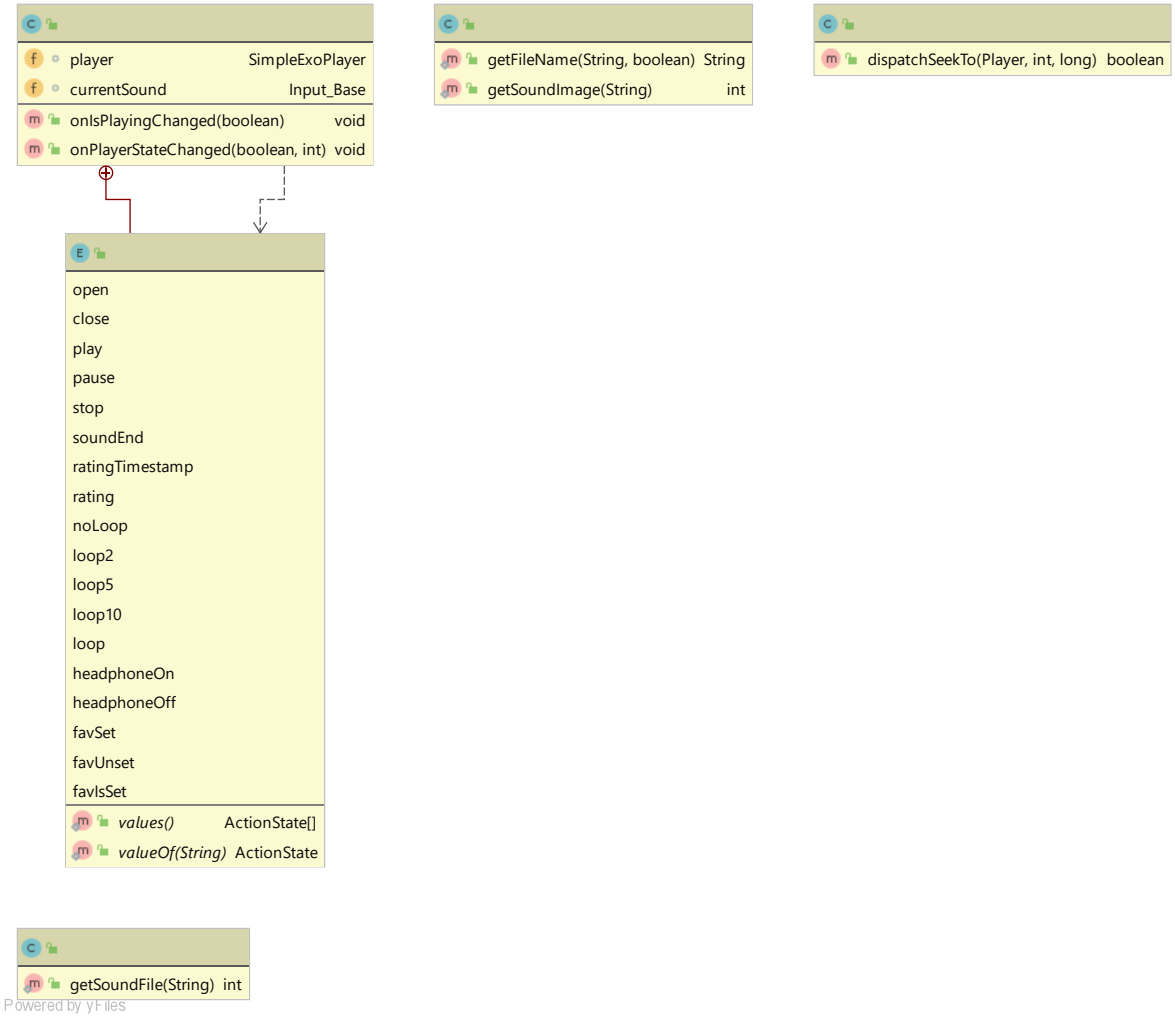


Abbildung 11: generiertes UML Klassendiagramm: relevante Klassen für Auditorische Stimulation

4.5. Views

Für die Präsentationsschicht der Applikation wurden verschiedene Views, in Android auch *Activities* genannt, entwickelt und implementiert. Dabei ergeben sich mehrere *Vererbungen*. Jede *Activity*[32]-Klasse [34] erbt von der Super-Klasse *AppCompatActivity*[33] [34]. 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“ *ExtendedAppCompatActivity* gestellt, wie in Abbildung 12 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*[34] [34] und die Dialogfunktionen zu nennen. Weiter werden dort die *ActivityCallbacks* deklariert, die anschließend in den entsprechenden *Activitys* überschrieben werden. Die einzelnen Sounds werden in der *QuestionaireStructureActivity* visualisiert.

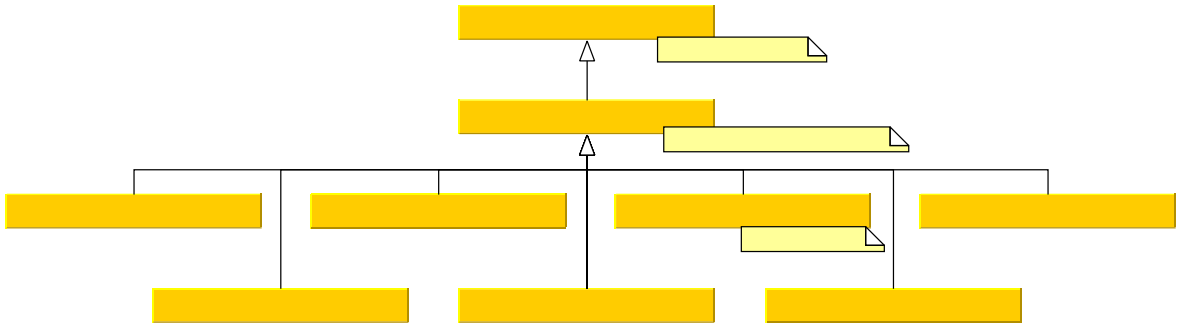


Abbildung 12: UML Klassendiagramm: Views Vererbung: ExtendedAppCompatActivity

Die *MenuActivity* stellt nochmals eine spezielle *Activity*-Klasse dar. Von ihr erben alle weiteren *Activitys*, die ein Menü benötigen, um durch die Applikation zu navigieren. Welche *Activities* dies sind, ist in Abbildung 13 zu sehen. Das Modul der auditorischen Stimulation wird über die *QuestionairesActivity* erreicht. Dafür wird diese lediglich minimal optimiert.

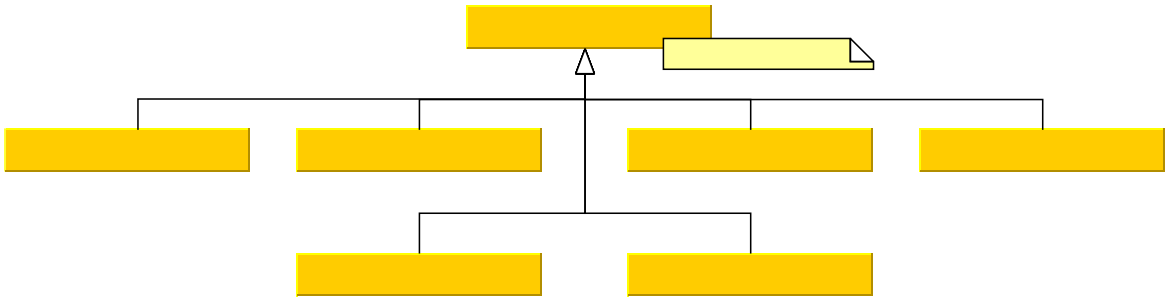


Abbildung 13: UML Klassendiagramm: Views Vererbung: MenuActivity

[32] Activity-Klasse: https://developer.android.com/reference/android/app/Activity [29.09.2020]

[33] AppCompatActivity: https://developer.android.com/reference/android/support/v7/app/AppCompatActivity.html [29.09.2020]

[34] AlertDialog: https://developer.android.com/reference/android/app/AlertDialog.Builder.html [29.09.2020]

Zusätzlich zu diesen *Activity*-Klassen wurde die *QuestionView*-Klasse implementiert. Diese ist eine reine *View-Model*-Klasse. Benötigt wird diese *QuestionView*-Klasse, um die jeweiligen, einzelnen Fragen der Fragebögen anhand ihres Typus zu unterscheiden und zu kreieren. Infolgedessen werden ebenso die entsprechenden Eingabemöglichkeiten (*Inputs*), zu den verschiedenen Fragetypen, erstellt und entsprechend visualisiert. Für die eben genannten *Inputs* wurden wiederrum mehrere *View*-Klassen implementiert. Für die auditorische Stimulation von Bedeutung sind *Input_Base* und *Input_Sound,* die restlichen Klassen sind der Vollständigkeit halber trotzdem bildlich angeführt (Abbildung 14). Die *Input_Base*-Klasse ist eine abstrakte „Überklasse" von welcher die restlichen *Input*-Klassen erben. Benutzt wurde diese Struktur abermals, um Variablen und Funktionen, die bei jedem *Input* benötigt werden, zentral zu realisieren. Beispiel hierfür sind die *delete* oder *set* Funktionalitäten. *Input_Base* erstellt eine Grundstruktur der Unterklassen und setzt den Pointer auf den entsprechenden *QuestionView*. Die einzelnen Klassen wie *Input_Sound* und *Input_SingleChoice* enthalten zusätzlich, für die Eingabe benötigte Strukturen, Funktionen und Objekte, wie einen *OnClickListener*[35] oder *RadioGroups*[36] [34].

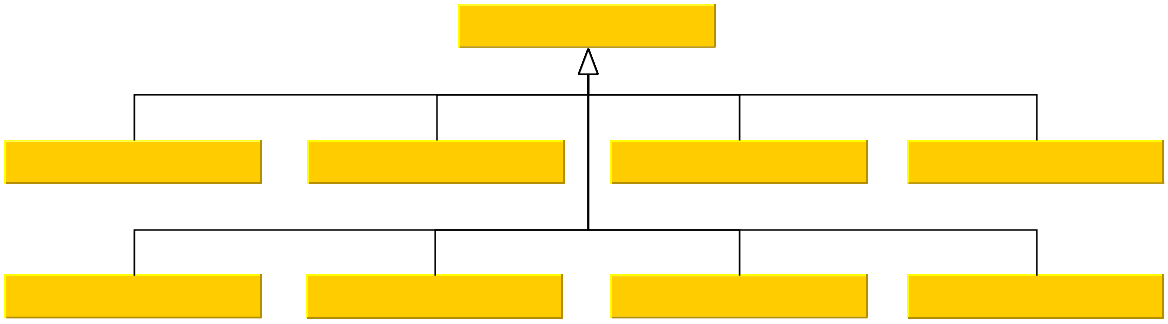


Abbildung 14: UML Klassendiagramm: Views Vererbung: Input_Base

4.5.1. ListAdapter

Für die Organisation und Visualisierung verschiedener Listen werden mehrere *ListAdapter* benötigt. Implementiert wurden deshalb ein *StudyListAdapter*, ein *AnswersheetListAdapter*, ein *ResultsListAdapter* sowie ein *QuestionaireListAdapter*. Allesamt erben jeweils von der Android-Klasse *ArrayAdapter*[37] [34]. Der *StudyListAdapter* unterstützt den Aufbau und die Funktionalität der Studienliste in der *StudiesActivity*. *AnswersheetListAdapter* und *ResultsListAdapter* spielen für die Organsisation und Visualisierung des *Feedbacks* eine wichtige Rolle. Für den Aufbau und das Aussehen der Liste an Fragebögen ist der *QuestionnaireListAdapter* mitverantwortlich.

[35] OnClickListener: https://developer.android.com/reference/android/view/View.OnClickListener [30.09.2020]

[36] RadioGroup: https://developer.android.com/reference/android/widget/RadioGroup [30.09.2020]

[37] ArrayAdapter: https://developer.android.com/reference/android/widget/ArrayAdapter.html [30.09.2020]

4.6. Zusammenfassung der Struktur

Im Allgemeinen lässt sich am Ende folgendes, in Abbildung 15 ersichtliches Schema des *Model-ViewModel + MainMainager-View* Aufbaus darstellen. Der *MainManager* ist für die Kommunikation zwischen Server und Applikation zuständig. Ebenso sorgt der *MainManager* zusammen mit dem *JsonParser* für die jeweils richtigen Datenstrukturen. Sei es für das Datenmodel der Applikation, um alles korrekt und ansprechend zu visualisieren oder die Struktur, welche entsprechende Daten benötigen, um an den Server gesendet zu werden. Ebenso ist der *MainManager* dazu da, zu erkennen, ob eine online Verbindung besteht, ob neue Daten geladen und lokal gespeichert werden oder ob momentan rein lokal gearbeitet wird.

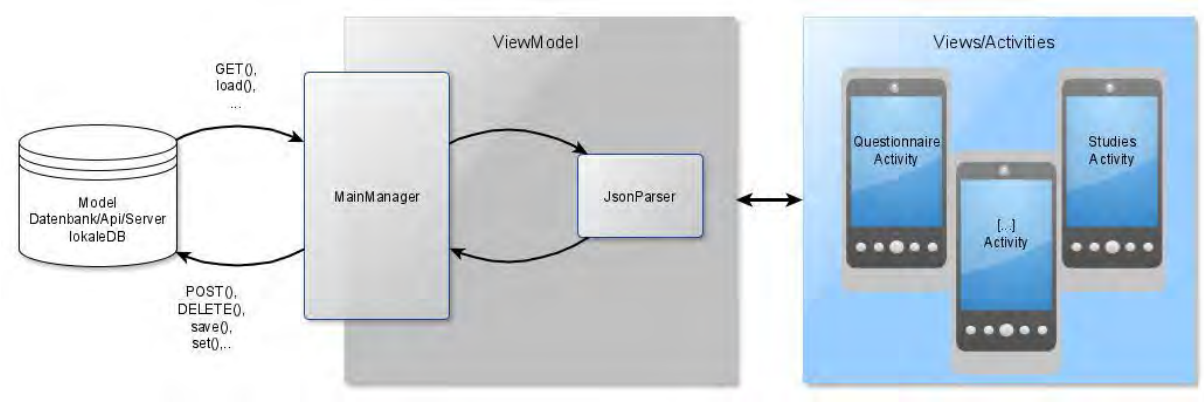


Abbildung 15: Gesamtstruktur der Applikation: Model-ModelView + MainManager-View

5. Implementierung

Dieses Kapitel befasst sich mit bestimmten Bereichen der Implementierung der *UNITI*-Applikation, welche für die Erfüllung der gestellten Anforderungen nötig sind. Dabei wird beleuchtet, wie erreicht wird, dass ein autorisierter Nutzer, neben der Studie zur auditorischen Stimulation, auch anderen Studien in der Applikation beitreten kann. Zu beachten galt dabei, dass nur diejenigen die sich selbst oder anonym in der Applikation registriert haben die Möglichkeit besitzen, Studien beizutreten oder diese wieder zu verlassen. Ebenso soll die Applikation dynamisch darauf reagieren, welchen Studien ein Nutzer beigetreten ist und das Bottom Menü entsprechend aufbauen. Gleichermaßen betrachtet wird die Implementierung des Moduls der auditorischen Stimulation. Eingegangen wird auf Aufbau, Datenhaltung, Datentransfer und Visualisierung bzw. Realisierung der Implementierung für die Wiedergabe der einzelnen Sounds. Zuletzt wird auf Implementierungstechniken der Serverkommunikation eingegangen, welche in Kapitel 4 bereits ausführlich strukturell erklärt wurde, sowie auf eine mögliche Auslagerung dieser Serverkommunikation als Library.

5.1. Studien: Teilnahme/Austritt und Verhalten der Gesamtapplikation

Eine der grundlegenden Funktionalitäten der *UNITI*-Applikation ist die Funktionalität an verschiedenen Studien teilnehmen zu können. Generell wäre es möglich unzählige Studien in dieser Applikation unterzubringen, sofern die Aktivitäten und Aufgaben der Studie durch die jetzigen Strukturen der Applikation und des Servers umsetzbar sind. Zum Zeitpunkt der Arbeit sind drei verschiedene, aber sich ähnelnde Studien in der Applikation enthalten. Diese sind neben dem auditorischen Stimulations-Modul, durch ein Edukations- und ein Tinnitus-Fragebogen-Modul umgesetzt. Die zwei Letzteren werden in einer anderen Arbeit [2] genauer betrachtet. Diese Arbeit bezieht sich auf das Modul der auditorischen Stimulation und den Gesamtrahmen der Applikation. Durch die Möglichkeit der Nutzung verschiedener Studien in einer Applikation, entsteht die Möglichkeit eventuelle Zusammenhänge, Korrelationen und Kausalitätsketten von Krankheitsbildern, verschiedener Nutzer, zu erkennen.

Nutzern, welchen vom Administrator ein Nutzername, sowie ein Passwort zugeteilt bekommen haben, ist es im Rahmen dieser Arbeit weder möglich den anderen Studien beizutreten, noch, den vorab eingeschriebenen Studien, wieder auszutreten. Dies ist für eine Mindestanzahl an Teilnehmern pro Studie, für ein aussagekräftiges Ergebnis, relevant. Generell sind, zusätzlich zu den Status „teilgenommen (*joined*)“ und „nicht teilgenommen (*none*)“, weitere Status möglich, die in dieser Arbeit, zu diesem Zeitpunkt, allerdings keine Rolle spielen. Nichtsdestotrotz ist in Abbildung 16 eine Übersicht der verschiedenen Zustände und Zustandswechsel dargestellt. Für die *UNITI*-Applikation sind aktuell lediglich die Zustände der *blau* markierten Pfade von Bedeutung. Über den von der *API* gegebenen *AccessType* werden, programmtechnisch, die verschiedenen Zustände für die Studien kreiert. Ein Nutzer könnte sich demnach, pro Studie, in je einem der vier Zustände befinden, bzw. mittels entsprechender Interaktionen die Zustände wechseln. In Bezug auf die verschiedenen möglichen Status der Studie war es nötig, wie in Kapitel 4.3.1 angeführt, eine Klasse *MyStudy* mit einem *String State* zu erstellen, um den jeweiligen Status der Studie festzuhalten.

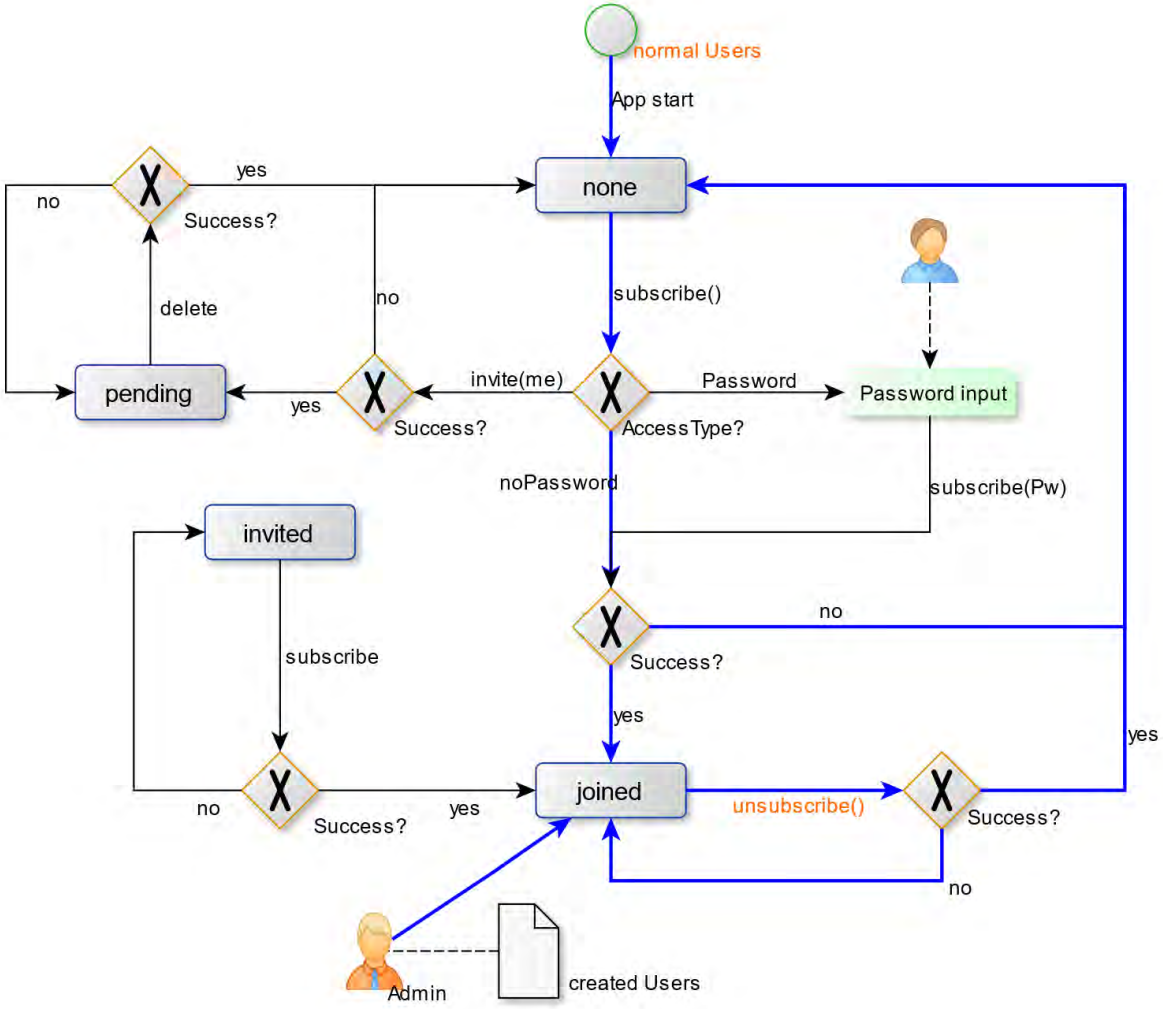


Abbildung 16: Prozessablaufdiagramm Studien: Status und Statuswechsel

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

Listing 1: Study

```
1 public class Study {
2
3     public String type;
4     public String id;
5     public Attributes attributes;
6
7     public Study() {
8         attributes = new Attributes();
9     }
10
11    public String getType() {
12        return type;
13    }
14
15    public void setType(String type) {
16        this.type = type;
17    }
18
19    public String getId() {
20        return id;
21    }
22
23    public void setId(String id) {
24        this.id = id;
25    }
26
27    public Attributes getAttributes() {
28        return attributes;
29    }
30
31    public void setAttributes(Attributes attributes) {
32        this.attributes = attributes;
33    }
34
35    public class Attributes {
36         public String name;
37         public String title;
38         public String description;
39         public String accesstype;
40         public String password;
41         public String picture;
42         public long starts_at;
43         public long ends_at;
44         public int is_private;
45         public boolean is_running;
46         public int is_unsubscribable;
47         public int is_shareable;
48         public String text;
49         public String consenttext;
50     }
51 }
```

Zusätzlich wurde eine Klasse MyStudy implementiert, in welcher ein zusätzlicher *String state* (Listing 2, Zeile 1) gehalten wird. Dieser wird je nach Nutzer und dessen aktuellen Zustand, bezüglich einer Studie, gesetzt. Mögliche Zustände können *none*, *invited*, *pending* oder *joined* sein. Für die *UNITI*-Applikation werden vorerst aber nur *none* und *joined* benötigt.

Listing 2: MyStudy

```
1 public class MyStudy {
2    public Study study;
3    public String state;
4 }
```

Wird die *StudiesActivity* (Listing 3), die Ansicht mit der Liste der Studien, vom Benutzer angefordert, wird in der *onCreate*-Funktion die *getAllStudies*-Funktion des *MainManagers* aufgerufen, wodurch alle Studien von der Datenbank geladen werden. Dabei wird dem *MainManager* die aktuelle *Activity* direkt als Parameter mitgeben. Dadurch wird sichergestellt, dass der *Callback* wieder zu dieser *Activity* zurückkommt und die entsprechende *getAllStudiesCallback*-Funktion aufgerufen wird (Zeile 133). In der besagten *Callback*-Funktion werden die aktiven Studien (*isRunning*), über die *MainManager* Funktion *getActiveStudies(),* geladen. Diese Liste wird dem *StudyListAdapter* (Listing 5) übergeben. Der *Adapter* wiederum wird der *ListView* übergeben, wodurch die Liste an Studien visualisiert werden kann (Zeile 152ff). Ab Zeile 22 sieht man die Funktion, welche die Änderung der Studien-Zustände triggert (*changeStudyState*()). Zuerst werden verschiedene Daten geprüft, ob ein Token vorhanden ist und der Studie nicht bereits beigetreten wurde, sowie die Art des *AccessTypes*. Relevant ist in dieser Arbeit der *else*-Fall in Zeile 35f, da dies der Programmablauf für frei zugängliche Studien ist. Durch *openConsentDialog*(*myStudy*) wird die Methode *subscribeStudy()* im *MainManager* angestoßen. Diese Funktion endet in der *StudyActionCallback*-Funktion in Zeile 67. Hier wird anhand der Boolean Variable *success* und der *action* eine positive oder negative Rückmeldung der Applikation, an den Nutzer, geliefert (Erfolgsfall in Zeile 82 *case subscribe*). Weiter werden die folgenden Reaktionen der Applikation getriggert, wie z.B. die Liste der Studien aktualisiert (Zeile 122ff). Der dynamische Aufbau bzw. die unterschiedliche Menüstruktur der *BottomNavigationView*, bedingt durch die Teilnahme an den einzelnen Studien wird in Zeile 72 angestoßen. Die Funktion ist in der *MenuActivity* Klasse deklariert, kann hier aber durch die Vererbung direkt aufgerufen werden. Ein Ausschnitt der Implementierung dieser Funktion ist in Listing 4 zu sehen.

Listing 3: StudiesActivity

```
1  public class StudiesActivity extends MenuActivity {
2
3      public ListView lvStudies;
4      public ArrayAdapter studiesAdapter;
5      public List<MyStudy> activeStudies;
6      public AlertDialog codeDialog;
7      TextView tvStudiesHint;
8      public final static int requestCode = 1;
9      public final static String resultString = "result";
10     @Override
11     int getBottomNavigationMenuItemId() {
12         return R.id.menu_studies;
13     }
14     @Override
15     int getLayoutId() {
16         return R.layout.activity_studies;
17     }
18
19
20     //ButtonClicked Methods
21
22     public void changeStudyState(final MyStudy myStudy) {
23         if(MainManager.token.equals("")){
24             showStatus(res.getString(R.string.offline_mode));
25             return;
26         }
27         if (myStudy.state.equals("none")) {
28             if (myStudy.study.attributes.accesstype != null) {
29                 if (myStudy.study.attributes.accesstype.equals("password")) {
30                    showCodeDialog(myStudy);
31                 } else {
32                     showLoading(res.getString(R.string.loading));
33                     manager.subscribeInviteStudy(myStudy, null);
34                 }
35             } else {
36                 openConsentDialog(myStudy);
37             }
38         } else {
39             switch (myStudy.state) {
40                 case "joined":
41                     openUnsubscribeDialog(myStudy);
42                     break;
43                 case "pending":
44                     showLoading(res.getString(R.string.loading));
45                     manager.deletePendingStudy(myStudy);
46                     break;
47                 case "invited":
```

```
                    showLoading(res.getString(R.string.loading));
                    manager.subscribeStudy(myStudy, null);
                    break;
                default:
                    break;
            }
        }
    }

    public void openConsentDialog(final MyStudy myStudy){...}
    public void showInformations(MyStudy myStudy, View ContentView){...}
    public void openQuestionnairesDialog(){...}
    public void openUnsubscribeDialog(final MyStudy study){...}
    public void opensubscribedDialog(String message, String title){...}


    //Overrided Callback Methods

    @Override
    public void studyActionCallback(boolean success, String action) {
        loadingDialog.cancel();
        if(success)
        {
            List<MyStudy> myStudies = manager.getMyStudies();
            updateNavigationBarItems(myStudies);
            int count =  manager.CountofQToFill();
            if(count > 0){
                (navigationView.getOrCreateBadge(R.id.menu_questionnaires)).setNumber(count);
            }
            else
            {
                navigationView.removeBadge(R.id.menu_questionnaires);
            }
            switch(action){
                case "subscribe":
                    opensubscribedDialog(getString(R.string.subscribed_text),
                            res.getString(R.string.successful_subscribed));
                    break;
                case "subscribeInvite":
                    showStatus(res.getString(R.string.successful_requested));
                    break;
                case "unsubscribe":
                case "delete":
                    showStatus(res.getString(R.string.successful_unsubscribed));
                    break;
                default:
                    break;
            }
        }
        else
        {
            switch(action){
                case "subscribe":
                    showStatus(res.getString(R.string.subscribing_failed));
                    break;
                case "subscribeInvite":
                    showStatus(res.getString(R.string.request_failed));
                    break;
                case "unsubscribe":
                case "delete":
                    showStatus(getString(R.string.unsubscribing_failed));
                    break;
                default:
                    break;
            }
        }
        if(manager.getMyStudies().isEmpty()){
            tvStudiesHint.setVisibility(View.VISIBLE);
        }
        else{
            tvStudiesHint.setVisibility(View.GONE);
        }

        //Aktualisierung der Tabelle

        activeStudies = manager.getActiveStudies();
        studiesAdapter = new StudyListAdapter(this,activeStudies);
        lvStudies.setAdapter(studiesAdapter);
    }

    @Override
    public void onResume() {...}
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {...}

    @Override
    public void getAllStudiesCallback(boolean success) {
        loadingDialog.cancel();
        int count =  manager.CountofQToFill();
```

```
136         if(count > 0 && getIntent().getBooleanExtra(FirstLaunchActivity.intent_extra_start,false)){
137             getIntent().removeExtra(FirstLaunchActivity.intent_extra_start);
138             openQuestionnairesDialog();
139         }
140         if(!success)
141         {
142             Toast.makeText(myView,res.getString(R.string.refresh_failed),Toast.LENGTH_LONG).show();
143         }
144         List<MyStudy> myStudies = manager.getMyStudies();
145         updateNavigationBarItems(myStudies);
146         if(myStudies.isEmpty()){
147             tvStudiesHint.setVisibility(View.VISIBLE);
148         }
149         else{
150             tvStudiesHint.setVisibility(View.GONE);
151         }
152         activeStudies = manager.getActiveStudies();
153         studiesAdapter = new StudyListAdapter(this,activeStudies);
154         lvStudies.setAdapter(studiesAdapter);
155     }
156 }
```

In Listing 4 ist die Update Funktionalität der Menüstruktur zu sehen. Zuerst werden alle Items deaktiviert und nicht mehr sichtbar gemacht. Danach wird anhand der Studiennamen, die in der Liste *studies* enthalten sind und übergeben wurden, geprüft welche Studien vorhanden sind. Hierdurch hat der Nutzer eine auf sich und seine teilgenommenen Studien angepasste Menüstruktur, da ihm nur die Module der gefundenen Studien im Menü angezeigt werden.

Listing 4: Aktualisierung des (Bottom) Menüs

```
1     void updateNavigationBarItems(List<MyStudy> studies){
2         navigationView.getMenu().findItem(R.id.menu_questionnaires).setVisible(false);
3         navigationView.getMenu().findItem(R.id.menu_sounds).setVisible(false);
4         navigationView.getMenu().findItem(R.id.menu_education).setVisible(false);
5         for(MyStudy study : studies){
6             QuestionnairesActivity.StudyNames studyName =
                    QuestionnairesActivity.StudyNames.valueOf(study.study.attributes.name);
7             switch (studyName){
8                 case TrackYourTinnitus:
9                     navigationView.getMenu().findItem(R.id.menu_questionnaires).setVisible(true);
10                    break;
11                case Sounds:
12                    navigationView.getMenu().findItem(R.id.menu_sounds).setVisible(true);
13                    break;
14                case Education:
15                    navigationView.getMenu().findItem(R.id.menu_education).setVisible(true);
16                    break;
17                default:
18                    break;
19            }
20        }
21    }
```

In Listing 5 wird die Implementierung, der Klasse *StudyListAdapter*, aufgezeigt. Innerhalb dieser Arbeit wurden mehrere *ListAdapter* eingeführt und implementiert. Diese Klassen erben von der Android-Klasse *ArrayAdapter* (Zeile 1) und dienen dazu, Listen zu organisieren und zu visualisieren. Hier wird im Konstruktor zuerst die Layout Ressource, eine einzelne *study_row* einer Studie definiert (Zeile 7). In dieser *study_row* wurde die grundlegende Struktur eines einzelnen Listenelements für die Studienliste festgelegt. Ein *study_row*-Element ist dabei aus mehreren einzelnen Visualisierungselementen zusammengesetzt. Ein Studienelement der Liste besteht aus zwei *CoordinatorLayouts*[38] [34]. Eines erstellt den oberen Teil der *Row* und enthält einen *TextView* für den Titel der Studie, einen *ImageView* zur Visualisierung der Teilnahme an der Studie, sowie ein Bild als Hintergrund. Das andere *CoordinatorLayout* kreiert den unteren Block und besteht aus zwei *Buttons*. Ein *Button* öffnet die Studien Details mit Hilfe der *StudyDetailActivity*, der andere Button ermöglicht die Teilnahme an einer Studie bzw. das Austreten aus einer beigetretenen Studie. Die *StudyDetailActivity* ist eine visuelle

[38] CoordinatorLayout: https://developer.android.com/reference/androidx/coordinatorlayout/widget/CoordinatorLayout [01.10.2020]

Erweiterung der *StudiesActivity* und funktioniert analog zu dieser, daher wird diese aus Platzgründen nicht ebenfalls aufgelistet. In der Funktion *getView* (Zeile 14) wird ein solches Listenelement, mit Hilfe eines *LayoutInflater*[39] [34] und auf Grundlage des *study-row_Layouts* aufgebaut. Nachdem ab Zeile 16 die einzelne Studie festgehalten und die einzelnen Bestandteile eines *study_row*-Elements, über die jeweilige *ID* der *Layout Ressource,* zugewiesen wurden, wird im Folgenden über den Studien-Status entschieden (Zeile 32), wie die einzelnen Listenelemente dem Nutzer visualisiert werden sollen. Dabei werden zusätzlich zu den Texten der *Adapter-Buttons* (Teilnahme/Austreten) auch die *ImageView* entsprechend gesetzt und sichtbar gemacht oder nicht. Zum Schluss wird noch das richtige Bild für den entsprechenden Hintergrund, für das obere *CoordinaterLayout* (*clStudyHeader*), gesetzt (Zeile 55ff) und verschiedene *OnClickListener* für die Bedienbarkeit implementiert. Zwischen Zeile 26 und Zeile 31 werden die Funktionen des Ein- und Austretens aus einer Studie, für vom Administrator angelegte Nutzer, deaktiviert. Am Schluss der Klasse wird der zusammengesetzte *customView,* zurückgegeben (Zeile 82). So wird eine Studie nach der anderen abgearbeitet und erstellt.

Listing 5: StudyListAdapter

```
1  public class StudyListAdapter extends ArrayAdapter {
2
3      Context context;
4      SharedPreferences sharedTexts ;
5
6      public StudyListAdapter(Context c, List<MyStudy> studies){
7          super(c,R.layout.study_row,studies);
8          context = c;
9          sharedTexts = c.getSharedPreferences("Texts" + MainManager.local, Context.MODE_PRIVATE);
10     }
11
12     @NonNull
13     @Override
14     public View getView(int position, @Nullable View convertView, @NonNull ViewGroup parent) {
15
16         LayoutInflater inflater = LayoutInflater.from(getContext());
17         View customView = inflater.inflate(R.layout.study_row,parent,false);
18         final MyStudy singleStudy = (MyStudy) getItem(position);
19         final LinearLayout llStudy = (LinearLayout) customView.findViewById(R.id.llStudy);
20         final TextView tvStudyTitle = (TextView) customView.findViewById(R.id.tvStudyTitle);
21         final CoordinatorLayout clStudyHeader =
                       (CoordinatorLayout) customView.findViewById(R.id.clStudyHeader);
22         ImageView ivStudyJoined = (ImageView) customView.findViewById(R.id.ivStudyJoined);
23         Button btnStudyInfos = (Button) customView.findViewById(R.id.btnStudyInfos);
24         final Button btnStudyAdapter = (Button) customView.findViewById(R.id.btnStudyAdapter);
25         Resources res = context.getResources();
26         SharedPreferences appInfo = context.getSharedPreferences("AppInfo", Context.MODE_PRIVATE);
27         String email = appInfo.getString("email","default");
28         if(email.endsWith("local.data")){
29             btnStudyAdapter.setEnabled(false);
30             btnStudyAdapter.setVisibility(View.GONE);
31         }
32         switch(singleStudy.state){
33             case "joined":
34                 btnStudyAdapter.setText(R.string.unsubscribe_study);
35                 ivStudyJoined.setVisibility(View.VISIBLE);
36                 break;
37             case "pending":
38                 btnStudyAdapter.setText(R.string.delete_pending_study);
39                 ivStudyJoined.setVisibility(View.GONE);
40                 break;
41             case "invited":
42                 btnStudyAdapter.setText(R.string.accept_study_invitation);
43                 ivStudyJoined.setVisibility(View.GONE);
44                 break;
45
46             case "none":
47                 btnStudyAdapter.setText(R.string.subscribe_study);
48                 ivStudyJoined.setVisibility(View.GONE);
49                 break;
50             default:
51                 break;
52         }
53
54         tvStudyTitle.setText(singleStudy.study.attributes.title);
55         Drawable d = new BitmapDrawable(context.getResources(),
                 BitmapHelper.getBitmap(context,ImageHelper.getFileName
                     (singleStudy.study.attributes.name,true),25,25,0,0,true));
```

[39] LayoutInflater: https://developer.android.com/reference/android/view/LayoutInflater.html [01.10.2020]

```
56          clStudyHeader.setBackground(d);
57
58          clStudyHeader.setOnClickListener(new View.OnClickListener() {
59              @Override
60              public void onClick(View view) {
61                  StudiesActivity myActivity = (StudiesActivity) context;
62                  myActivity.showInformations(singleStudy,llStudy);
63              }
64          });
65
66          btnStudyAdapter.setOnClickListener(new View.OnClickListener() {
67              @Override
68              public void onClick(View view) {
69                  StudiesActivity myActivity = (StudiesActivity) context;
70                  myActivity.changeStudyState(singleStudy);
71
72              }
73          });
74
75          btnStudyInfos.setOnClickListener(new View.OnClickListener() {
76              @Override
77              public void onClick(View v) {
78                  StudiesActivity myActivity = (StudiesActivity) context;
79                  myActivity.showInformations(singleStudy,llStudy);
80              }
81          });
82          return customView;
83      }
84  }
```

In der Klasse *MainManager* sind mehrere Funktionen implementiert, die jeweils von außen bzw. aus anderen Klassen aufgerufen werden. So auch bei Vorgängen, die verfügbaren Studien betreffend (Listing 6, Listing 7), wie z.B. die *getAllStudies*-Funktion (Listing 6, Zeile 1), welche in der *onCreate*-Funktion der *StudiesActivity* aufgerufen wird. Die Vorgehensweise bei Funktionen, die eine Kommunikation mit dem Server betreffen sind generell alle analog aufgebaut, wobei jeweils ein *ApiConnectionTask* kreiert wird. Beispielhaft ist dies in Listing 6 anhand der Studien zu betrachten. Eine kleine Ausnahme stellt die Funktion *getMyStudiesState* in Zeile 22 dar. Hier werden drei *ApiConnectionTasks* erzeugt und dadurch drei *AsyncTasks* parallel gestarte. Zusätzlich wird jeweils eine Variable *count* gesetzt, welche für eine spätere Überprüfung auf Erfolg und Vollständigkeit implementiert wurde. Ebenfalls wichtig ist die übergebene *doneMethod*.

Listing 6: MainManager Studies Funktionen

```
1     public void getAllStudies() {
2         if (!token.equals("")) {
3             new ApiConnectionTask(this).execute("/api/v1/studies?token=" + token
                + "&limit=1000", "GET", "", "allStudies", "", "Accept-Language", MainManager.local);
4         } else {
5             allStudies = dbHandler.loadAllStudies();
6             myActivity.getAllStudiesCallback(false);
7         }
8     }
9     public void getStudyDetails(){
10        count = 0;
11        onlineSuccess = true;
12        if (!token.equals("")) {
13            for(MyStudy study : allStudies){
14                new ApiConnectionTask(this).execute("/api/v1/studies/"+study.study.id+"?token="
                  +    token    ,    "GET",    "",    "getStudyDetails",    study.study.id,    "Accept-Language",
                  MainManager.local);
15            }
16        } else {
17            allStudies = dbHandler.loadAllStudies();
18            myActivity.getAllStudiesCallback(false);
19        }
20    }
21
22    public void getMyStudiesState() {
23        count = 0;
24        onlineSuccess = true;
25        new ApiConnectionTask(this).execute("/api/v1/my/studies?token=" + token
            + "&limit=1000", "GET", "", "myStudies", "", "Accept-Language", MainManager.local);
26
27        new ApiConnectionTask(this).execute("/api/v1/my/studies/pending?token=" + token
            + "&limit=1000", "GET", "", "myPendingStudies", "", "Accept-Language", MainManager.local);
28
29        new ApiConnectionTask(this).execute("/api/v1/my/invitations?token=" + token
            + "&limit=1000", "GET", "", "myInvitedStudies", "", "Accept-Language", MainManager.local);
30    }
```

Anhand dieser *doneMethod* wird, im *MainManager* in der *done()*-Funktion (Listing 7 Zeile 1), der weitere Programmablauf getriggert. Wir betrachten in Listing 7 lediglich jene Fälle, welche für die Studien-Funktionalität relevanten sind. Das Symbol in Zeile 3 soll verdeutlichen, dass dort eigentlich weitere Funktionen implementiert wurden. Die Fälle zwischen Zeile 48 und 56, sowie die Fälle zwischen Zeile 57 und 68 können jeweils zusammengefasst werden, da diese entweder die klasseninterne Funktion *myStudiesDone* (z.B. in Zeile 55) oder die klasseninterne Funktion *studyActionDone* (z.B. in Zeile 49) aufrufen. Dabei wird schlicht das *Result* aus dem *ApiConnectionTask* und ein neuer Status, bezüglich der Studie, übergeben, um dies später korrekt visualisieren zu können. In der *studyActionDone* wird zusätzlich noch ein *Callback*-Bezeichner übergeben, um anhand von jenem die weitere Abarbeitungsfolge entscheiden zu können, wie z.B. das Löschen einer Studie bei *unsubscribe*. Die anderen Fälle wie *allStudies* (Zeile 7) sind die Namen der zuvor im *ApiConnectionTask* übergebenen *doneMethods*. Im Fall *allStudies* wird ersichtlich, wie bei einer erfolgreichen Internetverbindung (*case 1* (Zeile 9)), die Liste *allStudies* mithilfe des *JsonParsers* gefüllt wird und die nächste Funktion *getStudyDetails* aufgerufen wird. Diese Funktion kommt im Fall *getStudyDetail* in Zeile 22 zurück und triggert den folgenden Programmablauf, wie das Aktualisieren der Infos zur Studie,

sowie den Aufruf zur Abfrage des aktuellen Studienstatus (Zeile 28). Ebenso wird mittels der Variable *count* überprüft, ob für jede Studie die Details geladen werden konnten. Ist keine Verbindung mit dem Server möglich gewesen, wird versucht die Daten aus der lokalen Datenbank zu laden (siehe z.B. Zeile 15).

Listing 7: MainManger done()

```
1  public void done(ApiConnectionTask.Result result) {
2          switch (result.getDoneMethod()) {
3            (...)
4
5              //StudiesActivity
6
7              case "allStudies":
8                  switch (exOrSuccess(result)) {
9                      case 1:
10                         allStudies = JsonParser.getAllStudies(result.getResultString());
11                         getStudyDetails();
12                         break;
13                     case 0:
14                     case -1:
15                         allStudies = dbHandler.loadAllStudies();
16                         myActivity.getAllStudiesCallback(false);
17                         break;
18                     default:
19                         break;
20                 }
21                 break;
22             case "getStudyDetails":
23                 switch (exOrSuccess(result)) {
24                     case 1:
25                         count++;
26                         updateStudyDetails(result.getResultString(),result.getId());
27                         if ((count == allStudies.size()) && (onlineSuccess)) {
28                             getMyStudiesState();
29                         }
30                         else if ((count == allStudies.size()) && (!onlineSuccess)) {
31                             allStudies = dbHandler.loadAllStudies();
32                             myActivity.getAllStudiesCallback(false);
33                         }
34                         break;
35                     case 0:
36                     case -1:
37                         count++;
38                         onlineSuccess = false;
39                         if ((count == allStudies.size())) {
40                             allStudies = dbHandler.loadAllStudies();
41                             myActivity.getAllStudiesCallback(false);
42                         }
43                         break;
44                     default:
45                         break;
46                 }
47                 break;
48             case "myStudies":
49                 myStudiesDone(result, "joined");
50                 break;
51             case "myPendingStudies":
52                 myStudiesDone(result, "pending");
53                 break;
54             case "myInvitedStudies":
55                 myStudiesDone(result, "invited");
56                 break;
57             case "unsubscribeStudy":
58                 studyActionDone(result, "none", "unsubscribe");
59                 break;
60             case "subscribeStudy":
61                 studyActionDone(result, "joined", "subscribe");
62                 break;
63             case "subscribeInviteStudy":
64                 studyActionDone(result, "pending", "subscribeInvite");
65                 break;
66             case "deletePendingStudy":
67                 studyActionDone(result, "none", "delete");
68                 break;
69
70         }
71     }
```

Listing 8 zeigt die Implementierung der in Listing 7 aufgerufenen Funktionen. *MyStudiesDone* bekommt ein *Result* und einen neuen Status übergeben. Lag kein Fehler vor wird die Funktion *fillMyStudyState*, mit dem *ResultString* und dem neuen Status der Studie aufgerufen. Danach wird die Kontrollvariable *count* inkrementiert, womit überprüft wird, ob auch alle drei, in Listing 6 (Zeile 22) gestarteten *Asynctasks*, erfolgreich waren. Nur dann wird dem *getAllStudiesCallback* ein wahrer *Boolean*-Wert übergeben (Zeile 6-8). Hat sich ein Nutzer für eine neue Studie angemeldet oder eine Studie verlassen, dann wird im Programmverlauf die Funktion *studyActionDone* aufgerufen. Wiederum wird zuerst eine Abfrage bezüglich eines Fehlers durchlaufen. Ist alles korrekt, werden die Funktionen *changeMyStudyState,* sowie *studyActionCallback* aufgerufen. *ChangeMyStudyState* verändert den Status einer Studie, je nachdem wie der Nutzer mit dieser Studie interagiert hat.

Listing 8: Weitere MainManager und Studies Funktionen

```
1 public void myStudiesDone(ApiConnectionTask.Result result, String newState) {
2         switch (exOrSuccess(result)) {
3             case 1:
4                 fillMyStudyState(result.getResultString(), newState);
5                 count++;
6                 if ((count == 3) && (onlineSuccess)) {
7                     dbHandler.saveAllStudies(allStudies);
8                     myActivity.getAllStudiesCallback(true);
9                 } else if ((count == 3) && (!onlineSuccess)) {
10                    allStudies = dbHandler.loadAllStudies();
11                    myActivity.getAllStudiesCallback(false);
12                }
13                break;
14            case 0:
15            case -1:
16                onlineSuccess = false;
17                count++;
18                if (count == 3) {
19                    allStudies = dbHandler.loadAllStudies();
20                    myActivity.getAllStudiesCallback(false);
21                }
22                break;
23            default:
24                break;
25        }
26    }
27
28    public void studyActionDone(ApiConnectionTask.Result result, String newState, String callback) {
29        switch (exOrSuccess(result)) {
30            case 1:
31                changeMyStudyState(result.getId(), newState);
32                dbHandler.updateStudyState(result.getId(),newState);
33                if (callback.equals("unsubscribe")) {
34                    deleteQuestionnairesOfStudy(result.getId(),callback);
35                } else {
36                    myActivity.studyActionCallback(true, callback);
37                }
38                break;
39            case 0:
40            case -1:
41                myActivity.studyActionCallback(false, callback);
42                break;
43            default:
44                break;
45        }
46    }
```

5.2. Modul auditorische Stimulation

Die Sounds bzw. die „Sound-Dummies“ werden ebenfalls über die *API* geladen. Erneut werden mithilfe der *GSON*-Klasse aus den *JSON*-Objekten die einzelnen *QuestionSounds* kreiert. Dafür musste, identisch zu den Studien in Kapitel 5.1, zuerst folgende Klasse in Listing 9 definiert werden. Zu beachten ist hierbei, dass es sich nur um die Struktur und den Aufbau der einzelnen Sounds handelt, da die eigentlichen Sounddateien lokal auf dem Gerät gehalten werden. Die Sounds und die zugehörigen Daten werden später über den String *label* in Zeile 15 *gematcht*, um die richtigen Audio-Dateien mit dem richtigen Sound zu verknüpfen.

Listing 9: QuestionSound Model

```
1  public class QuestionSound {
2      public String type;
3      public String id;
4      public Attributes attributes;
5      public Links links;
6
7      public class Attributes {
8          public Content content;
9          public String name;
10         public String elementtype;
11
12         public class Content{
13             public String question;
14             public List<Answers> answers;
15             public String label;
16             public String questiontype;
17             public Values values;
18             public int required;
19
20             public class Answers{
21                 public int value;
22                 public String label;
23             }
24
25             public class Values {
26                 public int min;
27                 public int max;
28                 public int step;
29             }
30         }
31     }
32     public class Links {
33         public String self;
34     }
35 }
```

Der Verständlichkeit halber, wird die weitere Implementierung des Moduls analog anhand des programmtechnischen, schrittweisen Ablaufs betrachtet – vom Auswählen eines Sounds der auditorischen Stimulation, bis zum Abspielen und Bewerten eines Sounds.

Man nehme an, der Nutzer befindet sich bereits im Modul für die auditorische Stimulation. Die *QuestionnairesActivity* listet, zusammen mit dem *QuestionnaireListAdapter,* die einzelnen „Fragebögen“ der aktiven Applikation auf, sofern diese bereits erfolgreich vom Server geladen wurden[40]. Im Modul der auditorischen Simulation ist zum Zeitpunkt der Arbeit jedoch nur ein Fragebogen vorgesehen, wobei weitere Fragebögen jederzeit addiert werden können. Der Fokus dieses Abschnitts liegt somit auf dem „Fragebogen“ der Sounds. Die Bezeichnung als Fragebogen, obwohl dieser eine auditorische Stimulation realisieren soll, ist dadurch bedingt, dass die Sounds und deren struktureller Aufbau, dem einer vorhandenen Fragebogenstruktur der *API* nachempfunden wurde. Über den *OnClickListener* des *QuestionnarieListAdapter* wird die Visualisierung des ausgewählten Fragebogens, durch die *QuestionnarieStructureActivity,* gestartet. Im Folgenden werden einzelne Funktionalitäten besagter *QuestionnarieStructureActivity* erläutert, die beim Aufbau, der

[40] Anmerkung: QuestionnairesActivity und QuestionnaireListAdapter verhalten sich weistestgehend analog zur StudiesActivity und StudyListAdapter

Darstellung und dem Benutzen der Sounds beteiligt sind. Durch den Start der *QuestionnaireStructureActivity* werden in der *onCreate*-Funktion verschiedene Funktionen aufgerufen. Von Belang für die Darstellung der Sounds ist das Triggern der internen Methode *orderSounds* mit einem wahren *boolschen* Wert als Parameter (*orderSounds(true)*). In Listing 10 ist diese Funktion abgebildet.

Zuerst wird die gesamte Liste an einzelnen „Fragen“ (*QuestionViews*) aufgeteilt. Es wird eine Liste mit *Text/Headline* Elementen und eine Liste mit den restlichen „Sound/Frage“ Elementen“ gefüllt (Zeile 7-15), wobei anhand des Frage Typs unterschieden wird. Im Folgenden wird nun die Struktur der Darstellung erzeugt. Zuerst soll eine Überschrift für den vorgeschlagenen Sound erscheinen. Hierfür wird die Liste mit den Textelementen solange durchlaufen, bis das Frageelement mit dem Name *element7*[41] gefunden wurde. Zuvor wurde noch die Liste mit der „neuen Ordnung“ (*newOrder*) und die Liste mit bereits „eingefügten Texten“ (*instertedTexts)* gefüllt. Vorläufig werden alle, bis dahin durchlaufenen, *Text-QuestionViews* in diese Listen geschrieben. Dadurch kann man nach der Schleife, die bereits hinzugefügten *Text*-Elemente aus der *Texte*-Liste löschen (Zeile 28f). Die *newOrder* wird dazu benutzt die neue Ordnung, die entsteht, festzuhalten und wird am Schluss der *allQuestionView* Liste zugewiesen (Zeile 88), wodurch, die bis dahin festgestellte, neue Ordnung gesichert wird. Schließlich kann die gewünschte Struktur korrekt visualisiert werden. Der weitere Ablauf des Aufbaus erfolgt generell analog und kann anhand der Quellcodekommentare verfolgt werden. Es wird wiederholt die entsprechende Liste durchlaufen, welche das nächste gewünschte Element enthält. Vergleichbar wird als nächstes der vorgeschlagene Sound eingesetzt. Soll ein neuer Sound vorgeschlagen werden, wird über die Random-Funktionalität einer *ArrayListe* ein beliebiger Sound aus der *sounds*-Liste gezogen und gesetzt. Soll jedoch kein neuer Sound vorgeschlagen werden, wird in der aktuellen Ordnung bzw. der aktuellen Anzeigeliste (*allQuestionViews*) schlichtweg das erste Element mit dem Frage-Typ *Sound* genommen, da dies der zuletzt vorgeschlagene Sound war (siehe Zeile 32ff). Kein Sound erscheint doppelt in der Visualisierung, es wird ausschließlich die Anordnung angepasst. Dies erreicht man durch ständiges Hinzufügen und Löschen aus den verschiedenen Listen. Ein Beispiel hierfür ist bei den Favoriten ab Zeile 62 zu sehen. Es wird eine temporäre Liste *favoriteSounds* angelegt. Diese wird mit den favorisierten Sounds gefüllt und am Ende komplett der *newOrder*-Liste hinzugefügt. Danach wird die temporäre Liste sofort wieder geleert (Zeile 69 und 70).

Listing 10: Sounds richtig ordnen

```
1  public void orderSounds(boolean changeRecommended){
2          List<QuestionView> newOrder = new ArrayList<>();
3          List<QuestionView> texts = new ArrayList<>();
4          List<QuestionView> sounds = new ArrayList<>();
5          List<QuestionView> insertedTexts = new ArrayList<>();
6
7          for (QuestionView qv:allQuestionView)
8          {
9              if(qv.question.type.equals("Sound")){
10                 sounds.add(qv);
11             }
12             else{
13                texts.add(qv);
14             }
15         }
16
17         // Texte bis zu Recommended füllen
18         for (QuestionView qv:texts)
19         {
20             newOrder.add(qv);
21             insertedTexts.add(qv);
22             if(qv.question.type.equals("elements/headlines")){
23                 if(((QuestionText)qv.question.q).attributes.name.equals("element7")){
24                     break;
25                 }
26             }
27         }
28         texts.removeAll(insertedTexts);
```

[41] *element7* steht für sounds_recommended bzw. den vorgeschlagenen Sound

```
29          insertedTexts.clear();
30
31          // Recommended Sound einsetzen
32          if(changeRecommended){
33              Random random = new Random();
34              int idx = random.nextInt(sounds.size());
35              newOrder.add(sounds.get(idx));
36              sounds.remove(idx);
37          }
38          else{
39              for(QuestionView qv: allQuestionView){
40                  if(qv.question.type.equals("Sound")){
41                      newOrder.add(qv);
42                      sounds.remove(qv);
43                      break;
44                  }
45              }
46          }
47          // texte einsetzen bis favourites
48          for (QuestionView qv:texts)
49          {
50              newOrder.add(qv);
51              insertedTexts.add(qv);
52              if(qv.question.type.equals("elements/headlines")){
53                  if(((QuestionText)qv.question.q).attributes.name.equals("element8")){
54                      break;
55                  }
56              }
57          }
58          texts.removeAll(insertedTexts);
59          insertedTexts.clear();
60
61          //-----------Favourite Sounds einfügen--------------------
62          SharedPreferences soundFavourites =
                  myView.getSharedPreferences("soundFavourites", Context.MODE_PRIVATE);
63          List<QuestionView> favouriteSounds = new ArrayList<>();
64          for(QuestionView qv : sounds){
65              if(soundFavourites.contains(((QuestionSound) qv.question.q).attributes.name)){
66                  favouriteSounds.add(qv);
67              }
68          }
69          newOrder.addAll(favouriteSounds);
70          sounds.removeAll(favouriteSounds);
71          //--------texte einsetzen bis restOfSounds----------------
72
73          for (QuestionView qv:texts)
74          {
75              newOrder.add(qv);
76              insertedTexts.add(qv);
77              if(qv.question.type.equals("elements/headlines")){
78                  if(((QuestionText)qv.question.q).attributes.name.equals("element9")){
79                      break;
80                  }
81              }
82          }
83          texts.removeAll(insertedTexts);
84          insertedTexts.clear();
85          //---------Restliche Sound hinzufügen-----------------------
86          newOrder.addAll(sounds);
87          // Neue Sortierung an AllQuestionViews übergeben
88          allQuestionView = newOrder;
89
90      }
```

Im Folgenden Listing sieht man einen Ausschnitt der *QuestionView*-Klasse. Diese Klasse dient dazu, den richtigen *input_frame*, d.h. die richtige Rahmenstruktur für die jeweiligen Fragetypen zu finden (Zeile 17ff). Im Rahmen dieser Arbeit sind dafür nur der Typ *Slider* sowie der Typ *Sound* von Wichtigkeit, es existieren im Gesamtprojekt allerdings mehrere *input_frame*-Objekte. Weiter enthält diese Klasse eine Liste an *results*, in welcher mögliche Antworten gehalten und dadurch an den Server geschickt werden können.

Listing 11: QuestionView

```
1  public class QuestionView {
2      public Question question;
3      public List<String> results;
4      public long collected_at;
5      public Input_Base input_frame;
6      public View question_frame;
7      public QuestionViewInfo info;
8
9      public QuestionView (Question question, Context c,
                ProgressWatcher progressWatcher, QuestionViewInfo info)
10     {
11         results = new ArrayList<>();
12         this.question = question;
13         this.info = info;
14         if(info != null){
15             info.questionView = this;
16         }
17         switch (question.type)
18         {
19             case "elements/headlines":
20             case "elements/texts":
21             case "elements/pages":
22                 input_frame = null;
23                 break;
24             case "Slider":
25                 QuestionSlider sQ = (QuestionSlider) question.q;
26                 input_frame = new Input_Slider(this,sQ.attributes.content.answers,
                                    sQ.attributes.content.values,c,progressWatcher);
27                 break;
28          (...)
29             case "Sound":
30                 QuestionSound soundQ = (QuestionSound) question.q;
31                 input_frame = new Input_Sound(this, soundQ.attributes.content.answers,
                                    soundQ.attributes.content.values, c,progressWatcher);
32                 break;
33             default:
34                 break;
35         }
36
37     }
```

Listing 12 zeigt die Klasse *Input_Sound*, allerdings wurde die *openRatingDialog*-Funktion (Zeile 81) aus Platzgründen komprimiert, da dort beinahe ausschließlich Funktionalitäten für den Aufbau des Dialogs, nach dem Anhören eines *Sound*s, implementiert wurden. Diese dienen hauptsächlich der ansprechenden Darstellung und können für das technische Verständnis der Funktionalität vernachlässigt werden.

Nach abermaliger Ressourcen- und Variablenzuweisungen wird für das *SoundLayout*, ähnlich einer *study_row* in Listing 5, auch ein *onClickListener* implementiert (Zeile 32). In diesem wird die Funktion *openExoPlayerDialog* der *QuestionnaireStructure* aufgerufen, wobei der geklickte *Sound* als Input für den Audio-Player[42] [43] [34] übergeben wird. In der Methode *checkListened* wird geprüft, ob ein Sound in der aktuellen Sitzung bereits einmal angehört wurde, entsprechend wird dies visualisiert. Komplettiert wird die Klasse durch die Funktionen *checkFavourite* (Zeile 41), *checkAndChangeFavourite* (Zeile 61) für die Visualisierung und das Halten der favorisierten Sounds, sowie die Funktionen *fillResultsWithRating* (Zeile 89) und *isRatingRequierd* (Zeile 95). Die beiden Letzteren sind beim Bewerten der auditorischen Stimulation relevant. So wird zwischen Zeile 89 und

[42] ExoPlayer: https://developer.android.com/guide/topics/media/exoplayer [Zugriff: 02.10.2020]

[43] Developer guide ExoPlayer: https://exoplayer.dev/hello-world.html [02.10.2020]

92 der Zeitpunkt und der Bewertungswert auf dem *Slider* mithilfe der *setActionState*-Funktion übermittelt. Diese Funktion wird in Bezug auf die auditorische Stimulation und deren Datenerfassung, anhand der Interaktion eines Nutzers, wiederholt benutzt. Implementiert ist diese Funktion in der Klasse *Input_Base*, von welcher *Input_Sound* erbt. Es wird eine Liste aus *Strings* mit dem übergebenen Aktionsstatus und einem Wert gefüllt. Im Normalfall wird kein Wert übergeben (*value* = null), wodurch der übergebene Aktionsstatus und der Zeitstempel der Aktion in die *String*-Liste geschrieben werden. Ausnahmen stellen die *ActionStates soundRating*, *volume* und *volumeDecibel* dar. Bei *soundRating* wird der angegebene Wert auf dem Slider übertragen und die Werte zu *volume* bzw. *volumeDecibel* repräsentieren die entsprechend aktuelle Lautstärke. Die einzelnen *ActionStates* sind in Abbildung 11 zu sehen und sollten durch ihre Benennung trivial zu verstehen sein. Es sollte somit bei der Datenauswertung möglich sein, die einzelnen Interaktionen eines Nutzers nachvollziehen zu können. Die Funktion *isRatingRequired* (Zeile 95) prüft, ob noch eine Bewertung vom Nutzer benötigt wird oder nicht. Dafür wird die Liste der *results* der *QuestionView* rückwärts durchlaufen und geprüft, ob als letztes ein *play* oder ein *soundRating* geschrieben wurde. Wird zuerst ein *soundRating* gefunden benötigt man keine erneute Bewertung. Findet man zuerst ein *play,* wird der Nutzer zum Bewerten aufgefordert. Die beiden *Favourite*-Funktionen dienen dazu die Favoriten des Nutzers zu erkennen, entsprechend zu markieren (Zeile 41) und die Favoriten neu zu setzen (Zeile 61). In *checkAndChangeFavourite* wird über die *setActionState*-Funktion festgehalten, wann der Nutzer welchen Sound als Favorit markiert (*favSet*) oder gelöscht (*favUnset*) hat.

Listing 12: Input_Sound

```
1  public class Input_Sound extends Input_Base{
2      List<QuestionSound.Attributes.Content.Answers> answers;
3      QuestionSound.Attributes.Content.Values values;
4      LinearLayout llRatingDialog;
5      SeekBar seekBar;
6      Context c;
7      ImageView sound_favourite;
8      ImageView sound_listened;
9      Input_Base my_Input;
10
11     public Input_Sound(QuestionView questionView,
                  final List<QuestionSound.Attributes.Content.Answers> answers,
                  final QuestionSound.Attributes.Content.Values values,
                  Context c, ProgressWatcher progressWatcher) {
12         this.answers = answers;
13         this.values = values;
14         this.questionView = questionView;
15         this.progressWatcher = progressWatcher;
16         this.c = c;
17         this.my_Input = this;
18         LayoutInflater inflater = LayoutInflater.from(c);
19         View v = inflater.inflate(R.layout.question_sound_layout,null);
20
21         LinearLayout llQuestionSound = (LinearLayout) v.findViewById(R.id.llQuestionSound);
22         ImageView sound_image =(ImageView) v.findViewById(R.id.question_sound_image);
23         sound_listened =(ImageView) v.findViewById(R.id.question_sound_listened);
24         TextView sound_title =(TextView) v.findViewById(R.id.question_sound_title);
25         sound_favourite =(ImageView) v.findViewById(R.id.question_sound_favorite);
26
27         sound_image.setImageResource(ImageHelper.getSoundImage(
                ((QuestionSound) questionView.question.q).attributes.content.label));
28         sound_title.setText(((QuestionSound) questionView.question.q).attributes.name);
29
30         checkFavourite();
31
32         llQuestionSound.setOnClickListener(new View.OnClickListener() {
33             @Override
34             public void onClick(View v) {
35                 ((QuestionnaireStructureActivity)c).openExoPlayerDialog(my_Input);
36             }
37         });
38         input_frame = v;
39     }
40
41     public void checkFavourite(){
42         SharedPreferences soundFavourites =
                c.getSharedPreferences("soundFavourites", Context.MODE_PRIVATE);
43         if(soundFavourites.contains(((QuestionSound) questionView.question.q).attributes.name)){
44             sound_favourite.setImageResource(R.drawable.favorite_set);
45         }
46         else{
```

```
47              sound_favourite.setImageResource(R.drawable.favorite_unset);
48          }
49      }
50
51      public void checkListened(){
52          for(String s : questionView.results){
53              if(s.equals(PlaybackStateListener.ActionState.play.toString())) {
54                  sound_listened.setImageResource(R.drawable.icon_listened);
55                  return;
56              }
57          }
58          sound_listened.setImageResource(R.drawable.icon_not_listened);
59      }
60
61      public boolean checkAndChangeFavourite(){
62          SharedPreferences soundFavourites =
                 c.getSharedPreferences("soundFavourites", Context.MODE_PRIVATE);
63          SharedPreferences.Editor editor = soundFavourites.edit();
64          if(soundFavourites.contains(((QuestionSound) questionView.question.q).attributes.name)){
65              editor.remove(((QuestionSound) questionView.question.q).attributes.name);
66              editor.apply();
67              setActionState(PlaybackStateListener.ActionState.favUnset,null);
68              sound_favourite.setImageResource(R.drawable.favorite_unset);
69              return false;
70          }
71          else{
72              editor.putBoolean(((QuestionSound) questionView.question.q).attributes.name,true);
73              editor.apply();
74              setActionState(PlaybackStateListener.ActionState.favSet,null);
75              sound_favourite.setImageResource(R.drawable.favorite_set);
76              return true;
77          }
78
79      }
80
81      public View openRatingDialog(){
82       //just the build, structure and design of the rating dialog
83          llRatingDialog = new LinearLayout(c);
84       //Text assignments, min-,max Values of seekBar(slider), layoutParameters
85       (...)
86          return llRatingDialog;
87      }
88
89      public void fillResultsWithRating(){
90          int progress = seekBar.getProgress();
91          setActionState(PlaybackStateListener.ActionState.soundRatingTimestamp,null);
92          setActionState(PlaybackStateListener.ActionState.soundRating,Integer.toString(progress));
93      }
94
95      public boolean isRatingRequired(){
96
97          for(int i = questionView.results.size()-1; i>=0; i--){
98              if(questionView.results.get(i).equals(
                        PlaybackStateListener.ActionState.soundRating.toString())){
99                  return false;
100             }
101             if(questionView.results.get(i).equals(PlaybackStateListener.ActionState.play.toString())) {
102                 return true;
103             }
104         }
105         return false;
106     }
107 }
```

Bis auf die *checkFavourite*-Methode werden alle Funktionen der *Input_Sound*-Klasse aus der *QuestionnaireStructureActivity* getriggert. Im Folgenden werden die einzelnen, für die auditorische Stimulation relevanten, Funktionen der *QuestionnarieStructureActivity* betrachtet.

Klickt der Nutzer auf einen Sound, so wird über den *OnClickListener* der *Input_Sound*-Klasse die Funktion *openExoPlayerDialog* der *QuestionnaireStructureActivity* aufgerufen. In Listing 13 ist diese Funktion einzusehen.

Zuerst wird dem *Input_Base currentSound* der übergebene *chosenSound* zugewiesen und zwei *ActionStates* gesetzt. Zum einen ein *open* für das Öffnen des Dialogs (Zeile 4) und danach ein *favIsSet,* sollte der Sound als Favorit markiert sein oder, im anderen Fall, ein *favIsNotSet*. Dies soll später der leichteren Auswertung der erhobenen Daten dienen. Weiter wird auch dem *PlaybackStateListener* und dem *deviceListener* der übergebene *chosenSound* zugewiesen (Zeile 14f). Durch den

PlaybackStateListener werden bestimmte Player-Zustände erkannt und festgestellt, wodurch bestimmte *ActionStates* zum aktuellen Sound gesetzt werden. Der *DeviceListener* ist dafür da, zu registrieren, ob sich die Lautstärke bei der Wiedergabe verändert hat. Ist dies der Fall werden über die *setActionState*-Funktion direkt die entsprechenden Zustände plus Werte oder Zeitstempel gesetzt. Anschließend werden das Layout und das Aussehen des Player-Dialogs gebaut. Ab Zeile 24 (bis Zeile 56) wird geprüft, ob der Nutzer Kopfhörer benutzt, sowie der entsprechende *ActionState* gesetzt (*headphoneOff* oder *headphoneOn*). Bei Nichtbenutzung von Kopfhörern wird der Nutzer zusätzlich, durch ein einmaliges PopUp (siehe Zeile 157) und Texte, darauf hingewiesen dies zu tun.

Danach folgen die verschiedenen *OnClick*-Funktionen für die einzelnen Buttons im Audio-Player. In Zeile 58 für die Sound wiederholen Funktion, in Zeile 67 für den Stopp Knopf, in Zeile 100 für die Favoriten-Funktionalität und in Zeile 120 für die Zurück-Taste. Der Wiederholen-Knopf pausiert den Player und startet den *RepeatDialog (*Listing 17). Klick auf Stopp lässt den Player stoppen, setzt den Audio-Player auf den Startzustand zurück und resettet die Media-Ressource. Geprüft wird erstens, ob der Player bereits am Wiedergeben war, wodurch ein *ActionState stop* gesetzt wird und zweitens, ob eine Bewertung vom Nutzer abgefragt werden soll. Der *onClickListener* des Favoriten Button setzt oder entfernt die entsprechende Markierung bei diesem Sound. Hier wird die *Input_Sound*-Funktion *checkAndChangeFavourite* aufgerufen, um das gewünschte Ziel zu erreichen. Der Zurück Button ruft zuerst die klasseninterne Funktion *releasePlayer* (Implementierung siehe Listing 21) auf. Dadurch wird vor dem *ActionState close* zusätzlich, über den *PlaybackStateListener*, ein *ActionState pause* gesetzt. Somit wird sichergestellt, dass der *ActionState stop* wirklich nur dann geschrieben wird, wenn der Nutzer mit dem Stopp-Button interagiert hat. Zusätzlich wird über *checkListened* (Zeile 125) die „bereits gehört-Markierung“ gesetzt. Ebenso wird geprüft, ob eine Bewertung benötigt wird. Ansonsten wird der der *currentSound* genullt, der Audio-Player zurückgesetzt und der gesamte Dialog geschlossen, damit der Nutzer auf die Ansicht der verschiedenen Sounds zurückkehrt.

Zwischen Zeile 113 und 119 werden Daten wie Titel, Bild, Dauer, etc. zugeordnet, um diese im *PlayerDialog* entsprechend anzeigen zu können. Am Ende wird der zuvor zusammengebaute Dialog mit Hilfe der *AlertDialog*[34] Klasse gebaut und angezeigt (Zeile 145 - 155).

Listing 13: ExoPlayerDialog

```
1  public void openExoPlayerDialog(Input_Base chosenSound){
2       playdialogopen = true;
3          currentSound = chosenSound;
4          currentSound.setActionState(PlaybackStateListener.ActionState.open,null);
5          SharedPreferences soundFavourites =
                myView.getSharedPreferences("soundFavourites", Context.MODE_PRIVATE);
6          if(currentSound.questionView.results.size()==2){
7              if(soundFavourites.contains(
                    ((QuestionSound) currentSound.questionView.question.q).attributes.name)){
8                  currentSound.setActionState(PlaybackStateListener.ActionState.favIsSet,null);
9              }
10             else{
11                 currentSound.setActionState(PlaybackStateListener.ActionState.favIsNotSet,null);
12             }
13         }
14         playbackStateListener.currentSound = chosenSound;
15         deviceListener.currentSound = chosenSound;
16         View customView = getLayoutInflater().inflate(R.layout.player_dialog,null);
17         Resources res = getResources();
18
19         playerView = (PlayerView)customView.findViewById(R.id.video_view);
20         playerView.setControlDispatcher(new SeekDisabler());
21
22         audio_headphone_hint = (TextView)customView.findViewById(R.id.audio_headphone_hint);
23         AudioManager audioManager = (AudioManager) getSystemService(AUDIO_SERVICE);
24         boolean headphoneOn = false;
25         if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
26             AudioDeviceInfo[] devices = audioManager.getDevices(AudioManager.GET_DEVICES_OUTPUTS);
27             for(AudioDeviceInfo device : devices){
28                 switch (device.getType()){
29                     case TYPE_BLUETOOTH_A2DP:
30                     case TYPE_USB_HEADSET:
31                     case TYPE_WIRED_HEADPHONES:
32                     case TYPE_WIRED_HEADSET:
```

```
                    headphoneOn = true;
                 break;
                default:
                    break;
            }
        }
        if(!headphoneOn){
            audio_headphone_hint.setVisibility(View.VISIBLE);
            currentSound.setActionState(PlaybackStateListener.ActionState.headphoneOff,null);
        }
        else{
            currentSound.setActionState(PlaybackStateListener.ActionState.headphoneOn,null);
        }
    }
    else{
        if(!(audioManager.isWiredHeadsetOn()||audioManager.isBluetoothA2dpOn())){
            audio_headphone_hint.setVisibility(View.VISIBLE);
            currentSound.setActionState(PlaybackStateListener.ActionState.headphoneOff,null);
        }
        else{
         headphoneOn=true;
            currentSound.setActionState(PlaybackStateListener.ActionState.headphoneOn,null);
        }
    }

    btnMyExoRepeat = (ImageButton)playerView.findViewById(R.id.my_exo_repeat);
    btnMyExoRepeat.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            player.setPlayWhenReady(false);
            openRepeatDialog();
        }
    });

    ImageButton stop_button = (ImageButton)playerView.findViewById(R.id.exo_stop);
    stop_button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            boolean callRating = false;
            if(player.isPlaying()){
                currentSound.setActionState(PlaybackStateListener.ActionState.stop,null);
            }
            if(player.getCurrentPosition()!= 0){
                callRating = true;
            }
            player.stop(false);
         playWhenReady = false;
            currentWindow = 0;
            playbackPosition = 0;
            resetPlayerMediaSource();
            if(callRating){
                openRatingDialog(false);
            }
        }
    });

    initializePlayer();

    Button btnBack = (Button)customView.findViewById(R.id.btnBack);
    ImageButton btn_favourite = (ImageButton)customView.findViewById(R.id.btn_favourite);

    if(soundFavourites.contains(
         ((QuestionSound) currentSound.questionView.question.q).attributes.name)){
        btn_favourite.setImageResource(R.drawable.favorite_set);
    }
    else{
        btn_favourite.setImageResource(R.drawable.favorite_unset);
    }
    btn_favourite.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if(((Input_Sound)currentSound).checkAndChangeFavourite()){
                btn_favourite.setImageResource(R.drawable.favorite_set);
            }
            else{
                btn_favourite.setImageResource(R.drawable.favorite_unset);

            }
        }
    });

    audioTitle = (TextView)customView.findViewById(R.id.audio_title);
    audioTime = (TextView)customView.findViewById(R.id.audio_time);
    audioImage = (ImageView)customView.findViewById(R.id.audio_image);
    audioTitle.setText(((QuestionSound)currentSound.questionView.question.q).attributes.name);
    audioTime.setText("1:00");
    audioImage.setImageResource(ImageHelper.getSoundImage(
         ((QuestionSound)currentSound.questionView.question.q).attributes.content.label));

```

```
120         btnBack.setOnClickListener(new View.OnClickListener() {
121             @Override
122             public void onClick(View v) {
123                 releasePlayer();
124                 currentSound.setActionState(PlaybackStateListener.ActionState.close,null);
125                 ((Input_Sound)currentSound).checkListened();
126                 if(((Input_Sound)currentSound).isRatingRequired()){
127                     openRatingDialog(true);
128                 }
129                 else{
130                     currentSound = null;
131                     playbackStateListener.currentSound = null;
132                     deviceListener.currentSound = null;
133                     playdialogopen = false;
134                     orderSounds(false);
135                     updateShowList();
136                 }
137                 playWhenReady = false;
138                 currentWindow = 0;
139                 playbackPosition = 0;
140                 playdialogopen = false;
141                 playerDialog.cancel();
142             }
143         });
144
145         AlertDialog.Builder  aDialogBuilder =
                 new AlertDialog.Builder(new ContextThemeWrapper(this,R.style.myLight_Dialog));
146         aDialogBuilder.setView(customView);
147         playerDialog = aDialogBuilder.create();
148         playerDialog.setCancelable(false);
149         if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
150             playerDialog.getWindow().setBackgroundDrawable(
                     getResources().getDrawable(R.drawable.white_background,getTheme()));
151         }
152         else{
153             playerDialog.getWindow().setBackgroundDrawable(
                     getResources().getDrawable(R.drawable.white_background));
154         }
155         playerDialog.show();
156
157      if(headphonePopup && !headphoneOn){
158             headphonePopup = false;
159             showStatus(res.getString(R.string.audio_headphone_hint));
160         }
161     }
```

In den Listings oben wird häufig der *PlaybackStateListener*[44] benutzt. Dessen Implementierung ist in Listing 14 zu sehen. Benutzt wird der *PlaybackStateListener* zum Setzen der *ActionStates* (*Enum* in Zeile 8ff), welche später das Nutzungsverhalten des Nutzers, beim Verwenden des Moduls der auditorischen Stimulation, bestmöglich widerspiegeln soll. Ebenfalls dient der *PlaybackStateListener* als eine Art *Observer* (Überwacher) des *SimpleExoPlayers*[42]. Ändert sich der Zustand des Audio-Players, z.B. von Wiedergabe zu Pause, wird das vom *PlaybackStateListener* bemerkt und entsprechend reagiert. Ebenso wird erkannt, ob der Player gerade etwas abspielt oder nicht und der entsprechende *play ActionState* plus Zeitstempel in den *results* gesetzt (Zeile 33f oder Zeile 36ff). Die Klasse ermöglicht auch die Erkennung, dass ein Sound zu Ende gespielt wurde, wodurch ebenfalls ein entsprechender *ActionState* in die *results*-Liste geschrieben, der Bewertungsdialog geöffnet und die Mediadatei zurückgesetzt wird. Durch letzteres kann der Nutzer den Sound nach einer Bewertung erneut starten.

Listing 14: PlaybackStateListener

```
1 public class PlaybackStateListener implements AnalyticsListener {
2
3     public PlaybackStateListener(SimpleExoPlayer player,Context context){
4         this.player = player;
5         this.context = context;
6     }
7
8     public enum ActionState {open,close,play,pause,stop,soundEnd,
9       soundRatingTimestamp,soundRating,noLoop,loop2,loop5,loop10,loop,
10      headphoneOn,headphoneOff,favSet,favUnset,favIsSet,favIsNotSet,
11      volumeTimeStamp,volume,volumeDecibel
12    }
13
```

[44] Implements AnalyticsListener: https://exoplayer.dev/doc/reference/com/google/android/exoplayer2/analytics/AnalyticsListener.html [20.10.2020]

```
14      SimpleExoPlayer player;
15      Input_Base currentSound;
16      Context context;
17
18      @Override
19      public void onPlaybackStateChanged(EventTime eventTime, int state) {
20          switch (state){
21              case Player.STATE_ENDED:
22                  currentSound.setActionState(ActionState.soundEnd,null);
23                  ((QuestionnaireStructureActivity)context).openRatingDialog(false);
24                  ((QuestionnaireStructureActivity)context).resetPlayerMediaSource();
25                  break;
26              default:
27                  break;
28          }
29      }
30
31      @Override
32      public void onIsPlayingChanged(EventTime eventTime, boolean isPlaying) {
33          if(isPlaying){
34              currentSound.setActionState(ActionState.play,null);
35          }
36          else{
37              if(player.getPlaybackState() == Player.STATE_READY){
38                  currentSound.setActionState(ActionState.pause,null);
39              }
40          }
41      }
42  }
```

In Listing 13 wird in Zeile 89 die Funktion *initializePlayer* aufgerufen. Deren Implementierung in Listing 15 zu sehen ist. Hier wird der Audio-Player initialisiert, weshalb diese Funktion entsprechend oft aufgerufen wird. Neben dem Erzeugen der Instanzen der benutzen Klassen und dem Verknüpfen mit dem *Player* werden dem Player verschiedene *Listener*-Klassen, wie *PlaybackStateListener* und *DeviceListener,* hinzugefügt, um bestimmte Verhaltensweisen und Zustände zu registrieren. Mithilfe des *MediaReceivers* kann später festgestellt werden, ob der Nutzer Kopfhörer oder ähnliche Devices benutzt. Am Ende des Initialisierens wird die klasseninterne Funktion *resetPlayerMediaSource* aufgerufen (Listing 15, Zeile 13). Die Klassen-Variable *playWhenReady* entscheidet, ob der *Player* die Wiedergabe startet oder nicht. Zusätzlich werden die letzte Position und das aktuelle Fenster übergeben. Am Schluss wird versucht die richtige *SoundFile* in den Player zu laden. Ersichtlich wird, wie die *SoundFile* über das Label des *currentSound,* mit dem Label der vom Server geladenen Sounds, im „Fragebogen“ *gematcht* werden.

Listing 15: Player initialisieren und MediaSource resetten

```
1  private void initializePlayer() {
2      player = new SimpleExoPlayer.Builder(myView).build();
3      playerView.setPlayer(player);
4      mr = new MediaReceiver(playerView);
5      myView.registerReceiver(mr,new IntentFilter(AudioManager.ACTION_HEADSET_PLUG));
6      playbackStateListener.player = player;
7      player.addAnalyticsListener(playbackStateListener);
8      player.addDeviceListener(deviceListener);
9
10     resetPlayerMediaSource();
11 }
12
13 public void resetPlayerMediaSource(){
14     player.setPlayWhenReady(playWhenReady);
15     player.seekTo(currentWindow, playbackPosition);
16     try {
17         setupExpPlayer(SoundsHelper.getSoundFile(
             ((QuestionSound)currentSound.questionView.question.q).attributes.content.label));
18     } catch (RawResourceDataSource.RawResourceDataSourceException e) {
19         e.printStackTrace();
20     }
21 }
```

Zusätzlich zu den *initializePlayer* und *resetPlayerMediaSource* Funktionen existieren in der *QuestionnaireStructureActivity* eine *createRawMediaSource*- und eine *setupExoPlayer* Methode, wie in Listing 16 zu sehen ist. Die *setupExoplayer*-Methode wird in der *resetPlayerMediaSource*-Funktion (Listing 15, Zeile 17) aufgerufen und gibt anhand des Labels des Sounds die *rawId* jenes Sounds zurück. Die ebenfalls beteiligte Klasse *SoundHelper,* sowie die *getSoundFile*-Funktion sind bereits in Abbildung 11 zu sehen und werden aufgrund der simplen *get*-Funktion nicht nochmals dargestellt. Mit jener *rawId* wird die *createRawMediaSource* Funktion getriggert (Listing 16, Zeile 17). In der *createRawMediaResource*-Methode wird mit mehreren programmspezifischen Aufrufen die abspielbare Sounddatei generiert und in Zeile 13 zurückgegeben. Mit übergeben wird dabei, neben der Source, auch ein *looping* Parameter, anhand dessen die Anzahl der Wiederholungen des Sounds bestimmt werden.

Listing 16: MediaSource kreieren und Player vorbereiten

```
1     public MediaSource createRawMediaSource(Integer rawId) throws
              RawResourceDataSource.RawResourceDataSourceException {
2         final RawResourceDataSource rawResourceDataSource = new RawResourceDataSource(this);
3         DataSpec dataSpec = new DataSpec(RawResourceDataSource.buildRawResourceUri(rawId));
4         rawResourceDataSource.open(dataSpec);
5
6         DataSource.Factory dataSourceFactory = new DataSource.Factory() {
7             @Override
8             public DataSource createDataSource() {
9                 return rawResourceDataSource;
10            }
11        };
12        MediaSource source = new ProgressiveMediaSource.Factory(dataSourceFactory).
                                      createMediaSource(rawResourceDataSource.getUri());
13        return new LoopingMediaSource(source, looping);
14    }
15
16    public void setupExpPlayer(Integer rawId) throws
                 RawResourceDataSource.RawResourceDataSourceException {
17        player.setMediaSource(createRawMediaSource(rawId),false);
18        player.prepare();
19    }
```

Für den Fall, dass ein Nutzer einen ausgewählten Sound länger abspielen lassen möchte als die standardisierte Zeit von einer Minute, wurde eine *Looping* Funktionalität implementiert. Diese sorgt dafür, dass der Sound so oft wiederholt wird, wie vom Nutzer angegeben.

In Listing 17, Zeile 1 kann man im *Enum* die verschiedenen Intervalle erahnen. Der Nutzer kann den Sound einmal, zweimal, fünfmal, zehnmal oder unendlich oft anhören. Die Ziffern stehen zum Zeitpunkt dieser Ausarbeitung eins zu eins für die Minutenanzahl der Wiedergabe. Klickt der Nutzer den Repeat-Button im *PlayerDialog,* wird die Funktion *openRepeatDialog* aufgerufen. Zuerst werden wieder die einzelnen Views per ID zugeordnet (Zeile 4-12). Danach folgen die einzelnen *OnClickListener* der jeweiligen Buttons. Vorab sei gesagt, dass die Funktionalität für loop5 und loop10 der Übersichtlichkeit halber nicht dargestellt werden, da diese analog zu den restlichen Fällen ablaufen. Wird im geöffneten Auswahl-Dialog (repeat_*dialog*) eine Wiedergabeanzahl ausgewählt, wird der entsprechende *ActionState* gesetzt und die *updateRepeat*-Funktion gestartet (Zeile 21-27). Eine Ausnahme stellt der Cancel-Button dar, wodurch der Auswahl-Dialog geschlossen wird und der Nutzer zum unveränderten *Player* zurückkehren kann. Zum Ende der Funktion wird der Dialog mit seinen Ressourcen und Designs erstellt und die Anzeige getriggert.

Listing 17: Looping: Enum und openRepeatDialog

```
1  public enum LoopingState {noLoop,loop2,loop5,loop10,loop}
2
3  public void openRepeatDialog(){
4          View customView = getLayoutInflater().inflate(R.layout.player_repeat_dialog,null);
5          Resources res = getResources();
6
7          Button btn_no_repeat = (Button)customView.findViewById(R.id.btn_no_repeat);
8          Button btn_repeat_two = (Button)customView.findViewById(R.id.btn_repeat_two);
9          Button btn_repeat_five = (Button)customView.findViewById(R.id.btn_repeat_five);
10         Button btn_repeat_ten = (Button)customView.findViewById(R.id.btn_repeat_ten);
11         Button btn_repeat = (Button)customView.findViewById(R.id.btn_repeat);
12         Button btn_cancel = (Button)customView.findViewById(R.id.btnDialogCancel);
13
14         btn_no_repeat.setOnClickListener(new View.OnClickListener() {
15             @Override
16             public void onClick(View v) {
17                 currentSound.setActionState(PlaybackStateListener.ActionState.noLoop,null);
18                 updateRepeat(LoopingState.noLoop);
19             }
20         });
21         btn_repeat_two.setOnClickListener(new View.OnClickListener() {
22             @Override
23             public void onClick(View v) {
24                 currentSound.setActionState(PlaybackStateListener.ActionState.loop2,null);
25                 updateRepeat(LoopingState.loop2);
26             }
27         });
28       //loop5 and loop10
29         btn_repeat.setOnClickListener(new View.OnClickListener() {
30             @Override
31             public void onClick(View v) {
32                 currentSound.setActionState(PlaybackStateListener.ActionState.loop,null);
33                 updateRepeat(LoopingState.loop);
34             }
35         });
36         btn_cancel.setOnClickListener(new View.OnClickListener() {
37             @Override
38             public void onClick(View v) {
39                 playerRepeatDialog.cancel();
40             }
41         });
42         AlertDialog.Builder  aDialogBuilder =
                new AlertDialog.Builder(new ContextThemeWrapper(this,R.style.myLight_Dialog));
43         aDialogBuilder.setView(customView);
44         playerRepeatDialog = aDialogBuilder.create();
45
46         playerRepeatDialog.setCancelable(false);
47         if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
48             playerRepeatDialog.getWindow().setBackgroundDrawable(
                    getResources().getDrawable(R.drawable.white_background,getTheme()));
49         }
50         else{
51             playerRepeatDialog.getWindow().setBackgroundDrawable(
                    getResources().getDrawable(R.drawable.white_background));
52         }
53         playerRepeatDialog.show();
54     }
```

Hat der Nutzer eine bestimmte Anzahl an Wiederholungen ausgewählt wurde die *updateRepeat*-Funktion aufgerufen. Wie man in Listing 18 (Zeile 1ff) erkennen kann, wird anhand des übergebenen *loopingStates* entschieden, auf welchen Wert die Variable *looping* gesetzt wird. Danach wird der *Player* auf den Startzustand zurückgesetzt (Zeile 17 bis 19), wodurch dieser, nach dem Schließen des Dialogs, wieder von vorne abspielbar ist und dadurch die volle, gewählte Wiederholungsdauer gewährleistet werden kann. Direkt danach wird die Mediadatei neu gesetzt, wodurch auch sichergestellt wird, dass die *MediaSource* entsprechend oft geladen wird und wiedergegeben werden kann. Dies hat den Grund, da bei einer zeitlich begrenzten Wiedergabe, von beispielhaften zwei Minuten, die Sounddatei zweimal in den Audio-Player geladen wird, wohingegen bei einer unendlichen Wiedergabe die interne Schleifenfunktion des Players genutzt und die Sounddatei nur einmal geladen wird. Die Entscheidung, ob die interne Wiederholungsfunktion des Players (Zeile 29f) genutzt wird oder nicht (Zeile 23-28), wird in der zweiten switch-Fallunterscheidung getroffen (Zeile 22). Anschließend wird der Repeat-Button durch die *updateRepeatButton*-Funktion entsprechend aktualisiert. Auch dort wird anhand des *loopingState* differenziert (Zeile 38).

Listing 18: Looping: looping updaten und Button anpassen

```
1 public void updateRepeat(LoopingState state){
2         loopingState = state;
3         switch (state){
4             case noLoop:
5                 looping = 1;
6                 break;
7             case loop2:
8                 looping = 2;
9                 break;
10          //case loop5 and loop10
11            case loop:
12                looping = 1;
13                break;
14            default:
15                break;
16        }
17        playWhenReady= false;
18        currentWindow = 0;
19        playbackPosition = 0;
20        resetPlayerMediaSource();
21        updateRepeatButton();
22        switch (state){
23            case noLoop:
24            case loop2:
25            case loop5:
26            case loop10:
27                player.setRepeatMode(Player.REPEAT_MODE_OFF);
28                break;
29            case loop:
30                player.setRepeatMode(Player.REPEAT_MODE_ONE);
31                break;
32            default:
33                break;
34        }
35        playerRepeatDialog.cancel();
36    }
37
38    public void updateRepeatButton(){
39        if(btnMyExoRepeat!= null){
40            switch(loopingState){
41                case noLoop:
42                    btnMyExoRepeat.setImageResource(R.drawable.icon_no_repeat);
43                    break;
44                case loop2:
45                    btnMyExoRepeat.setImageResource(R.drawable.icon_repeat_two);
46                    break;
47              //case loop5 and loop10
48                case loop:
49                    btnMyExoRepeat.setImageResource(R.drawable.icon_repeat);
50                    break;
51                default:
52                    break;
53            }
54        }
55    }
```

Stoppt ein Nutzer die Wiedergabe, schließt den Player oder wurde der Sound zu Ende gespielt, wird, sofern die Bedingungen korrekt sind, ein Bewertungsdialog benötigt. Die Implementierung hierfür ist in Listing 19 zu sehen. In Zeile 9 wird die Funktion *openRatingDialog* der *Input_Sound* Klasse aufgerufen. Danach wird in den *OnClickListenern* dafür gesorgt, dass der *PlayerInfoDialog* geschlossen und die Werte für *currentSounds* genullt bzw. entsprechend zurückgesetzt werden. Das *nullen* und zurücksetzen der Werte geschieht nur im Falle des Schließens des Players (*closePlayer* = *true*). Der Unterschied der beiden *onClickListener* besteht darin, dass beim Button *ok* zusätzlich die abgegebene Bewertung des Nutzers festgehalten wird (Zeile 16). Am Ende wird ähnlich wie bei anderen Dialogen, der Dialog mit Hilfe der *AlertDialog*-Klasse erstellt und die Visualisierung getriggert.

Listing 19: Bewertungsdialog

```
1     public void openRatingDialog(boolean closePlayer){
2         View customView = getLayoutInflater().inflate(R.layout.player_info_dialog,null);
3         Resources res = getResources();
4
5         ScrollView sv_player_info_dialog =
                (ScrollView)customView.findViewById(R.id.sv_player_info_dialog);
6         Button btn_ok = (Button)customView.findViewById(R.id.btnDialogAccept);
7         Button btn_cancel = (Button)customView.findViewById(R.id.btnDialogCancel);
8
9         sv_player_info_dialog.addView(((Input_Sound)currentSound).openRatingDialog());
10        btn_cancel.setText(R.string.no_rating);
11        btn_ok.setText(res.getString(R.string.ok));
12
13        btn_ok.setOnClickListener(new View.OnClickListener() {
14            @Override
15            public void onClick(View v) {
16                    ((Input_Sound)currentSound).fillResultsWithRating();
17                    playerInfoDialog.cancel();
18                    if(closePlayer){
19                        currentSound = null;
20                        playbackStateListener.currentSound = null;
21                        deviceListener.currentSound = null;
22                        playdialogopen = false;
23                        orderSounds(false);
24                        updateShowList();
25                    }
26            }
27        });
28        btn_cancel.setOnClickListener(new View.OnClickListener() {
29            @Override
30            public void onClick(View v) {
31                playerInfoDialog.cancel();
32                if(closePlayer){
33                    currentSound = null;
34                    playbackStateListener.currentSound = null;
35                    deviceListener.currentSound = null;
36                    playdialogopen = false;
37                    orderSounds(false);
38                    updateShowList();
39                }
40            }
41        });
42
43        AlertDialog.Builder  aDialogBuilder =
                new AlertDialog.Builder(new ContextThemeWrapper(this,R.style.myLight_Dialog));
44        aDialogBuilder.setView(customView);
45        playerInfoDialog = aDialogBuilder.create();
46
47        playerInfoDialog.setCancelable(false);
48        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
49            playerInfoDialog.getWindow().setBackgroundDrawable(
                      getResources().getDrawable(R.drawable.white_background,getTheme()));
50        }
51        else{
52            playerInfoDialog.getWindow().setBackgroundDrawable(
                      getResources().getDrawable(R.drawable.white_background));
53        }
54        playerInfoDialog.show();
55    }
```

Abschließend wird noch das Ressourcenmanagement des Gerätes bezüglich des Audio-Players betrachtet. Da die Geräteressourcen nicht unnötig belastet und verbraucht werden sollen, soll die Applikation in bestimmten Situationen entsprechend reagieren und Ressourcen, die zuvor benötigt wurden, auch wieder freigeben. Um dies besser verständlich zu machen, wird versucht dies anhand des *Android Activity Lifecycle*[45] [34] (Abbildung 17) zu erklären. Dabei wird mehr oder weniger genau auf die interne Implementierung der einzelnen Funktionen eingegangen. Sämtliche Funktionen des *Activity Lifecycle* werden in den entsprechenden *Activities* mit den benötigten Funktionalitäten überschrieben.

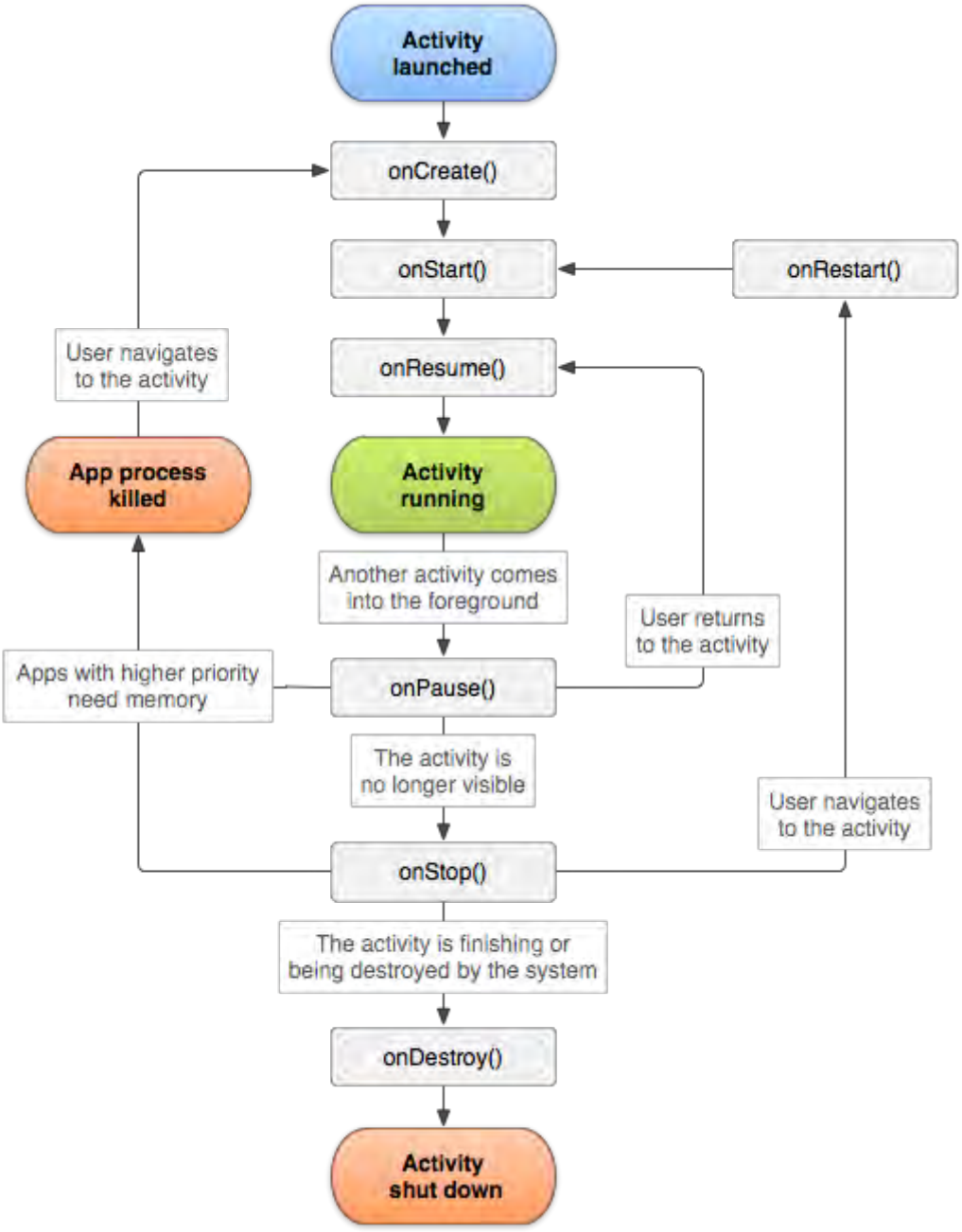


Abbildung 17: Android Activity Lifecycle[46] [34]

[45] Activity-Lifecycle: https://developer.android.com/guide/components/activities/activity-lifecycle [06.10.2020]

[46] Webseite der Bildquelle: https://developer.android.com/guide/components/activities/activity-lifecycle [06.10.2020]

Die *onCreate*-Methode wurde in diesem Kapitel wiederholt erwähnt. Dort werden sämtliche, die entsprechende *Activity* betreffende, Funktionen, Variablen und Objekte implementiert, die beim Anlegen einer solchen Instanz von Bedeutung sind.

Hat ein Nutzer den „Fragebogen“ der Sounds geöffnet wurde als erstes die *onCreate*-Funktion der *QuestionnaireStructureActivity* aufgerufen und entsprechend überschrieben. Direkt danach wird die onStart-Funktion getriggert. Hier wird in der *QuestionnaireStructureActivity* die Funktion *initializePlayer* aufgerufen, sofern der Playerdialog bereits geöffnet ist. Die *onResume* ist für die *QuestionnaireStructureActivity* identisch und *initialisiert* ebenfalls den Audio-Player. Einziger Unterschied hierbei ist die Abfrage nach der Betriebssystemversion, da diese verschiedene Vorgehensweisen beinhalten. Wird die Fragebogenansicht, aus welchem Grund auch immer pausiert, wird die *onPause*-Methode aufgerufen. In dieser wird die Funktion *releasePlayer* gestartet, unter der Bedingung, dass der *PlayerDialog* geöffnet war. Die *releasePlayer*-Methode (Listing 21) gibt die benötigten Ressourcen des *Players,* mithilfe der *SimpleExoPlayer* Funktion *release()*[43]*,* frei. Zunächst wird jedoch geprüft, ob der Player zu diesem Zeitpunkt etwas wiedergegeben hat und gegebenenfalls gestoppt (Listing 21, Zeile 3f). Dies erkennt der *PlaybackStateListener*, ruft seine interne Funktion *OnIsPlayingChanged* auf und setzt den entsprechenden *pause-ActionState*. Für das zurückkehren zur *Activity* über *onResume* werden mehrere Dinge festgehalten. Unter anderem eine Variable, die angibt, ob der *Player* bereit ist zu spielen, sowie die letzte Wiedergabeposition des *Players*. Danach wird der *PlaybackStateListener* genauso entfernt wie der *MediaReceiver* und der *DeviceListener*. *OnStop* verhält sich in diesem Fall programmtechnisch identisch zu *onPause* und ruft die *releasePlayer*-Funktion auf. Analog zu *onPause* und *onStop* verhalten sich *onResume* und *onStart* zueinander. Auch hier wird wieder versucht alle Betriebssystemversionen durch *if*-Abfragen entsprechend abzufangen. Wie in Abbildung 17 zu sehen ist, wird dieser andauernde Lebenszyklus einer *Activity* erst durch das bewusste Beenden oder die Zerstörung der *Activity* durch das Betriebssystem durchbrochen. In diesem Fall wird die *onDestroy*-Methode getriggert. In dieser Arbeit wird dies dazu genutzt, die zuvor erhobenen Daten und Ergebnisse der auditorischen Stimulation an den Server zu senden. Bedingung für eine erfolgreiche Übermittlung ist trivialerweise die Beantwortung aller obligatorischer Fragen durch den Nutzer. Abgesehen davon hat der Nutzer selbst die Möglichkeit die Session zu beenden, indem er auf den entsprechenden Button im „Fragebogen“ klickt. In Listing 20 sind die oben erklärten und überschriebenen *ActivityLifecycle*-Funktionen der *QuestionnaireStructureActivity* für einen kleinen Überblick dargestellt.

Listing 20: QuestionnaireStrucutreActivity Lifecyle Funktionen

```
@Override
    public void onResume()
    {
        super.onResume();
      (...)
        if ((Util.SDK_INT < 24 || player == null) && playdialogopen) {
            initializePlayer();
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        if(manager.getChosenQName().equals("SoundQuestionnaire")) {
            checkAnswersAndSend();
        }
    }

    @Override
    public void onPause()
    {
        super.onPause();
      (...)
        if (Util.SDK_INT < 24 && playdialogopen) {
            releasePlayer();
        }
    }

    @Override
    public void onStop()
    {
        super.onStop();
        (...)
        if (Util.SDK_INT >= 24 && playdialogopen) {
            releasePlayer();
        }
    }
    @Override
    protected void onStart() {
        super.onStart();
        if (Util.SDK_INT >= 24 && playdialogopen) {
            initializePlayer();
        }
    }
```

Listing 21: ReleasePlayer

```
    private void releasePlayer() {
        if (player != null) {
            if(player.isPlaying()){
                player.setPlayWhenReady(false);
            }
            playWhenReady = player.getPlayWhenReady();
            playbackPosition = player.getCurrentPosition();
            currentWindow = player.getCurrentWindowIndex();
            player.removeAnalyticsListener(playbackStateListener);
            player.removeDeviceListener(deviceListener);
            unregisterReceiver(mr);
            player.release();
            player = null;
        }
    }
```

5.3. APIConnection Library

Obwohl zum Stand der Erstellung dieser Arbeit, die selbst implementierte Bibliothek noch nicht verwendet wird, wurden bereits weite Teile der Kommunikation mit der *API* parallel als Library programmiert, um diese im weiteren Verlauf des Projekts, selbst nach Abschluss dieser Arbeit, schrittweise zu integrieren. Während der Entwicklung und Realisierung der Applikation wurde immer mehr bewusst, dass gerade die *API*-Kommunikation ein zentraler Baustein der Applikation ist und bestenfalls in anderen Projekten verwendet werden können soll. Die bereits vorhandenen Bibliothekbausteine sind zum Abschluss dieser Arbeit noch nicht integriert, da die Zeit bis zum Fertigstellen dieser Arbeit, für diese weitreichende Umstrukturierung der Implementierung, sowie eine umfassende Testphase leider nicht ausgereicht hätte. Dies hätte zur Folge, dass die Applikation zum Lieferzeitpunkt möglichweise nicht lauffähig wäre. Nichtsdestotrotz werden in diesem Abschnitt verschiedene Teile der Implementierung dieser Bibliothek für die Serverkommunikation betrachtet und erläutert. Der Einfachheit halber und zum leichteren Verständnis erfolgt dies anhand des Logins.

In Listing 22 sieht man einen Ausschnitt der Klasse *ApiConnection*. Diese Klasse ist der Kern der gesamten Bibliothek. Hier sind sämtliche Variablen, Objekte und Funktionen definiert, welche später für die Funktionalität benötigt und vom Nutzer gebraucht oder benutzt werden. Hier wird beispielsweise eine Basis-URL, ein Kontext und eine Liste angelegt. In diese Liste (*connectionTasks*) werden alle angelegten *ApiConnectionTasks* gefüllt, um bei einem eventuellen Abbruch, alle zusammen, über die Funktion *cancelConnectionTasks* abbrechen zu können. Wird später diese Bibliothek von einem Entwickler genutzt, wird dieser zuerst eine Instanz dieser Klasse anlegen. Dabei übergibt er den Kontext und eine Basis-URL, welche den hier angelegten Variablen im Konstruktor zugewiesen werden. Ansonsten werden im Konstruktor in Zeile 42 sämtliche Instanzen der zuvor deklarierten Objekte angelegt. Will der Entwickler wie in diesem Beispiel die *login*-Funktion benutzen, dann wird er aufgefordert eine Instanz des *LoginHandlers* anzulegen. Dadurch ist es erforderlich die abstrakten Funktionen *onFailure*, welche durch den *FailureHandler* vererbt wird und *onSuccess* zu implementieren, da der *LoginHandler* zusammen mit E-Mail-Adresse und Passwort beim Login-Aufruf übergeben wird. Je nachdem, ob der folgende Serverzugriff erfolgreich war oder nicht wird eine der beiden Funktionen des Handlers beim *Callback* ausgeführt. In diesen Funktionen kann der Entwickler das gewünschte Verhalten seiner Anwendung, beim entsprechenden Fall realisieren. Im Ablauf der *login*-Funktion (Zeile 50) wird anhand der internen Funktion *isActiveNetworkAvailable* (Zeile 28) geprüft, ob das Gerät beim Ausführen der *login*-Funktion eine funktionierende Internetverbindung hat. Besteht diese nicht, wird sofort der *FailureHandler* mit entsprechender Fehlermeldung aufgerufen. War die Prüfung auf eine funktionierende Internetverbindung erfolgreich, wird die Funktionalität in der *login*-Funktion ausgeführt. Zuerst wird eine Instanz der Klasse *ApiConnectionTask*, welche von der Android Klasse *AsyncTask* erbt, kreiert. Übergeben wird der Kontext und die erzeugte *ApiConnection*. Im nächsten Schritt wird der neue *Task* der Liste mit allen *Tasks* hinzugefügt. Danach wird die eigentliche *Request*-Nachricht an die *API* zusammengesetzt, bestehend aus URL, HTTP-Methode, den Login Daten, mittels *JsonParser* geparst, dem *Enum* für die *doneMethod* des Callbacks und einer *ID*. Im Anschluss werden noch die *Header* befüllt, die für eine erfolgreiche *Request* an den Server benötigt werden. Am Ende wird eine Instanz der Klasse *ApiJob* erzeugt, wobei die zusammengebaute *Request* und der übergebene *Handler* als Parameter mitgegeben werden. Der *ApiJob* kann hierbei als eine Art Container verstanden werden, der *Request*, *Result* und *Handler* zu einer Anfrage an den Server zusammenhält. Anschließend wird der anfangs erzeugte *ApiConnectionTask* mit dem *ApiJob* als Parameter ausgeführt (*execute*). Ist keine Netzwerkverbindung möglich wird nichts ausgeführt, da bereits die *OnFailure*-Methode des *FailureHandlers* getriggert wurde.

Die *ApiConnectionTask*-Klasse der Library unterscheidet sich nur marginal zur bisher in dieser Arbeit verwendeten *ApiConnectionTask*-Klasse. Wiederrum wird die *doInBackground*-Funktion aufgerufen, welche den übergebenen Job mit seiner Request abarbeitet. Die Daten werden gleichermaßen via *BufferedOutputStreams* und *BufferedReadern* geschrieben und gelesen. Unterschied ist nur, dass jetzt der Job, bzw. das im Job enthaltene *Result* befüllt und zurückgegeben wird. Danach wird der gesamte *ApiJob,* mit enthaltenem *Result*, in der *onPostExecute* an die *done*-Funktion der *ApiConnection*-Klasse zurückgegeben.

In der *done*-Funktion (Zeile 65) wird zuerst mithilfe der Funktion *exOrSuccess* geprüft, ob eine *Exception* oder ein bestimmter *StatusCode* im *Result* des *ApiJobs* zurückkam. Anhand des Ergebnisses dieser Funktion wird in der *done*-Funktion entschieden, ob die *OnSuccess*-Funktion des (Login)Handlers, mit *StatusCode* und *loginToken* als Parameter, oder die *Failure*-Funktion des *FailureHandlers,* mit Fehlerstatus und Fehlermeldung, aufgerufen wird. Hierbei ist die Ausimplementierung dieser Funktionen dem Entwickler und Nutzer der Library überlassen, wodurch dieser frei entscheiden kann, wie auf die jeweiligen Ereignisse reagiert wird.

Listing 22: Library: ApiConnection

```
1  public class ApiConnection {
2
3      String urlBegin;
4      Context context;
5      List<ApiConnectionTask> connectionTasks;
6    //parallel tasks
7      Map<enumRequests,Integer> counts;
8      Map<enumRequests,Integer> countsToGo;
9      Map<enumRequests,Boolean> onlineSuccess;
10
11     public void cancelConnectionTasks() {
12         for(ApiConnectionTask task: connectionTasks){
13             if(!task.isCancelled()){
14                 task.cancel(true);
15             }
16         }
17     }
18
19     protected enum enumRequests {
20         login,
21         getQuestionnairesOfStudy,
22         getQuestionnairesOfStudyWithStructure,
23         fillQuestionnairesOfStudyWithStructure,
24         getQuestionnaireStructure,
25          /*...*/
26     }
27
28  protected boolean isActiveNetworkAvailable(FailureHandler failureHandler) {
29      ConnectivityManager connectivityManager =
30              (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
31      NetworkInfo networkInfo = connectivityManager.getActiveNetworkInfo();
32      if (networkInfo == null || !networkInfo.isConnected() ||
33              (networkInfo.getType() != ConnectivityManager.TYPE_WIFI
34                      && networkInfo.getType() != ConnectivityManager.TYPE_MOBILE)) {
35
36          failureHandler.OnFailure(-1, "No InternetConnection");
37          return false;
38      }
39      return true;
40  }
41
42     public ApiConnection (Context context,String urlBegin){
43         this.context = context;
44         this.urlBegin = urlBegin;
45         connectionTasks = new ArrayList<>();
46         counts = new HashMap<enumRequests,Integer>();
47         countsToGo = new HashMap<enumRequests,Integer>();
48         onlineSuccess = new HashMap<enumRequests,Boolean>();
49     }
50     public void login(String email, String password, LoginHandler loginHandler){
51         if(isActiveNetworkAvailable(loginHandler)){
52             ApiConnectionTask task = new ApiConnectionTask(context, this);
53             connectionTasks.add(task);
54             Request request = new Request("/api/v1/auth/login", "POST",
                                    JsonParser.setLoginJSON(email, password), enumRequests.login, "");
55             Map<String,String> headers = new HashMap<String, String>();
56             headers.put("Content-Type", "application/json");
57             headers.put("Accept-Language", Locale.getDefault().getLanguage());
58             request.setHeader(headers);
```

```
59              ApiJob job = new ApiJob(request,loginHandler);
60
61              task.execute(job);
62          }
63      }
64  /*...*/
65      protected void done(ApiJob job){
66              switch (exOrSuccess(job)) {
67                  case 1:
68                      switch (job.request.doneMethod) {
69                          case login:
70                             ((LoginHandler)job.handler).OnSuccess(job.result.getStatusCode(),
                                    JsonParser.getLoginToken(job.result.getResultString()));
71                              break;
72                          case getQuestionnairesOfStudy:
73                             ((QuestionnaireHandler)job.handler).OnSuccess(job.result.getStatusCode(),
                                    JsonParser.getMyQuestionnaires(job.result.getResultString()));
74                              break;
75                          case getQuestionnaireStructure:
76                             ((QuestionnaireStructureHandler)job.handler).
                                    OnSuccess(job.result.getStatusCode(),
                                    JsonParser.getQuestions(job.result.getResultString()));
77                              break;
78                       /*...*/
79                          default:
80                              break;
81                      }
82                  case 2:
83                      break;
84                  case 0:
85                      job.handler.OnFailure(job.result.getStatusCode(),job.result.getResultString());
86                      break;
87                  case -1:
88                   /*...*/
89                      break;
90                  case -2:
91                      break;
92                  default:
93                      break;
94              }
95
96      }
97  /*...*/
98      private int exOrSuccess(ApiJob job) {
99          if (job.result.getEx() != null) {
100             return -1;
101         }
102         if ((job.result.getStatusCode() >= 200) && (job.result.getStatusCode() <= 280)) {
103             return 1;
104         }
105         if (job.result.getStatusCode() == 304) {
106             return 2;
107         }
108         if (job.result.getStatusCode() == 401) {
109             if (job.request.doneMethod.equals("login")) {
110                 return 0;
111             }
112
113             return -2;
114         }
115         return 0;
116     }
117 /*...*/
118 }
```

Für das weitere Verständnis sind in Abbildung 18 und den folgenden Listings noch die Wichtigsten, der beteiligten Klassen, oben beschriebener Vorgehensweise dargestellt, sowie die Vererbung der *Handlerklassen* verbildlicht.

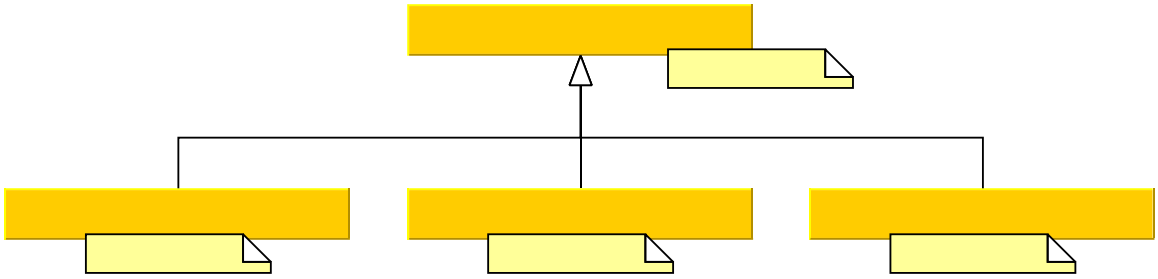


Abbildung 18: UML Klassendiagramm: Vererbung Handlerklassen ApiLibrary

Listing 23: FailureHandler und LoginHandler

```
1 public abstract class FailureHandler {
2    public abstract void OnFailure(int status, String error);
3 }
4
5 public abstract class LoginHandler extends FailureHandler {
6    public abstract void OnSuccess(int statusCode, String token);
7 }
```

Listing 24: ApiJob, Request und Result

```
1  class ApiJob {
2     public Result result;
3     public Request request;
4     public FailureHandler handler;
5
6     public  ApiJob (Request request, FailureHandler handler){
7         this.request = request;
8         this.result = new Result();
9         this.handler = handler;
10    }
11 }
12
13 class Request{
14    public String urlTitle;
15    public String httpMethod;
16    public String jsonData;
17    public ApiConnection.enumRequests doneMethod;
18    public String id;
19    public Map<String,String> header;
20    public String token;
21    public Object dataToFill;
22
23    protected void setHeader(Map<String, String> header) {
24        this.header = header;
25    }
26    public Request(String urlTitle, String httpMethod, String jsonData,
                     ApiConnection.enumRequests doneMethod, String id) {
27        this.urlTitle = urlTitle;
28        this.httpMethod = httpMethod;
29        this.jsonData = jsonData;
30        this.doneMethod = doneMethod;
31        this.id = id;
32        header = new HashMap<String,String>();
33    }
34 }
35
36 class Result{
37    private String resultString;
38    private Exception ex = null;
39    private int statusCode;
40    private String etag;
41
```

```
42      public String getEtag() {
43          return etag;
44      }
45      public void setEtag(String etag) {
46          this.etag = etag;
47      }
48      public String getResultString() {
49          return resultString;
50      }
51      public void setResultString(String resultString) {
52          this.resultString = resultString;
53      }
54      public Exception getEx() {
55          return ex;
56      }
57      public void setEx(Exception ex) {
58          this.ex = ex;
59      }
60      public int getStatusCode() {
61          return statusCode;
62      }
63      public void setStatusCode(int statusCode) {
64          this.statusCode = statusCode;
65      }
66  }
```

6. Vorstellung der Applikation

Dieses Kapitel stellt das *UNITI* Rahmenwerk und das Modul der auditorischen Stimulation aus der Sicht eines Nutzers dar. Zuerst wird der Gesamtaufbau der Applikation *UNITI* beschrieben, danach der Aufbau des internen Moduls. Dabei werden die einzelnen Funktionen der Android Applikation und des Moduls aufgezeigt. Vorab sei gesagt, dass zum Teil verfälschte Daten visualisiert werden. Dies hat den Grund, dass der Server des *UNITI*-Projektes noch nicht vollständig implementiert, bzw. deployed wurde und daher teilweise auf andere, bereits verfügbare, Serverdaten zurückgegriffen wurde, um eine Vorstellung der Applikation zu ermöglichen. Beispielhaft ist dies bei der Visualisierung der Studien (Kapitel 6.1.4) oder der Feedbacks (Kapitel 6.1.5) anhand der Benennungen und den textuellen Inhalten zu sehen. Nichtsdestotrotz wurde versucht das endgültige Design und der gewünschte visuelle Aufbau der Applikation korrekt umzusetzen und hier zu präsentieren. Unterstützt werden jegliche Android Geräte ab Android Version 5.0.

6.1. Vorstellung der Applikation *UNITI*

Ziel der Applikation ist die individuelle Unterstützung von Tinnitus-Patienten, sowie die Erforschung der Krankheit Tinnitus an sich. Zentraler Bestandteil der Applikation ist die Möglichkeit an mehreren Studien zum Thema Tinnitus teilzunehmen. Eine dieser Studien ist jenes Modul der auditorischen Stimulation, wobei die Hauptfunktion die Wiedergabe vorgegebener Sounds ist.

6.1.1. Anmeldung mit Nutzername und Passwort

Um die Applikation nutzen zu können, muss ein Nutzer einen gültigen Account besitzen. Die erste Ansicht nach dem Öffnen ist daher die Login Ansicht, wie sie in Abbildung 19 zu sehen ist. Der Nutzer kann sich nun direkt anmelden, sofern dieser schon einen gültigen Account besitzt. Bestimmten Nutzern wird ein Account vom Administrator zugeteilt. Alle anderen können sich über den *Registrierung Button* direkt ein Konto erstellen oder sich mittels *Anonym teilnehmen* (Abbildung 20 (1)) einen Zufallsaccount generieren lassen. Vorsicht gilt für anonyme Teilnehmer bei einem eventuellen manuellen Logout, wodurch der automatische *Re-Login* deaktiviert wird. Da der anonyme Nutzer generierte Anmeldedaten besitzt, ist ein erneuter manueller Login aufgrund der Komplexität dieser Anmeldedaten äußerst problematisch. Für anonyme Nutzer empfiehlt es sich daher, die generierte E-Mail-Adresse zu notieren und das erzeugte Passwort zu ändern, um einen manuellen Login zu erleichtern. Im gleichen Zuge der anonymen Registrierung wird der Nutzer automatisch eingeloggt und kann die Applikation in vollem Umfang nutzen. Spätestens nach einer erfolgreichen manuellen Registrierung kann sich der Nutzer mit E-Mail-Adresse und Passwort, über Klick auf den *Login 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 E-Mail-Adresse oder Passwort auftreten, wird eine entsprechende Meldung ausgegeben.

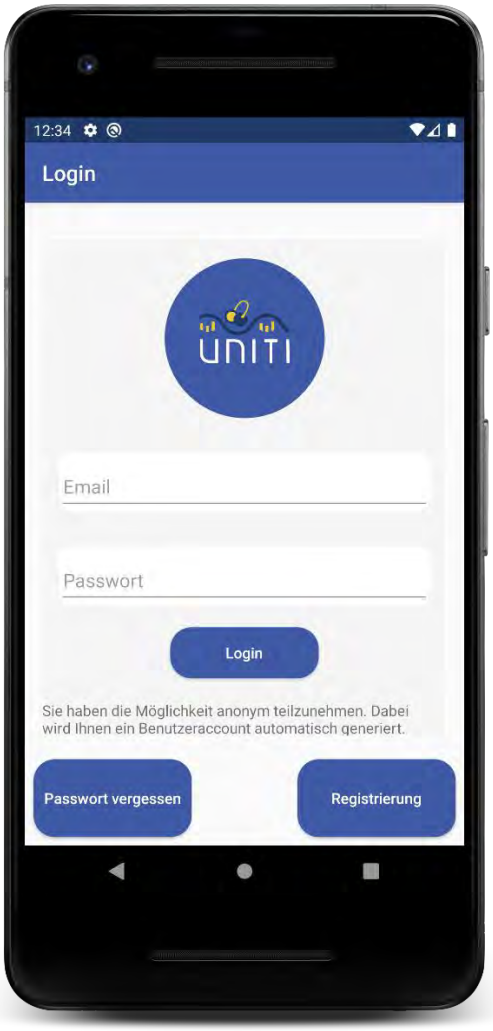


Abbildung 19: Login Ansicht

Nach einer erfolgreichen Anmeldung werden automatisch alle Studien geladen und dem Nutzer, inklusive Teilnahme-Status, visualisiert. Ist der Nutzer bereits in eine oder mehrere Studien eingeschrieben, werden ihm die einzelnen Studien bzw. Module im Bottom-Menü dargestellt und sind dadurch für ihn nutzbar. Bei der erstmaligen Anmeldung ist der Nutzer jedoch standardmäßig in keiner Studie eingeschrieben, wodurch noch keines der Module benutzbar ist. Ausnahme stellen die vorinstallierten Nutzer dar, da diese bestimmten Studien als Teilnehmer zugeteilt sind. Immer aufrufbar sind die Ansichten des Impressums, des Profils, sowie die Ansicht der Feedbacks, wobei die Feedback-Ansicht ohne vorherige Nutzung eines oder mehrerer Module leer ist.

6.1.2. Registrierung

Eine Registrierung kann über dreierlei Wege erfolgen:

- Der Nutzer bekommt vom Administrator einen vorinstallierten Account mit E-Mail-Adresse und Passwort zugeteilt. Eine Registrierung durch den Nutzer selbst ist somit nicht nötig. Der Nutzer kann die Applikation, nach erfolgreichem Login über den zugewiesenen Account, sofort benutzen.
- Der Nutzer kann sich über den entsprechenden Button vollständig anonym registrieren (Abbildung 20 (1)). Dabei werden eine E-Mail-Adresse und ein Passwort automatisiert generiert und dem Nutzer zugewiesen. Es erfolgt ein direkter Login in die Applikation, wodurch der Nutzer diese in vollem Umfang nutzen kann, ohne persönliche Daten preisgeben zu müssen.
- Der Nutzer registriert sich manuell über den Startbildschirm der Applikation. Für die Registrierung muss ein Benutzer seine E-Mail-Adresse und ein Passwort angeben. Visualisiert wird dies, wie in Abbildung 20 (2) zu sehen ist. Um eine falsche Eingabe zu verhindern, müssen die Eingaben wiederholt werden und jeweils übereinstimmen. Beim Klick auf den *Button Registrieren* wird überprüft, ob alle Daten eingegeben wurden. Wenn ja, werden diese an die *API* übertragen. Im Erfolgsfall wird dem Nutzer mitgeteilt, dass eine Bestätigung per E-Mail gesendet wurde. Bei fehlenden Daten, Existenz identischer Registrierungsdaten, fehlender Internetverbindung des Gerätes, sowie bei sonstigem Scheitern der Registrierung, wird eine entsprechende Fehlermeldung ausgegeben. Über *Zurück zum Login* gelangt der Nutzer zurück zur *Login* Ansicht.

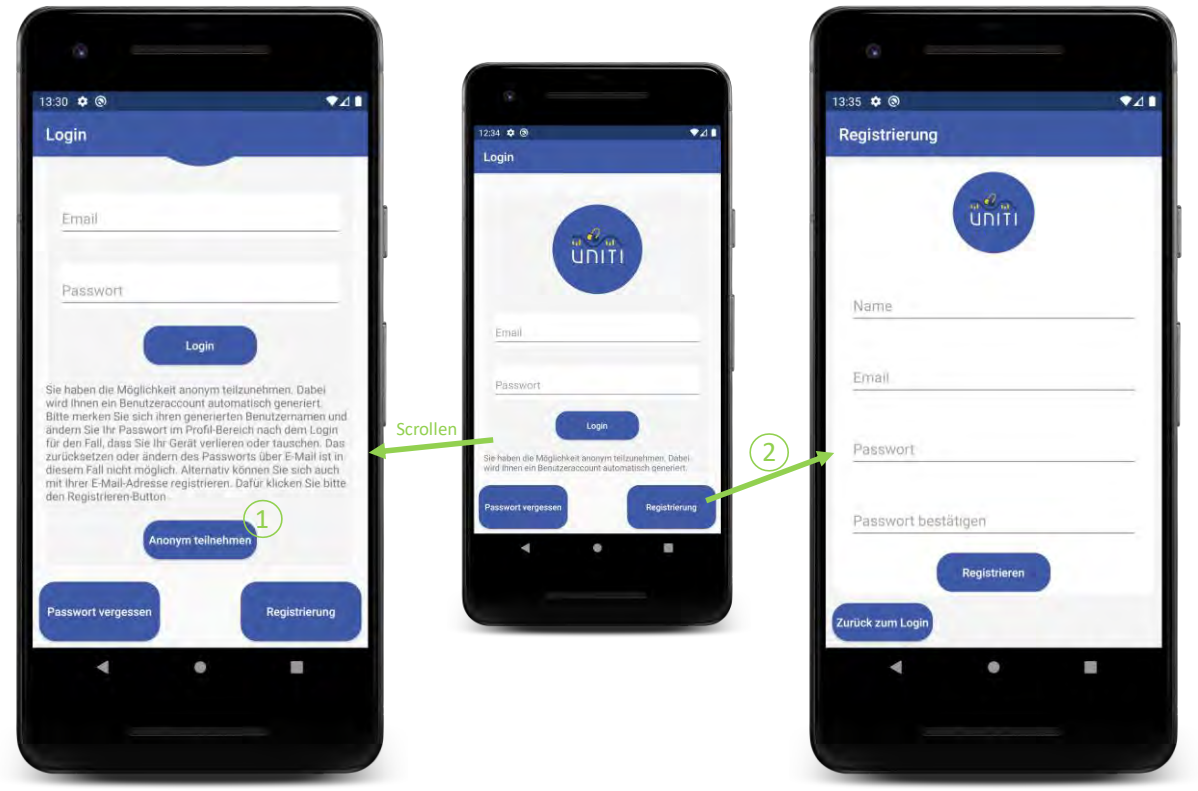


Abbildung 20: Registrierungsmöglichkeiten und Registrierungsansicht

6.1.3. Passwort vergessen / Passwort zurücksetzen

Abbildung 21 zeigt den Ablauf auf, sollte ein Nutzer sein Passwort vergessen haben und infolgedessen zurücksetzen müssen. In der *Login* Ansicht betätigt dieser den *Passwort-vergessen-Button* und erreicht dadurch die *ForgotPasswordActivity*. Dort muss der Nutzer die E-Mail-Adresse des Accounts angeben, dessen zugehöriges Passwort neu gesetzt werden soll. Mit diesem neuen Passwort kann sich der Nutzer danach wieder in der Applikation anmelden. Zur *Login* Ansicht zurück gelangt man über *Zurück zum Login* oder die Android Zurück-Taste.

Anonymen und vorinstallierten Nutzern steht diese Funktionalität nicht zu Verfügung, da diesen eine generierte Dummy-E-Mail-Adresse zur Nutzung der Applikation zugewiesen wurde. Die generierte E-Mail-Adresse eines solchen Nutzers stellt keine existierende elektronische Adresse dar, sondern spiegelt programmtechnisch eine Art Platzhalter einer E-Mail-Adresse wider. Dadurch kann über diese generierten E-Mail-Adressen auch keine E-Mail empfangen werden. Eine eventuelle Zurücksetzung des Passworts ist in bestimmten Fällen nur durch den Support möglich.

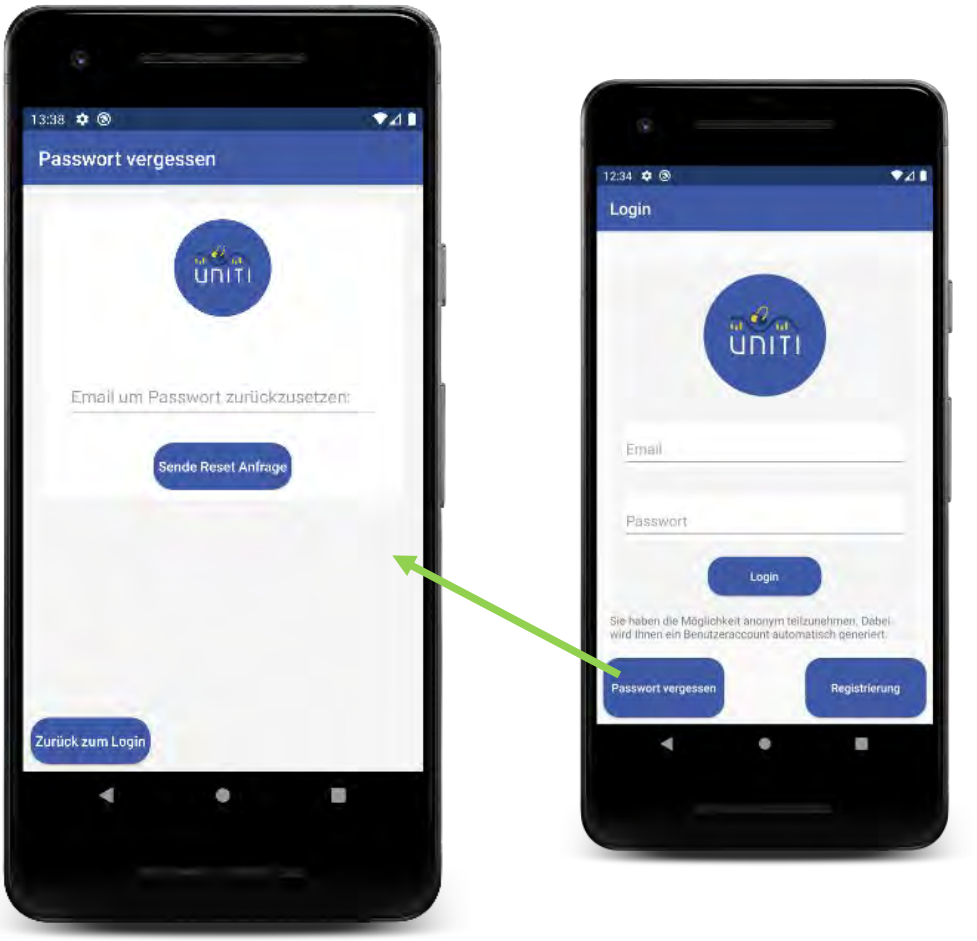


Abbildung 21: Passwort vergessen

6.1.4. Ansicht aller Studien

Nach erfolgreichem *Login* gelangt der Nutzer direkt zur Oberfläche der Studien-Liste, der *StudiesActivity*. Dort werden sämtliche verfügbare, in die Applikation integrierte, Studien aufgelistet. Dem Nutzer wird zusätzlich visualisiert, welchen Studien er bereits beigetreten ist und welchen nicht. Hierbei wird ebenso entschieden, ob und an welchen Studien ein bestimmter Nutzer teilnehmen darf. Die Unterscheidung erfolgt anhand der Art und Weise der Registrierung der Nutzer. Einem vorinstallierten Nutzer ist es nicht möglich Studien beizutreten oder Beigetretene zu verlassen. Erreicht wird dies, indem die Interaktionsmöglichkeiten zum Beitreten und Verlassen einer Studie deaktiviert bzw. entfernt werden. Hintergrund ist die Gewährleistung einer Mindestanzahl an Nutzern je Studie, um eine repräsentative Anzahl an Daten pro Studie zu erhalten. Die restlichen Nutzer können den vorhandenen Studien beliebig bei- und austreten. Die Informationen zu den einzelnen Studien sind jedoch für alle Nutzer gleichermaßen einsehbar. Über jene Bei- bzw. Austritts-Buttons kann der Nutzer an der gewünschten Studie teilnehmen oder diese wieder verlassen. Schreibt sich ein Nutzer für eine Studie ein, wird das Bottom-Menü automatisch um das jeweilige Modul erweitert und aktualisiert. Analog erfolgt dies beim Verlassen einer Studie. Nach dem erfolgreichen Studienbeitritt ist es dem Nutzer möglich zum studienzugehörigen Modul und dessen Aktivitäten zu wechseln oder auf der Studienseite zu verweilen.

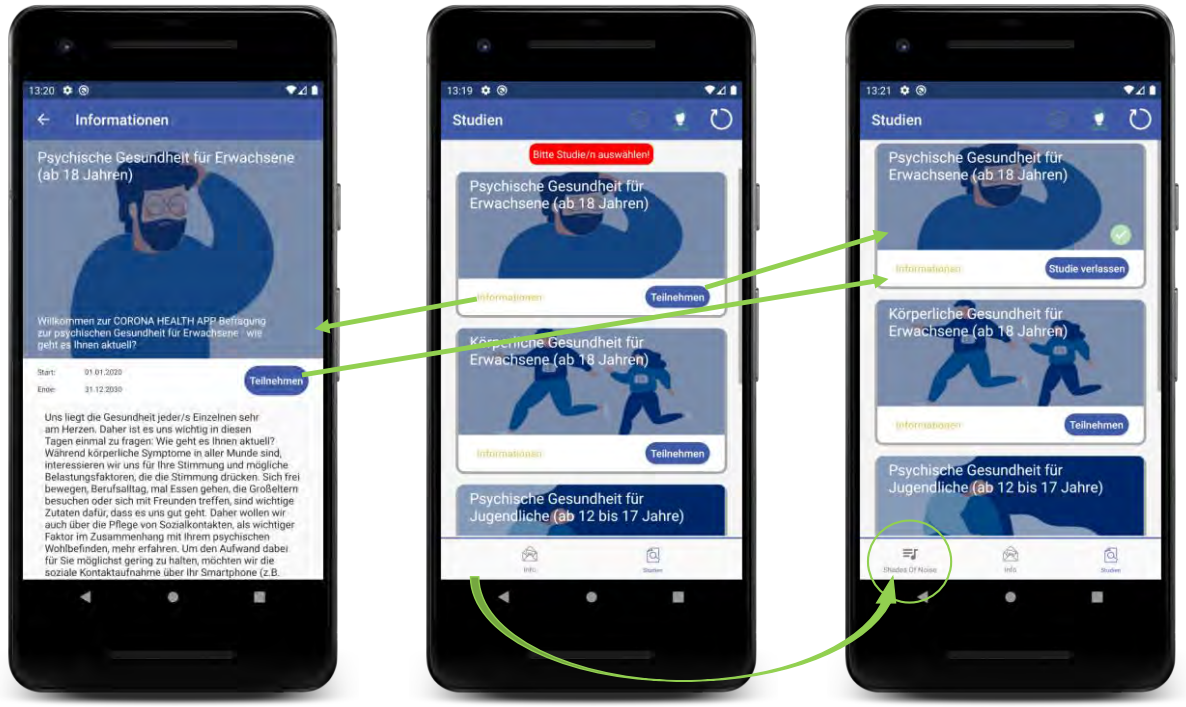


Abbildung 22: Ansicht aller Studien, Studienbeitritt, Studiendetails

6.1.5. Feedback-Tab

Mögliche Ergebnisse bereits ausgewerteter Fragebögen und bearbeiteter Aktivitäten kann der Nutzer im Menü-Unterpunkt *Ergebnisse* einsehen. Visualisiert wird dies wie in Abbildung 23. Es werden alle verfügbaren Ergebnisse untereinander aufgelistet, wobei markiert wird, welche der Feedbacks neu bzw. ungelesen sind. Beschriftet sind die einzelnen Ergebnisse mit dem Namen des Fragebogens. Ferner werden zum einzelnen Ergebnis Datum und Uhrzeit der Abgabe der Antworten des Fragebogens angezeigt. Interagiert der Nutzer mit einem dieser Ergebnisse, dann wird die eigentliche Auswertung des ausgewählten Ergebnisses auf einer neuen Seite angezeigt. Jedes Ergebnis kann in einzelne Kategorien unterteilt und zusätzlich für eine bessere Übersichtlichkeit, je nach Ergebnis, farblich unterschieden werden. Über den internen *Zurück-Pfeil* gelangt der Nutzer wieder zur Übersicht aller Ergebnisse zurück.

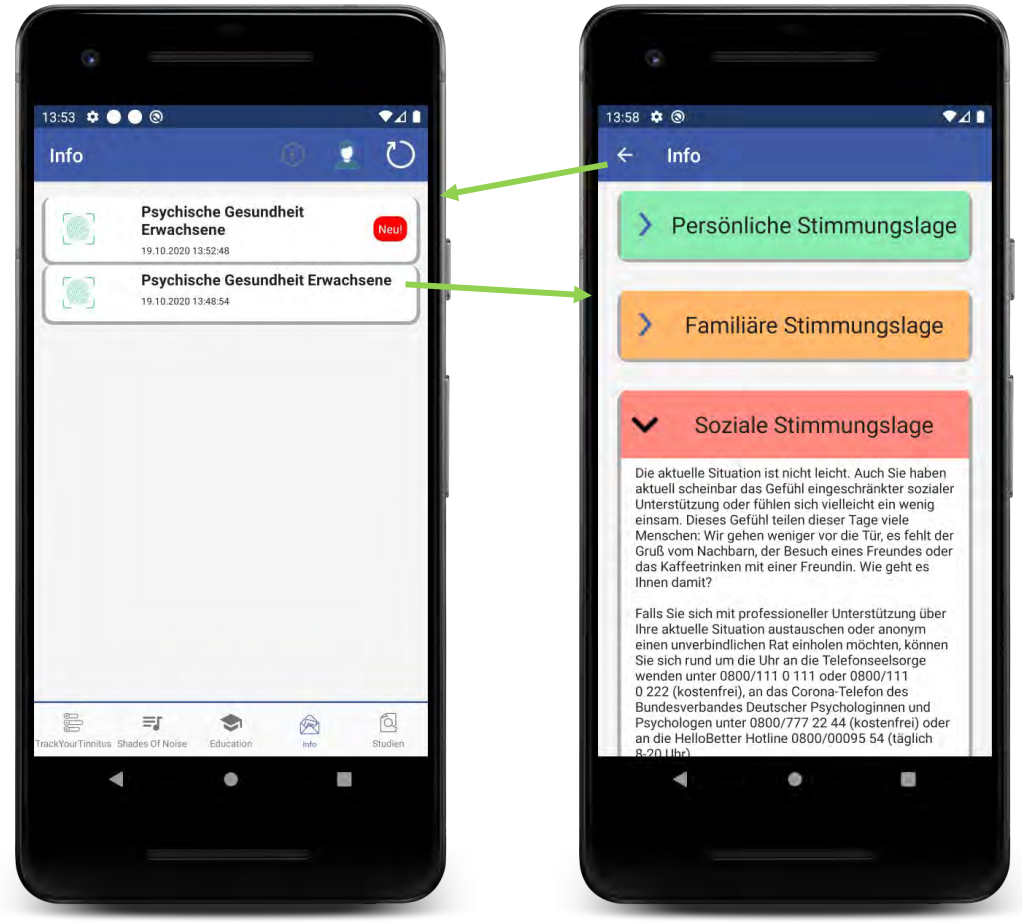


Abbildung 23: Feedback, Ergebnisse, Infos

6.1.6. Profil

Der Menüpunkt *Profil* befindet sich entgegen der einzelnen Modul-Tabs nicht im Bottom-Menü, sondern separat als Button oben rechts in der Applikation. Über diesen Button kann der Nutzer seine persönlichen Profilinformationen aufrufen und einsehen (Abbildung 24). Dem Nutzer ist es hier ebenso möglich sein Passwort individuell zu verändern und neu zu setzen, sowie sich über den entsprechenden Button aus der Applikation auszuloggen. Anonymen Nutzern wird ein Logout nicht empfohlen, da diese generierte und dadurch komplexe Anmeldedaten besitzen. Durch einen manuellen Logout wird der automatisierte Re-Login-Prozess deaktiviert wird. Ein erneuter Login ist nur durch eine korrekte Eingabe valider Daten möglich. Anonymen Nutzern wird daher vor einem Logout empfohlen, die ihnen zugeteilte E-Mail-Adresse zu notieren und das Passwort zu ändern, ansonsten ist es diesen Nutzern nicht mehr möglich, auf deren bereits bestehendes Konto zuzugreifen.

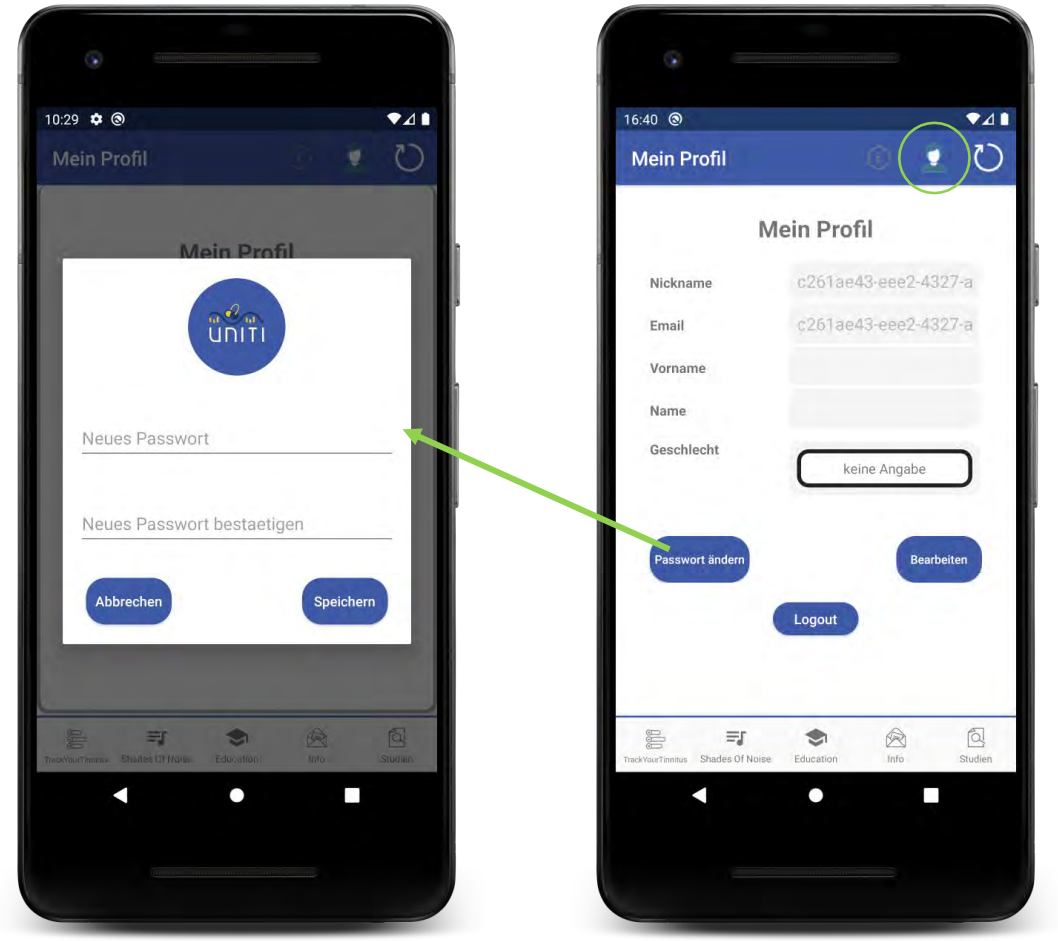


Abbildung 24: Profil und Passwort ändern

6.1.7. Über uns

Analog zum *Profil*-Button findet sich auch der Button für das Aufrufen der Projektinformationen in der rechten oberen Ecke der Applikation wieder. In der *AboutActivity* (Über uns) sind sämtliche Informationen zum Projekt hinterlegt und anschaulich visualisiert. Unterteilt sind diese dabei in die Kategorien Entwicklerteam, Datenschutz, Richtlinienkonformität, Geräte-Nutzungsdaten und Impressum. Diese Ansicht stellt dem Nutzer somit alle Informationen bereit, die für jenen relevant sein könnten und erzeugt somit eine höchstmögliche Transparenz.

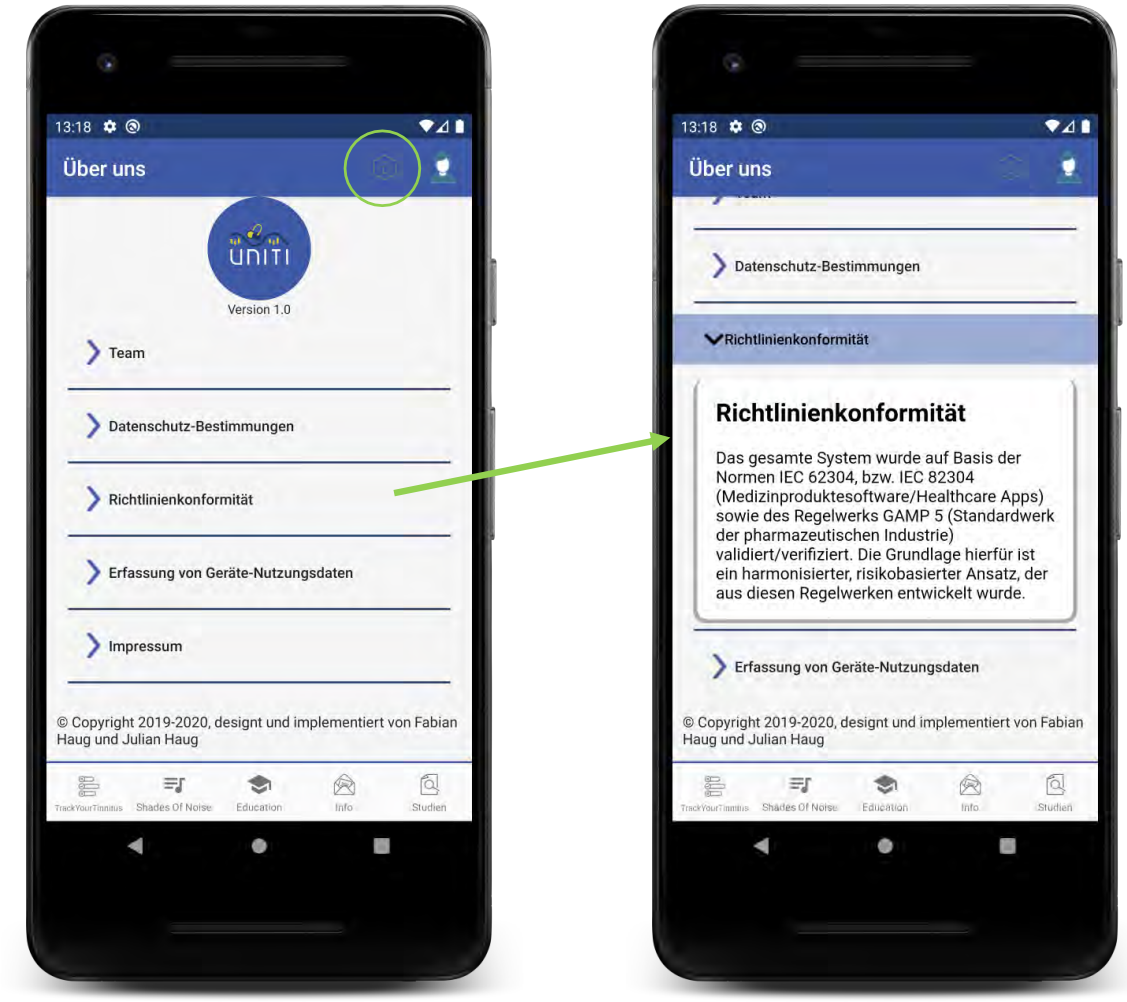


Abbildung 25: AboutUsActivity, Informationen zum Projekt

6.2. Vorstellung des Moduls der auditorischen Stimulation

Ein Modul bzw. eine Studie der *UNITI*-Applikation und gleichermaßen Hauptbestandteil dieser Arbeit ist das Modul der auditorischen Stimulation. Das Modul soll den Studienteilnehmern eine jederzeit verfügbare und unabhängige auditive Stimulation ermöglichen. Hierzu wurde die vorhandene Fragebogenstruktur der *API* benutzt, um lokal gespeicherte, mit der Applikation ausgelieferte, Sounddateien auf dem Gerät wiederzugeben. Ziel ist es, die Beschwerden von Tinnitus-Patienten zu lindern, die Forschung hinsichtlich der Erkrankung vorantreiben und im Idealfall, Rückschlüsse auf die Effektivität einer auditiven Stimulationstherapie ziehen zu können.

6.2.1. Fragebögen und Ansicht aller Sounds

Ist ein Nutzer in die Studie der auditorischen Stimulation eingeschrieben, hat er über das Bottom-Menü Zugriff auf das realisierende Modul. In besagtem Modul werden zuerst alle, der Studie zugehörigen, Fragebögen als Liste visualisiert. Zum Zeitpunkt der Fertigstellung dieser Arbeit ist jedoch nur ein „Fragebogen“ verfügbar. Der Begriff Fragebogen wird hier in Anführungszeichen gestellt, da es sich bei diesem Fragebogen um die Liste der einzelnen Sounds handelt. Fragebogen deshalb, da diese Sounds und deren Darstellung in die Fragebogenstruktur der *API* integriert wurden. Der Nutzer kann eine Sitzung starten, indem dieser auf den *Starten*-Button des gewünschten Fragebogens klickt. Im Falle der Sounds werden diese listenhaft aneinandergereiht. Zuvor wird der Nutzer aufgefordert, zwei obligatorische Fragen zu seinen aktuellen Tinnitus-Beschwerden zu beantworten. Visualisiert wird zusätzlich, welche der Sounds vom Nutzer als Favorit markiert wurden und welche Sounds während der laufenden Sitzung bereits abgespielt wurden. Angeführt wird diese Liste durch einen regelmäßig wechselnden, vorgeschlagenen Sound (Abbildung 26).

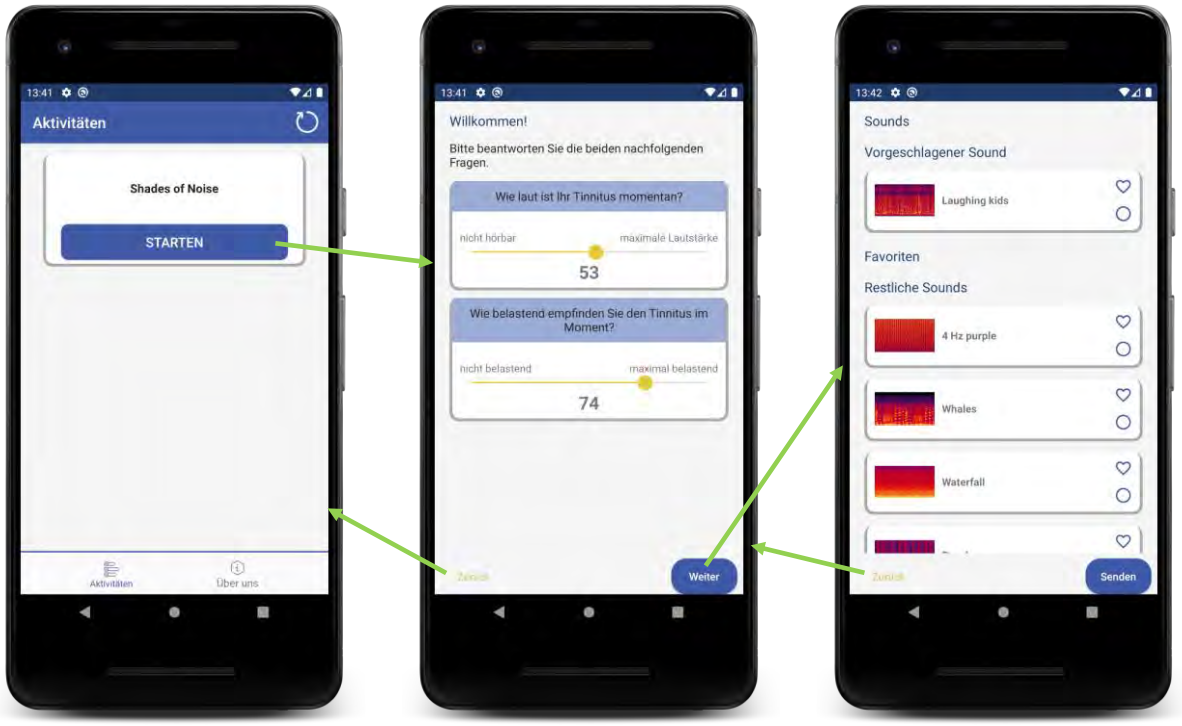


Abbildung 26: Fragebogen und Ansicht aller Sounds

6.2.2. Audio-Player öffnen

Dem Nutzer ist es möglich jeden beliebigen Sound der Liste, durch einfaches Anklicken, auszuwählen und dadurch den Audio-Player zu öffnen. Hat der Nutzer keine Kopfhörer am Gerät angeschlossen, wird er über ein *PopUp* und einen Hinweis im Audio-Player darauf aufmerksam gemacht (vgl. Abbildung 27).

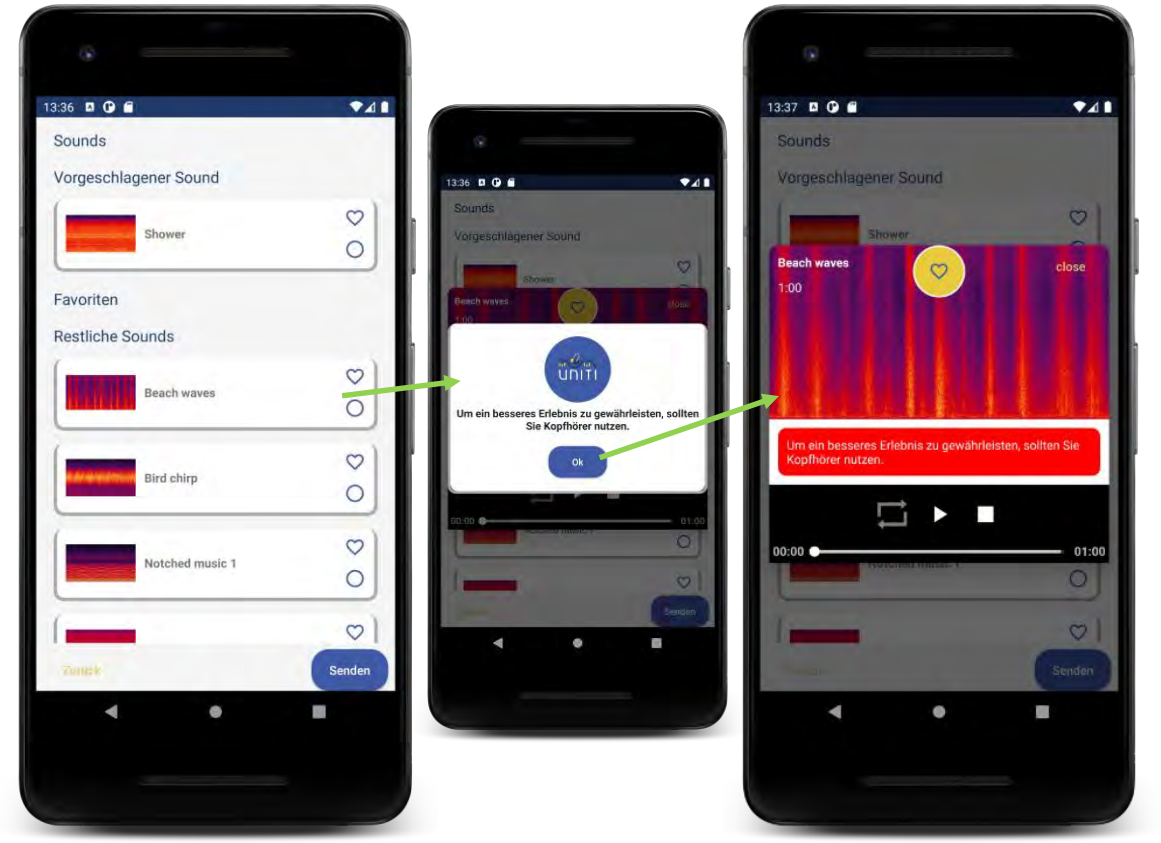


Abbildung 27: Player öffnen ohne Kopfhörer

6.2.3. Play-, Pause- und Favoriten-Funktionalität

Der Nutzer kann den ausgewählten Sound wiedergeben, diesen pausieren und wieder fortsetzen. Zusätzlich gibt es die Möglichkeit den aktuellen Sound zu den Favoriten hinzuzufügen. Im Hintergrund erfolgt die Überwachung, ob Kopfhörer angeschlossen sind oder nicht, wodurch der Hinweis im Player entsprechend gesetzt oder entfernt wird. Im Folgenden werden die unterschiedlichen Aktivitäten und deren getriggerte Folgefunktionalitäten erläutert (vgl. Abbildung 28).

- Play (Wiedergabe): Der ausgewählte Sound wird für die standardisierte Zeitspanne, d.h. eine Minute lang wiedergegeben. Der *Play*-Button wechselt während der Wiedergabe zum *Pause*-Button. Nach einer Minute wird die Wiedergabe beendet und der Bewertungsdialog automatisch geöffnet.
- Pause: Die Wiedergabe des aktuellen Sounds wird pausiert. Der *Pause*-Button wechselt wieder zum *Play*-Button, wodurch der Sound später fortgesetzt werden kann.
- Favoriten: Dem Nutzer steht es jederzeit offen, den aktuellen Sound über den entsprechenden Button zu favorisieren oder wieder aus den Favoriten zu entfernen. Diese Funktionalität erfolgt voll parallel zur Wiedergabe. Jeder favorisierte Sound wird in der Liste aller Sounds der Favoritengruppe hinzugefügt. Analog werden Sounds, die nicht mehr als Favorit markiert sind, aus dieser Sortierung entfernt.

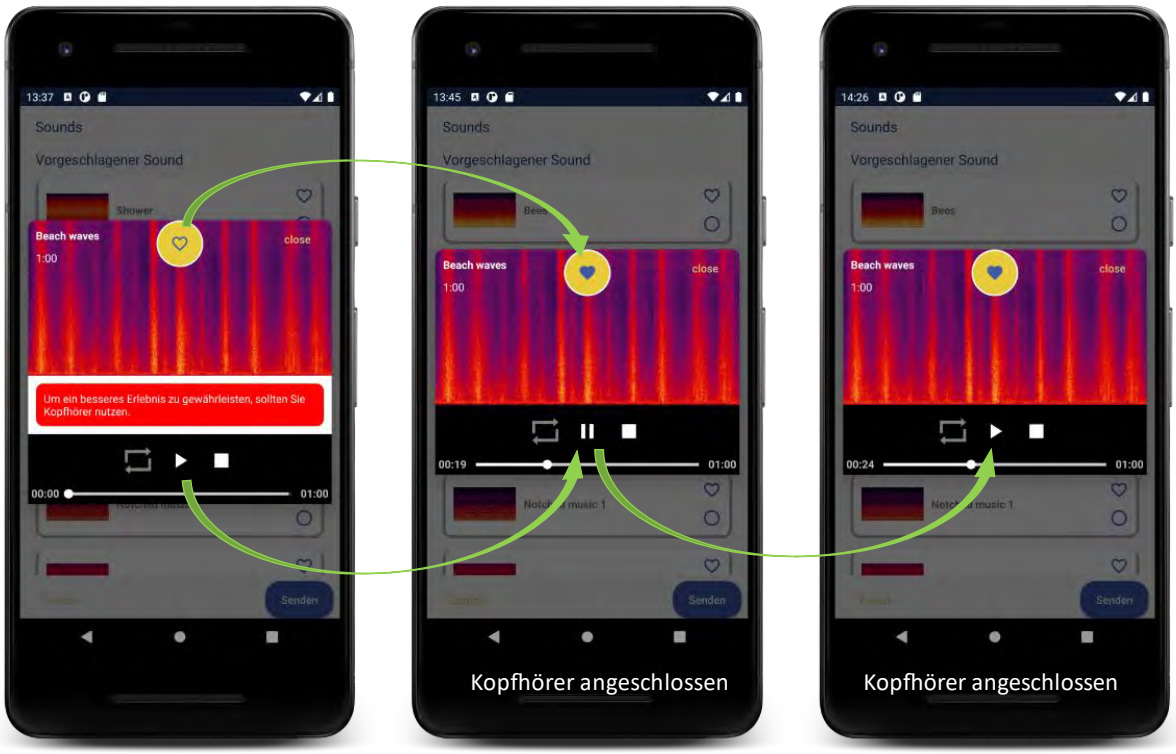


Abbildung 28: Player starten, pausieren und Kopfhörer angeschlossen

6.2.4. Schleifenfunktion

Interagiert der Nutzer mit dem Loop-Button wird zuerst die Wiedergabe des aktuellen Sounds pausiert. Dem Nutzer erscheint ein Auswahldialog, welcher die gewünschte Wiedergabezeit abfragt. Mögliche Zeitspannen sind eine, zwei, fünf und zehn Minuten, sowie eine Wiedergabe auf unbegrenzte Zeit. Die Auswahl einer Minute wurde integriert, damit ein Nutzer, auf Wunsch, wieder zur ursprünglichen Wiedergabezeit zurückkehren kann. Die Auswahl kann ebenso jederzeit abgebrochen werden, wodurch der Nutzer zum Player zurückkehrt und den Sound, an zuvor pausierter Stelle, mittels Play fortsetzen kann. Entscheidet sich ein Nutzer für eine bestimmte Wiedergabezeit, wird der Nutzer ebenfalls zum Audio-Player zurückgeleitet. Allerdings wird der Audio-Player auf den Startzustand zurückgesetzt, um die volle Wiedergabezeit, von beispielhaften fünf Minuten, gewährleisten zu können. Zusätzlich wird die gewählte Wiedergabezeit durch ein Anpassen des *Loop*-Buttons benutzerfreundlich visualisiert. Die Wiedergabe erfolgt mittels Interaktion mit dem *Play*-Button, wodurch der Sound so lange wiedergegeben wird, wie vom Nutzer gewünscht. Nach Ablauf der Zeitspanne stoppt der Audio-Player und der Bewertungsdialog erscheint (vgl. Abbildung 29).

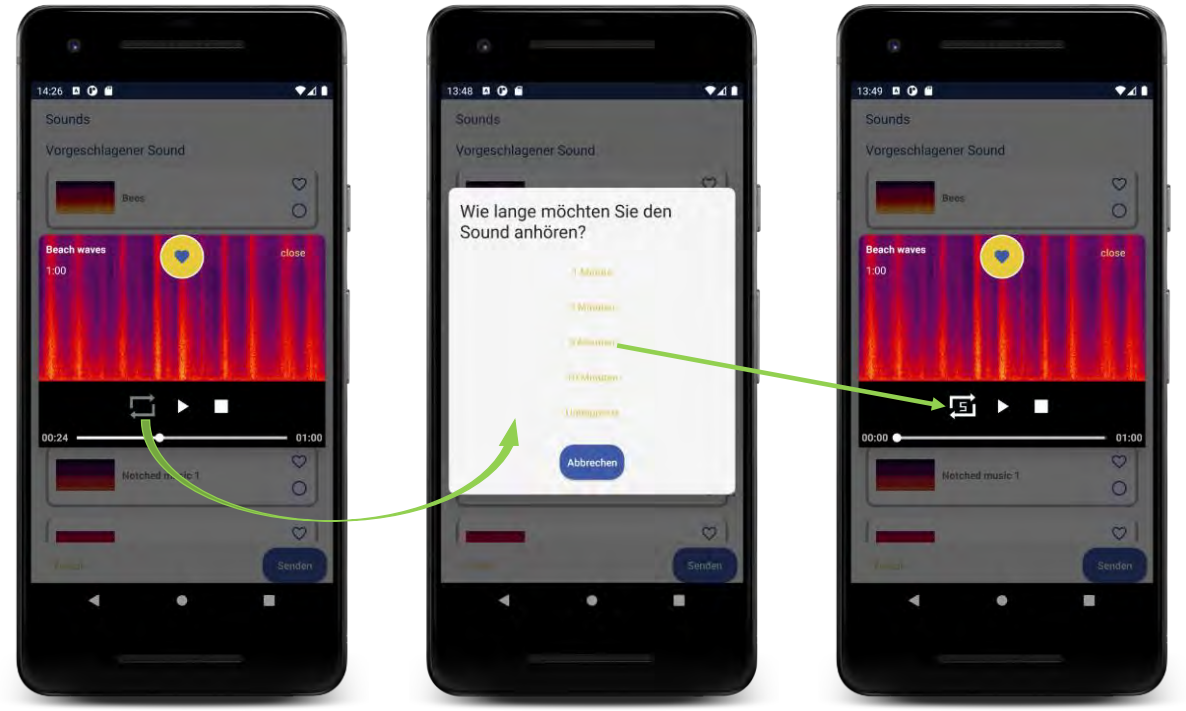


Abbildung 29: Wiederholungsfunktion

6.2.5. Player stoppen, Player schließen und Bewertungsdialog

Der *Stopp*-Button stoppt die Wiedergabe und setzt den Audio-Player auf den Ausgangszustand zurück. Bei erneuter Wiedergabe wird der Sound demnach wieder von vorne gestartet. Wurde der ausgewählte Sound, vor einer Interaktion mit dem *Stopp*-Button, wiedergegeben, erscheint zusätzlich ein Bewertungsdialog. In diesem kann der Nutzer angeben, wie sich jener Sound auf seinen Tinnitus ausgewirkt hat. Dem Nutzer steht es dabei frei, ob er eine Bewertung abgeben möchte oder nicht. Anschließend wird der Bewertungsdialog geschlossen, wodurch der Nutzer zum Audio-Player zurückkehrt (vgl. Abbildung 30).

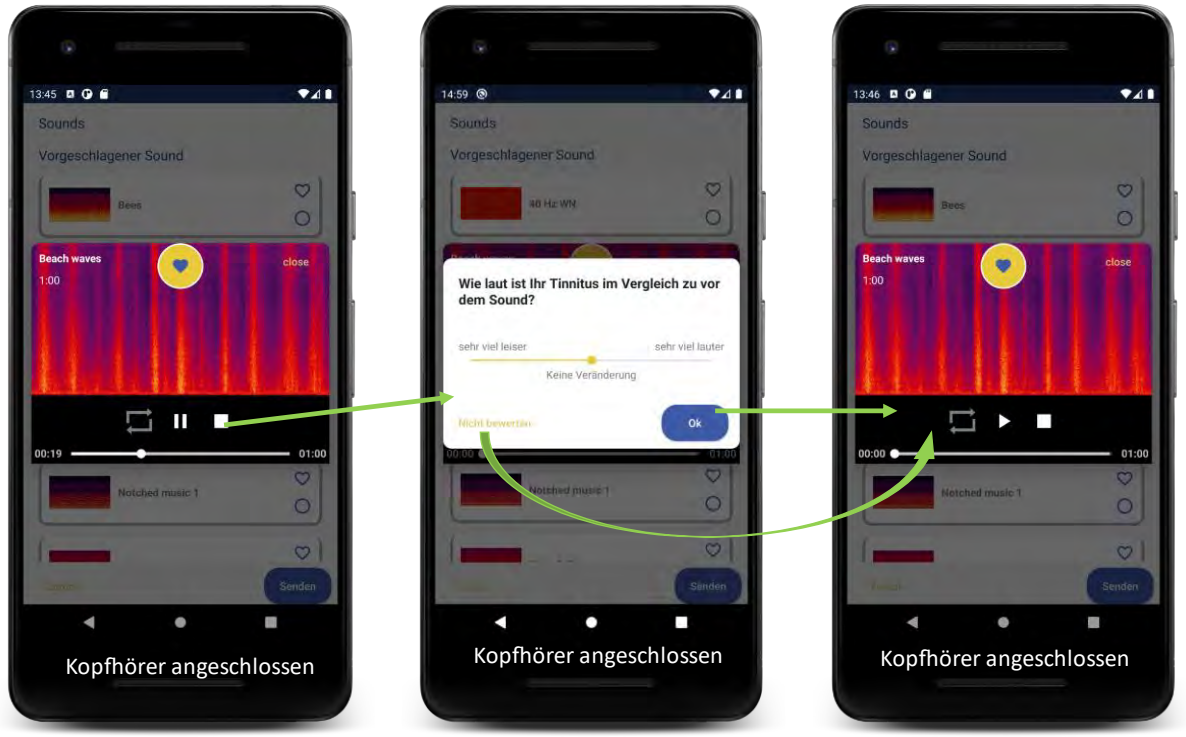


Abbildung 30: Player stoppen und Sound bewerten

Über *close* kann der Nutzer den Audio-Player jederzeit wieder schließen und zur Liste aller Sounds zurückgelangen. Wurde der Sound zuvor oder während der Interaktion mit dem *close*-Button wiedergegeben, erscheint ebenfalls besagter Bewertungsdialog. Sollte der Sound nicht abgespielt worden und nur der Player geöffnet gewesen sein, wird trivialerweise keine Bewertung des Sounds erbeten (vgl. Abbildung 31).

Mit Hilfe des Bewertungsdialogs soll die Effektivität der Sounds, bezüglich der Tinnitus-Beschwerden der Patienten, ermittelt werden. Befragt wird, wie positiv oder wie negativ, sich kurz zuvor wiedergegebener Sound, auf den individuellen Tinnitus ausgewirkt hat. Als Eingabemöglichkeit wird ein Slider (Schieberegler) bereitgestellt, welcher von *sehr viel leiser* über *keine Veränderung* bis zu *sehr viel lauter* reicht. Dem Nutzer steht es dabei frei, ob er eine Bewertung abgeben möchte oder nicht.

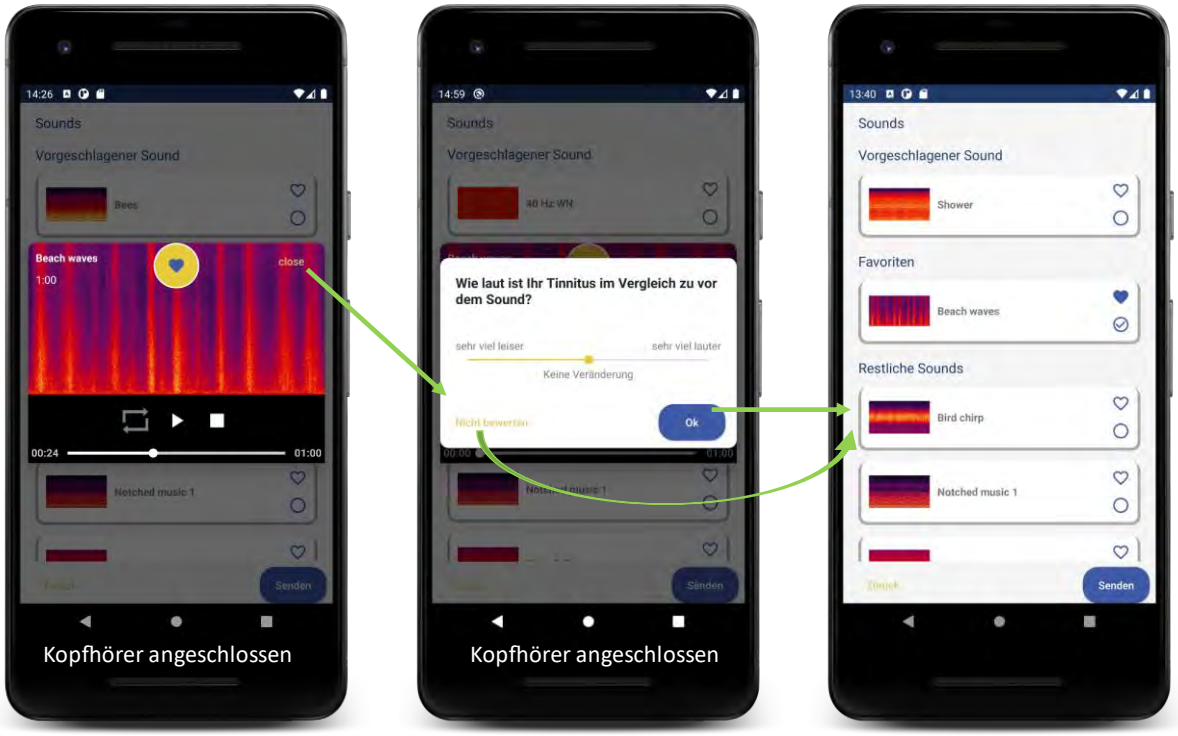


Abbildung 31: Player schließen und Sound bewerten

6.2.6. Sitzungsende

Den standardmäßigen Abschluss einer Sitzung markiert die Interaktion des Nutzers mit dem *Senden*-Button in der Fragebogenansicht, infolgedessen das Übermitteln der Daten an die *API* getriggert wird. Dem Nutzer wird dies anhand einer Erfolgsmeldung visualisiert.

Wird eine Sitzung unterbrochen, wird über die *ActivityLifecyle*[47] Funktionen sichergestellt, dass die zuvor erhobenen und festgehaltenen Daten erhalten bleiben. Die, bis zum Zeitpunkt der Unterbrechung erhobenen Daten, werden versucht an den Server zu senden, sobald die *Activity* zerstört wird. Eine Unterbrechung einer Sitzung kann z.B. ein Anruf auf dem Gerät sein. Bricht der Nutzer die Sitzung ab, indem er wiederholt die *Android Zurück-Taste* benutzt, wird dieser durch ein *PopUp* gewarnt, dass aktuelle Daten der Sitzung verworfen werden.

[47] Siehe Kapitel 5.2 und Abbildung 17

7. Anforderungsabgleich

In diesem Kapitel werden die an die Applikation gestellten Anforderungen, welche in Kapitel 3 definiert wurden, mit der Implementierung und den Funktionen der Applikation abgeglichen. Wie schon bei der Definition der Anforderungen, wird zwischen Rahmenprojekt *UNITI* und dem integrierten Modul differenziert. Gleichermaßen sind die Anforderungen erneut in funktionale und nichtfunktionale Anforderungen unterteilt.

7.1. Funktionale Anforderungen

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

Nummer	Beschreibung	Problembeschreibung
***UNITI*-Applikation Gesamtprojekt**		
1	Registrierung in der Applikation	Anforderungen erfüllt (siehe Kapitel 6.1.2)
2	Anonyme Registrierung	Anforderungen erfüllt (siehe Kapitel 6.1.1 und Kapitel 6.1.2)
3	Login registrierte Nutzer	Anforderungen erfüllt (siehe Kapitel 6.1.1)
4	Login angelegte Nutzer	Anforderungen erfüllt (siehe Kapitel 6.1.1)
5	Automatischer Re-Login	Anforderungen erfüllt. Der Nutzer bleibt dauerhaft eingeloggt, bzw. wird intern ein erneuter Login-Prozess mit den gesicherten Anmeldedaten gestartet. Voraussetzung hierfür ist, dass der Nutzer die Daten der Applikation nicht löscht.
6	Passwort ändern	Anforderungen erfüllt (siehe Kapitel 6.1.6)
7	Passwort vergessen	Anforderungen bestmöglich erfüllt (siehe Kapitel 6.1.3)
8	Profilinformationen	Anforderungen erfüllt (siehe Kapitel 6.1.6)
9	Logout	Anforderungen erfüllt (siehe Kapitel 6.1.6)
10	Impressum	Anforderungen erfüllt (siehe Kapitel 6.1.7)
11	Seite aktualisieren	Anforderungen erfüllt. Es wurde ein Refresh-Button eingeführt. Dieser triggert einen erneuten Ladevorgang der entsprechenden *Activitiy* und den erforderlichen Daten. Zu sehen ist besagter Button in Kapitel 6.1.6 und Kapitel 6.1.5.
12	Ergebnisse	Anforderungen erfüllt (siehe Kapitel 6.1.5)
13	An Studien teilnehmen	Anforderungen erfüllt (siehe Kapitel 6.1.4)
14	Menüstruktur (Navigation Bar)	Anforderungen erfüllt (siehe Kapitel 6.1.4)
15	Navigation	Anforderungen erfüllt. Sobald ein Nutzer in die jeweilige Studie eingeschrieben ist, kann dieser die Module benutzen und beliebig zu diesen navigieren (siehe Kapitel 6.1.4 und Kapitel 6.2).
16	Zurück Button (in Applikation)	Anforderungen erfüllt (siehe Kapitel 6.1.4 und Kapitel 6.1.5)
17	Zurück Taste (Android Gerät)	Anforderungen erfüllt. Betätigt ein Nutzer die *Android-Zurück-Taste* erscheint eine kurze

		Toast-Nachricht, welche diesen darauf hinweist, dass bei abermaliger Interaktion die Applikation geschlossen wird.
18	Netzwerkunabhängigkeit	Anforderungen erfüllt (siehe Kapitel 4.3.2)
19	Benachrichtigungen	Anforderungen erfüllt (siehe Kapitel 4.3.1 und Kapitel 4.4)
Modul für auditorische Stimulation		
20	Sounds visualisieren	Anforderungen erfüllt (siehe Kapitel 6.2.1)
21	Sortierung der Sounds	Anforderungen erfüllt (siehe Kapitel 6.2.1 und Kapitel 6.2.5)
22	Interaktion mit einem Sound	Anforderungen erfüllt (siehe Kapitel 6.2.2 und Kapitel 6.2.3)
23	Playerfunktionalitäten	Anforderungen erfüllt (siehe Kapitel 6.2.3, Kapitel 6.2.4 und Kapitel 6.2.5)
24	Bewertungsdialog	Anforderungen erfüllt (siehe Kapitel 6.2.5, Kapitel 6.2.6 und Listing 19)
25	Synchronisierung der Ergebnisse	Anforderungen erfüllt (siehe Kapitel 6.2.6 und Listing 20)

7.2. Nichtfunktionale Anforderungen

Im folgenden Abschnitt werden die in Kapitel 3.2 definierten nichtfunktionalen Anforderungen an die Applikation abgeglichen.

Nummer	Beschreibung	Problembeschreibung
1	Design	Bestmöglich erfüllt. Das Design der Applikation wurde ansprechend aber nicht überladen gehalten. Es wurde versucht den üblichen Designstandards möglichst nahe zu kommen.
2	Benutzbarkeit	Bestmöglich erfüllt. Es wurde darauf geachtet, dass die Applikation intuitiv und leicht verständlich ist. Falsche oder unerwünschte Eingaben sind bestmöglich ausgeschlossen. Entsprechende Fehlermeldungen werden bei unerwünschtem Verhalten ausgegeben.
3	Transparenz	Bestmöglich erfüllt. Der Nutzer wird direkt zu Beginn auf entsprechende Informationen und den Datenschutz aufmerksam gemacht. Zusätzlich kann der Nutzer jederzeit auf die *AboutActivity* (Über uns) zugreifen, in welcher sämtliche Informationen zum Projekt bereitgestellt werden.
4	Medizinproduktegesetz	Die Applikation wurde so entwickelt, dass sie im weiteren Verlauf die Prüfung zur Zulassung als Medizinprodukt durchläuft und bestenfalls besteht. Allerdings sind hierfür mehrere Testphasen und eine Risikoanalyse der Applikation Voraussetzungen für den Erhalt des Zertifikats. Für besagte Testphasen müssen allerdings alle Inhalte der Applikation bereitstehen.
5	Mehrsprachigkeit	Bestmöglich erfüllt. Durch das Verwenden von lokalen Texten als Verweis auf die Text-Ressourcen der Applikation, wurde es möglich jene Text-Ressourcen in die gewünschten Sprachen zu übersetzen. Texte vom Server werden, sofern vorhanden, ebenfalls in der auf dem Gerät eingestellten Sprache geladen.
6	Verfügbarkeit	Bestmöglich erfüllt. Die Applikation ist für alle Android Geräte ab Betriebssystemversion 5.0 verfügbar. Dies entspricht einer Abdeckung von ca. 95% der weltweit aktiven Geräte.

8. Fazit

Die erste Phase der Entwicklung des *UNITI*-Projektes ist mit der Fertigstellung dieser Arbeit abgeschlossen. Die Funktionalität der Applikation derart umgesetzt, dass die definierten Anforderungen an das Projekt erfüllt wurden. Es wird zwischen den serverseitig vordefinierten und restlichen Nutzern unterschieden. Entsprechend kann sich ein normaler Nutzer manuell, mit E-Mail-Adresse und Passwort, oder anonym in der Applikation registrieren und diese nutzen. Je nach Art des Nutzers wurde es ermöglicht, Studien beizutreten und wieder zu verlassen. Sobald ein Nutzer einer Studie beigetreten ist, wird ihm dynamisch das Bottom-Menü aufgebaut, wodurch jener durch die Applikation navigieren kann. Profil- und Informations-Menüpunkte sind standardmäßig immer verfügbar. Ist ein Nutzer für die Studie der auditorischen Stimulation eingeschrieben, kann er über den entsprechenden Menüpunkt, zum Fragebogen und den Sounds navigieren. Dem Nutzer wird mithilfe des Sound-Fragebogens ermöglicht, jeden in der Applikation verfügbaren Sound wiederzugeben. Die Sounds-Liste ist durch die Favorisierung einzelner Sounds, individuell, benutzerfreundlich und übersichtlich, seitens des Nutzers, gestaltbar. Die willkürliche, regelmäßige Empfehlung eines Sounds verhindert Langeweile und erinnert den Nutzer an noch unbekannte Klänge. Die erhobenen Daten, unter anderem das festgehaltene Wiedergabeverhalten, werden nach jeder Sitzung an den Server gesendet und können in einem zweiten Schritt ausgewertet werden.

Das erste der folgenden Kapitel (8.1) fasst die Erkenntnisse dieser Arbeit zusammen. Das zweite und letzte Kapitel (8.2) behandelt mögliche Ideen, um die Funktionsweise der Applikation noch weiter zu verbessern.

8.1. Zusammenfassung

Im Rahmen dieser Masterarbeit wurde eine mobile Android Applikation, zur Visualisierung und Ausführung mehrerer Studien realisiert. Im gleichen Schritt wurde ein Modul zur auditorischen Stimulation entwickelt und in die Applikation integriert. Die Anforderungen hierfür wurden in mehreren Meetings definiert. Während der Arbeit aufgetretene neue Anforderungen und Ideen wurden bestmöglich berücksichtigt und umgesetzt. Es wurde gezeigt, wie an verschiedenen Studien teilgenommen werden kann, wobei darauf geachtet wurde, ob ein Nutzer die Autorisierung hierzu besitzt. Weiter wurde erreicht, die Funktionalität einer auditiven Stimulation zu realisieren, welche nicht nur eine einfache Beantwortung von Fragebögen, sondern die Wiedergabe von Sounddateien ermöglicht. Hierfür wurde erfolgreich versucht, die vorhandene Fragebogenstruktur derart zu nutzen, um die lokal vorliegenden Sounddateien über einen „Fragebogen“ zu visualisieren und benutzbar zu machen. Die gesammelten Daten werden so verpackt und gespeichert, dass es weiterhin möglich ist, diese nach der vorhandenen Struktur an den Server zu senden. Außerdem wurde im eigentlichen Player auf entsprechend gewünschte Interaktionsmöglichkeiten, wie *Looping*-Funktion und *Stopp*-Button, eingegangen. Zusätzlich wurden Unterbrechungen durch äußere Einflüsse bei einer Wiedergabe des Sounds berücksichtigt, um den geforderten Ansprüchen gerecht zu werden und nutzbare Datensätze zu erhalten. Die im Laufe der Entwicklung aufkommende Forderung nach einer Library für die Serververbindung wurde zwar implementiert, jedoch aus zeitlichen Gründen, zur Fertigstellung dieser Arbeit, noch nicht in das Projekt integriert. Dies wäre ein erster möglicher Verbesserungsvorschlag.

8.2. Ausblick

Dieser Abschnitt beschreibt mögliche Ideen, die während der Entwicklung der Applikation für das Projekt aufgekommen sind. Ziel sollte es sein, die mobile Applikation weiterhin zu verbessern. Infrage kommen dabei die Integration der *API*-Kommunikation als Library, Verbesserungen am User-Inferface oder auch neue nützliche Funktionen, welche andere Projekte bereits besitzen und im Rahmen des *UNITI*-Projektes eventuell ebenfalls sinnvoll wären.

8.2.1. API-Kommunikation als Library

Wie erwähnt[48] wurde es im Laufe der Entwicklung der Applikation deutlich, dass die Kommunikation und Verbindung zur *API* zentrale Bestandteile des Projektes sind. Da diese Verbindungen generell immer nach dem gleichen Schema ablaufen, lag es auf der Hand diese Funktionalität als Library auszulagern. Aufgrund von anderen Prioritäten wurde dies jedoch aufgeschoben. Zum Zeitpunkt dieser Arbeit wurde bereits ein Großteil dieser Kommunikation mit dem Server als Bibliothek implementiert. Da es aber noch keine ausreichende Testphase der implementierten Library gab, wäre der zeitliche Aufwand bis zur Abgabe dieser Arbeit schlichtweg zu enorm gewesen, um die Struktur, den Aufbau und die Modelle entsprechend anzupassen. Deshalb wurde die Entscheidung getroffen, die vervollständigte und getestete Library zu einem späteren Zeitpunkt in das bestehende Projekt zu integrieren.

8.2.2. Verbesserung am User-Interface und Hilfe

Generell standen zuallererst 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 Nutzers mit der Applikation noch besser zu gestalten. Nichtdestotrotz wurde bestens darauf geachtet, die Designs und Darstellungen so ansprechend und intuitiv wie möglich zu gestalten. Allerdings gibt es momentan noch wenige bis keine Hilfs- oder Infotexte bezüglich der Bedienung der Applikation, was einem Nutzer bei eventuellen Verständnisproblemen sicherlich helfen würde. Abgesehen davon muss man sagen, dass Design und Interaktionsmöglichkeiten sehr subjektiv empfunden werden. Daher sollte man erste, größere Feedbacks der Nutzer abwarten und auf eventuelle Verbesserungs- und Veränderungsvorschläge eingehen.

8.2.3. Sounds aus der eigenen Mediathek

Eine Funktionalität, die vergleichbare Applikationen oftmals schon implementiert haben, ist die Möglichkeit, dass der Nutzer dem Modul der auditorischen Stimulation, Melodien, Lieder oder Sounds aus der eigenen Mediathek des Geräts hinzufügen kann. Dies hätte den Vorteil, dass die Applikation noch individueller auf jeden einzelnen Nutzer zugeschnitten wäre. Dazu wissen viele Tinnitus-Patienten welche Melodien oder Lieder Sie beruhigen, Ihnen guttun oder sich sogar positiv auf ihre wahrgenommenen Töne auswirken. Eventuell könnte in einer späteren Datenauswertung, anhand der Bewertung und des Wiedergabeverhaltens der Nutzer, sogar erkannt werden, ob es bestimmte Künstler, Genres oder Ähnliches gibt, die sich nicht nur bei Einzelnen positiv auswirken, sondern bei ganzen Patientengruppen. Voraussetzung einer solchen Funktionalität ist immer die Frage, ob es

[48] Siehe Kapitel 5.3

wissenschaftlich Sinn ergibt, dem Nutzer, jeden ihm möglichen Sound zur Verfügung zu stellen oder, ob er auf bestimmte, ausgewählte Sounds beschränkt sein soll.

8.2.4. Listenstruktur

Eine ähnliche betrachtete App[49] ermöglicht es Nutzern zum einen, bestimmte Sounds zu ausgewählten Listen zuzuordnen und zum anderen, eigene Listen für bestimmte Momente oder Situationen zu erstellen. Dies hat den Vorteil, dass der Nutzer, Sounds, die Ihn z.B. beruhigen, genauso zusammenfassen kann, wie welche, die jener gern im Hintergrund anhört. Ebenso wäre es möglich Sounds für verschiedene, spezielle, emotionale Situationen zu bündeln. Als Beispiel könnte ein Nutzer eine Liste mit Sounds anlegen, welche dieser gerne beim oder vor dem Einschlafen anhört, sowie eine zweite Liste mit Sounds zur Wiedergabe im Auto. Auch dies würde die Applikation weiter individuell gestaltbar und nutzbar machen. Eine ähnliche Funktion ist durch die Favoriten Funktion generell bereits gegeben, allerdings kann der Nutzer bisher nur markieren, ob ihm der gewählte Sound zusagt oder nicht.

[49] Siehe Phonak Tinnitus Balance Applikation in Kapitel 2.5

Literaturverzeichnis

[1] Unification of treatments and Interventions for Tinnitus patients: Cordis, UNITI. https://cordis.europa.eu/project/id/848261/de. Abgerufen am 09.10.2020.

[2] Haug, F (2020): Konzeption und Realisierung eines Patienten-Edukationsmoduls für eine multizentrische und multinationale mHealth-App für eine paneuropäische Tinnitus-Studie.

[3] Herrmann, J (März.2014): Konzeption und technische Realisierung eines mobilen Frameworks zur Unterstützung tinnitusgeschädigter Patienten.

[4] Der deutsche Wortschatz: dwds. https://www.dwds.de/wb/Tinnitus. Abgerufen am 22.09.2020.

[5] Deutsche Tinnitus-Liga: Deutsche Tinnitus-Liga. https://www.tinnitus-liga.de/pages/tinnitus-sonstige-hoerbeeintraechtigungen/tinnitus-hoersturz-hyperakusis-morbus-meniere.php. Abgerufen am 09.10.2020.

[6] Cederroth, CR, Gallus, S, Hall, DA, Kleinjung, T, Langguth, B, Maruotti, A, Meyer, M, Norena, A, Probst, T, Pryss, R, Searchfield, G, Shekhawat, G, Spiliopoulou, M, Vanneste, S, Schlee, W (2019): Editorial: Towards an Understanding of Tinnitus Heterogeneity. Frontiers in Aging Neuroscience, 11:53.

[7] proaktustik: www.proakustik.de. https://www.proakustik.de/2019/10/15/tinnitus-das-klingeln-im-ohr/#:~:text=Tinnitus%20ist%20inzwischen%20zur%20Volkskrankheit,Knistern%2C%20Zischen%20oder%20auch%20Klopfen. Abgerufen am 15.09.2020.

[8] Tinnitus in Zahlen: TIEX Tinnito Terapia. http://www.tinnitus.cc/it/orecchie-e-tinnito/tinnituszahlen-1/index.php. Abgerufen am 06.10.2020.

[9] HNO am Marienring: dr-thorn.de/hno-landau-intratympanale-kortikoidtherapie. https://www.dr-thorn.de/hno-landau-intratympanale-kortikoidtherapie.html#:~:text=Bei%20der%20Intratympanalen%20Kortikoidtherapie%20wird,auch%20als%20gesonderte%20Therapieform%20erfolgen. Abgerufen am 20.10.2020.

[10] Schlee, W, Hall, DA, Canlon, B, Cima, RFF, Kleine, E de, Hauck, F, Huber, A, Gallus, S, Kleinjung, T, Kypraios, T, Langguth, B, Lopez-Escamez, JA, Lugo, A, Meyer, M, Mielczarek, M, Norena, A, Pfiffner, F, Pryss, RC, Reichert, M, Requena, T, Schecklmann, M, van Dijk, P, van de Heyning, P, Weisz, N, Cederroth, CR (2018): Innovations in Doctoral Training and Research on Tinnitus: The European School on Interdisciplinary Tinnitus Research (ESIT) Perspective. Frontiers in Aging Neuroscience, 9:447.

[11] DBIS Universität Ulm: TrackYourTinnitus. https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/trackyourtinnitus/. Abgerufen am 22.10.2020.

[12] Schlee, W, Pryss, P, Probst, T, Schobel, J, Bachmeier, A, Reichert, M, Langguth, B (2016): {Measuring the Moment-to-Moment Variability of Tinnitus: The TrackYourTinnitus Smart Phone App. Frontiers in Aging Neuroscience, (8):294.

[13] Schickler, M, Reichert, M, Pryss, R, Schobel, J, Schlee, W, Langguth, B (2015): Entwicklung mobiler Apps: Konzepte, Anwendungsbausteine und Werkzeuge im Business und E-Health.

[14] Pryss, R, Schlee, W, Langguth, B, Reichert, M (2017): Mobile Crowdsensing Services for Tinnitus Assessment and Patient Feedback. In: Press, ICS (Hrsg), *6th IEEE International Conference on AI \& Mobile Services (IEEE AIMS 2017)*.

[15] Pryss, R, Reichert, M, Langguth, B, Schlee, W (2015): Mobile Crowd Sensing Services for Tinnitus Assessment, Therapy and Research. In: Press, ICS (Hrsg), *IEEE 4th International Conference on Mobile Services (MS 2015)*.

[16] Pryss, R, Reichert, M, Herrmann, J, Langguth, B, Schlee, W (2015): Mobile Crowd Sensing in Clinical and Psychological Trials ? A Case Study. In: Press, ICS (Hrsg), *28th IEEE Int'l Symposium on Computer-Based Medical Systems*.

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

[18] Probst, T, Pryss, R, Langguth, B, Spiliopoulou, M, Schobel, J, Reichert, M, Schlee, W (2017): Does tinnitus depend on time-of-day? An ecological momentary assessment study with the TrackYourTinnitus application. Frontiers in Aging Neuroscience, (9):253.

[19] Probst, T, Pryss, R, Langguth, B, Spiliopoulou, M, Landgrebe, M, Vesala, M, Harrison, S, Schobel, J, Reichert, M, Stach, M, Schlee, W (2017): Outpatient Tinnitus Clinic, Self-Help Web Platform, or Mobile Application to Recruit Tinnitus Study Samples? Frontiers in Aging Neuroscience, (9):113--113.

[20] Probst, T, Pryss, R, Langguth, B, Schlee, W (2016): Emotional states as mediators between tinnitus loudness and tinnitus distress in daily life: Results from the TrackYourTinnitus application. Scientific Reports.

[21] DBIS Universität Ulm: AssesYourStress. http://dbis.eprints.uni-ulm.de/1395/. Abgerufen am 20.10.2020.

[22] Bundesministerium für Gesundheit: Präventionsgesetz. https://www.bundesgesundheitsministerium.de/service/begriffe-von-a-z/p/praeventionsgesetz.html. Abgerufen am 22.10.2020.

[23] Bundesministerium für Gesundheit: Bundesgesetzblatt online Zugang. https://www.bgbl.de/xaver/bgbl/start.xav?startbk=Bundesanzeiger_BGBl&jumpTo=bgbl115s1368.pdf#__bgbl__%2F%2F*%5B%40attr_id%3D%27bgbl115s1368.pdf%27%5D__1603528963852. Abgerufen am 24.10.2020.

[24] Groll, T (2014): Zeit Online. http://www.zeit.de/karriere/2014-08/anti-stress-gesetz-chancen. Abgerufen am 28.05.2017.

[25] Duden: duden online Stress. http://www.duden.de/rechtschreibung/Stress. Abgerufen am 28.09.2020.

[26] Berres, I (2013): Spiegel Online. http://www.spiegel.de/gesundheit/diagnose/so-gestresst-sind-die-deutschen-umfrage-der-techniker-krankenkasse-a-930696.html. Abgerufen am 28.05.2017.

[27] Sonormed GmbH Tinnitracks: Tinnitracks. https://www.tinnitracks.com/de. Abgerufen am 07.10.2020.

[28] Trias, Wissen was gut tut: Counselling. https://www.thieme.de/de/gesundheit/counselling-tinnitus-retraining-45284.htm#:~:text=%E2%80%9ECounselling%E2%80%9C%20im%20Rahmen%20des%20Tinnitus,%C3%A4rztliche%20Beratung%20des%20Tinnituspatienten%20verstanden. Abgerufen am 07.10.2020.

[29] Springer Medizin: tailor-made notched music training. https://www.springermedizin.de/tinnitus/haelt-das-tailor-made-notched-music-training-was-es-verspricht/16161948#:~:text=Von%20sich%20reden%20machte%20in,sie%20keine%20Signalanteile%20mehr%20enth%C3%A4lt. Abgerufen am 07.10.2020.

[30] Sonova GmbH: Phonak. life is on. https://www.phonak.com/de/de.html. Abgerufen am 24.10.2020.

[31] audibene GmbH: audibene. So klingt Glück! https://www.audibene.de/. Abgerufen am 08.10.2020.

[32] Tinntius Help. http://www.tinnitus-help.eu/. Abgerufen am 08.10.2020.

[33] Gerhard Hesse: Deutsche Tinnitus-Liga. https://www.tinnitus-liga.de/pages/presse/pressemitteilungen/apps-und-tinnitus-behandlung.php. Abgerufen am 08.10.2020.

[34] Google Inc: Android Developer. https://developer.android.com/. Abgerufen am 22.10.2020.

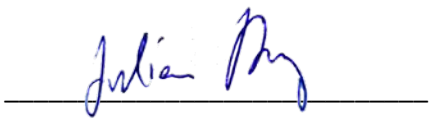
[35] (2009): Model-view-controller pattern. Learn Objective-C for Java Developers:353–402.

[36] Android Pedia: Gson. https://androidpedia.net/en/tutorial/4158/gson. Abgerufen am 21.10.2020.

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 29.10.2020



Julian Haug