Hochschule Kaiserslautern
University of Applied Sciences

Mobile Anwendungen mit Android
Sommersemester 2018
Abgabedatum: 06.07.2018

BOSCall - Dokumentation

Dorian Weidler, B.Sc. (868407) - Christoph Suffel, B.Sc. (866555)

Zusammenfassung
BOSCall wurde im Rahmen der Veranstaltung Mobile Anwendungen mit Android an der Hochschule Kaiserslautern im Sommersemester 2018 entwickelt. Der betreuende Dozent der Arbeit war Prof. Dr. Manh Tien Tran.
Ziel der Veranstaltung war die Entwicklung einer Android App, die Wahl der App stand den Studierenden frei. Die Entscheidung fiel recht schnell, dass eine App zur mobilen Alarmierung von Einsatzkräften entwickelt werden sollte. Zum Zeitpunkt der Entwicklung gab es keine kostengünstige beziehungsweise kostenlose Variante für mehrheitlich ehrenamtlich besetzte Behörden mit Sicherheitsaufgaben (BOS). BOSCall soll als Open-Source Ansatz jeder Feuerwehr mit dem nötigen Know-How eine Basis geben Ihre Einsatzkräfte zusätzlich auch über ihr privates mobiles Endgerät zu alarmieren. Durch die hohe Verbreitung von Smartphones und die stetig flächendeckendere Mobilfunkabdeckung kann durch diesen Zusatzalarmiert mit einer hohen Wahrscheinlichkeit der Alarm jedes einzelnen Mitglieds sichergestellt werden.

Keywords
Android — Feuerwehr — Alarmierung

Corresponding authors: dowe0012@stud.hs-kl.de, chsu0001@stud.hs-kl.de

Inhaltsverzeichnis

1. Einleitung

Für Fördervereine der freiwilligen Feuerwehr ist es eine massive Kostenbelastung, wenn bereits wenige hundert Euro jedes Jahr zu bezahlen sind. Die Träger der Feuerwehren berufen sich bezüglich der Alarmierung per Smartphone darauf, dass bereits durch analoge oder digitale Meldeempfänger ausreichend vorgesorgt ist und die Einsatzkräfte adäquat alarmiert werden.

In der Feuerwehr Kusel wurde beispielsweise eine firEmergency[1] Installation betrieben, die durch den Förderverein finanziert wurde. Das heißt Spenden, Mitgliederbeiträge und Veranstaltungserlöse werden genutzt,

um eine grundlegende Anforderung sicherzustellen. Ein Kostenvergleich zwischen einer aktuellen firEmergency 2 Installation und der BOSCall Variante folgt in Kapitel 2.

In Kapitel 3 wird auf die Funktionsweise der Alarmierung mittels BOSCall eingegangen. Die beiden Alarmetappen, die dabei durchlaufen werden, werden in dem Kapitel näher erläutert. Anschließend dient Kapitel 5 dazu, verschiedene Problemstellungen und deren Lösungen zu erörtern, die bei der Implementierung der App aufgetreten sind.

Kapitel 6 beschreibt im Anschluss die Schritte, die erforderlich sind, um die App in Betrieb zu nehmen. Abschließend wird in Kapitel 7 ein Fazit über gezogen. Dabei wird insbesondere auf die erreichten und nicht erreichten Ziele der App eingegangen.

2. Kostenvergleich

Wie bereits einleitend erwähnt und auch technisch bedingt entstehen Kosten, um diese Art von digitalem Alarm zu realisieren. Zunächst soll hierbei auf eine fertige Lösung von einem etablierten Anbieter eingegangen werden. Diese ist bereits in der Feuerwehr Kusel, die als Testeinheit dient, in einer älteren, nicht mehr verfügbaren Version im Einsatz. Die App nennt sich APager (Pro), die Backend Software, die unabdingbar für die App ist, nennt sich firEmergency. Beides sind fertige Lösungen der Firma Alamos GmbH.

Betrachtet wird im Folgenden eine Lösung auf Basis einer serverseitigen Lizensierung. Das heißt die Kosten fallen aufseiten des Betreibers an und sind entsprechend in Summe günstiger als der Kauf einzelner Lizenzen pro App. Die Preise stammen von [2]. Unterstützt werden soll eine Nutzerbasis von 200 Personen.

Tabelle 1. Kostenaufstellung APager & firEmergency

Beschreibung	**Preis**
firEmergency Paket 1 (30 Personen)	179,99 €/Jahr
+170 Personen	593,30 €/Jahr
Windows Office PC	ca. 400€
Summe	**1173,29 €**

Bei der Aufstellung der Kosten in Tabelle 1 wurde berücksichtigt, dass Einzelpersonen günstiger sind als die größeren Pakete, wenn man wirklich nur die Personen und nicht weitere Features benötigt. Da BOSCall die anderen Features nicht bietet, wird der Fairness halber mit dem preislich günstigsten Paket und den zusätzlichen Personen gerechnet. Ein Computer mit Windows Betriebssystem ist erforderlich um firEmergency zu betreiben.[3] Aufsummiert fallen im ersten Jahr bereits 1173,29 € an. In den Folgejahren, unter der Voraussetzung die Preise ändern sich nicht und der Rechner wird einfach stetig weiterbetrieben, 773,29 €.

Das ist für einen kleinen, gemeinnützigen Verein kaum zu stemmen, insbesondere weil es sich nicht um einmalige, sondern laufende Kosten handelt. Die Ersatzlösung, die im Rahmen dieser Veranstaltung entwickelt wurde, soll diese Kosten also drastisch unterschreiten, damit sich der zusätzliche Aufwand rechtfertigt. In Tabelle 2 befindet sich eine Beispielaufstellung der Kosten, mit denen zu rechnen ist, um BOSCall zu betreiben. Da die Anwendung noch nicht in der Praxis erprobt ist, handelt es sich dabei zunächst um eine rein theoretische Aufstellung sämtlicher Kosten.

Tabelle 2. Kostenaufstellung BOSCall

Beschreibung	**Preis**
Google Entwickler Zugang	$ 25
Virtueller Server	5 €/Monat
Raspberry Pi m. Zubehör	50 €
USB Soundkarte	6,25 €
Summe	**137,75 €**

Man erkennt direkt, dass die Kosten erheblich geringer sind. Zudem verteilen sie sich fast nur auf einmalige Kosten. Der Google Entwickler Zugang ist erforderlich um Apps im Play Store zu veröffentlichen und sie so der breiten Masse an Feuerwehrmitgliedern der Feuerwehr Kusel einfach zugänglich zu machen. Der virtuelle Server ist für das Bereitstellen der Backend API erforderlich. Was der Dienst können muss wird in Kapitel 3 erläutert. Der Raspberry Pi mit der USB Soundkarte ist für die lokale Installation eines Alarmauswerters erforderlich. Auch dabei wieder der Verweis auf Kapitel 3. In Summe liegt man dabei im ersten Jahr bei ca. 137,75 €, abhängig vom Wechselkurs des Dollar. In den Folgejahren belaufen sich die Kosten auf 60 €. Dabei besteht zusätzlich der große Vorteil, dass man auch problemlos mehr Mitglieder anlegen kann ohne, dass sich die jährlichen Kosten erhöhen. Bereits der schwächste Linux Server, der heute üblicherweise angeboten wird, bietet erheblich mehr Leistung als erforderlich.

3. Funktionsweise

Prinzipiell dreht sich die App um zwei essenzielle Funktionen. Die Registration und der Alarmempfang. Zusätzlich findet auch noch ein wichtiger Abgleich der registrierten Einheiten mit dem Backend System statt. Bei dem Abgleich werden Einheiten lokal deregistriert, falls der Benutzer im Backend keine Zuordnung mehr zu ihr hatte. Diese drei Funktionalitäten werden nachfolgend genauer beschrieben. Es gibt natürlich noch weitere Funktionen, die aber entweder im Kapitel 5 bereits Erwähnung finden oder schlicht nicht erwähnenswert sind.

3.1 Registration

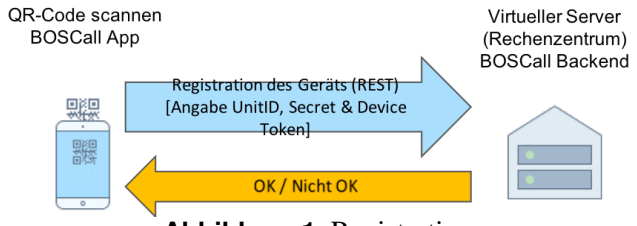

Abbildung 1. Registration

Anforderungen an die Registration war eine möglichst geringe Komplexität und zugleich eine hohe Sicherheit zu realisieren. Der Fokus lag aber zwecks des begrenzten Entwicklungszeitraums auf der geringen Komplexität.

Um die Komplexität zu reduzieren wird jedem Leiter einer Einheit ein QR-Code ausgestellt, der dazu dient, dass sich Mitglieder registrieren können. Es ist für den Anwender also nur erforderlich, dass er seinen Namen eingibt und einen QR-Code mit seinem Smartphone scannt. In diesem QR-Code befinden sich alle Details, die zur Registration erforderlich sind. Zum Beispiel die ID der Einheit (UnitID) und der private Schlüssel (Secret). Zusätzlich sendet das Gerät bei der Registrationsanfrage den eigenen Gerätetoken (Devicetoken) mit. Damit können Push Nachrichten gezielt an das Endgerät gesendet werden.

3.2 Alarmempfang

Der Ablauf der Alarmierung, graphisch in Abbildung **??** zu sehen, teilt sich in drei Schritte. Zunächst wird der Alarm über eine Auswertung des analogen Funks erkannt und es wird die E-Mail der Leitstelle mit den Einsatzdetails ausgewertet. Dieser Prozess wird nicht durch die Anwendungen abgedeckt, die im Rahmen der Veranstaltung entwickelt wurden. Es gibt bereits eine funktionierende Bestandsanwendung, die sich relativ problemlos auf die Unterstützung von BOSCall ändern lässt.

Im nächsten Schritt wird durch die soeben genannte Anwendung per HTTP Aufruf (REST) ein Alarmprozess angestoßen. Der Dienst, der dabei aufgerufen wurde, liegt dabei auf einem öffentlichen virtuellen Server. Der Service wertet nun den eingegangenen Alarm aus, indem er die Einheit aus der Anfrage bezieht. Mittels diesem Datum bekommt er aus einer Datenbank alle registrierten Endgeräte dieser Einheit. Deren Token nutzt er, um über die API von Firebase eine Push Nachricht an das Gerät zu senden.

Im letzten Schritt wird die erhaltene Push Nachricht auf dem Endgerät ausgewertet. Diese sogenannte Data Message enthält alle Daten, die für die Darstellung einer Alarmierung benötigt werden, also Alarmtitel und Alarmtext. Es gilt hier zwischen Data Message und einer sogenannten Notification Message zu unterscheiden. Diese Unterscheidung bereitete zunächst Probleme, weshalb diese beiden Typen weiter im Kapitel 5 erläutert werden.

3.3 Einheitenabgleich

Wie eingangs beschrieben ist das Ziel des Einheitenabgleichs, dass lokal keine Einheiten mehr registriert erscheinen, deren Registration auf dem Backend nicht vorliegt. Schließlich ist die Single Source of Truth (SSOT) das Backend. Die Daten, die zum Senden jedes Alarms notwendig sind stammen nämlich aus dem Backend. Aktuell ist es nur vorgesehen, dass sich Benutzer für Einheiten registrieren. Es ist zwar möglich, dass ein Administrator einen Benutzer in der Datenbank zu einer weiteren Einheit hinzufügt, diese erscheint dann aber nicht in der Übersicht der Einheiten in der App des Benutzers. Er kann sich dann auch nicht von dieser Einheit abmelden. Die Alarmierungen werden aber trotzdem an das Endgerät ausgeliefert.

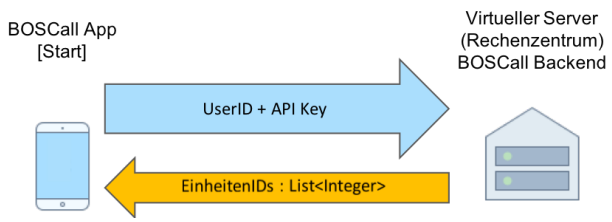

Abbildung 3. Einheitenabgleich

In Abbildung 3 ist der Ablauf dieses Abgleichs zu sehen. Er findet jedes Mal statt, wenn die App geöffnet wird. Aus diesem Grund wurde auch ein Ladebildschirm

implementiert. Es wird die BenutzerID und der bei der Registration zugeteilte API Schlüssel mit dem Backend ausgetauscht. Nach erfolgreicher Prüfung wird dann eine Liste mit allen EinheitenIDs an das Endgerät geschickt. Im Anschluss werden alle Einheiten entfernt, deren IDs nicht in der Liste sind.

4. Architektur

Die App startet immer mit einem Splashscreen, dieser findet sich in der SplashActivity. Dort können Dinge geladen werden, die im Anschluss in der App dringend benötigt werden. Beispielsweise findet dort der Abgleich der Einheiten statt. Das heißt es werden alle lokal gespeicherten Einheiten gelöscht, denen das Gerät nicht mehr angehört.

Im Anschluss startet die MainActivity. Da die App eine BottomNavigationView nutzt, gibt es nur wenige Activities, die wichtigste ist die MainActivity. In ihr wird auch festgelegt, welches Fragment geladen wird, wenn der Benutzer etwas in der Navigation wählt.

Außer den bereits erwähnten Activities gibt es nur noch die UnitReaderActivity. Diese wird beim Registrieren einer Einheit geöffnet und enthält einen QR-Code Reader, der nach erfolgreichem Scannen wieder geschlossen wird.

Im services-Package befinden sich die Datenbank-Abstraktionen, Webservice-Abstraktionen und die benötigten Services für Firebase Cloud Messaging (BosCallMessagingService - Wird bei jeder Push Nachricht benutzt, BosCallIdService - Wird bei Änderungen des Gerätetokens für FCM benötigt).

Das dbTasks-Paket enthält asynchron auszuführende Aufgaben um mit der SQLite Datenbank zu interagieren. Die dort enthaltenen Operationen dauern an dem Ort, an dem sie ausgeführt werden, zu lange und werden automatisch durch das Android Betriebssystem unterbunden, sofern sie synchron ausgeführt werden.

Innerhalb des dao-Package befinden sich Interfaces, die in Kombination mit Room genutzt werden können um mit der SQLite Datenbank zu interagieren. Darin befinden sich Methoden, um mit den abstrahierten Daten zu arbeiten, beispielsweise neue einzufügen, einzelne oder alle zu löschen.

Im util-Paket enthält Klassen, um die lokal gespeicherten Registrationen zu speichern und zu laden. Dabei handelt es sich noch um die Persistierungsvariante auf dem Dateisystem im JSON Format, anstatt der Nutzung der SQLite Datenbank.

Das dto-Package besitzt Klassen allerhand reine Datenklassen die an den verschiedensten Stellen der App benötigt werden. Die auf Request endenden Klassen werden beispielsweise zur Befüllung der Anfragen an die Webservices genutzt.

Das letzte Paket constants, enthält ein Enum mit den Service Konfigurationsdaten. Hier kann angepasst werden, welche URL aufgerufen wird, um den Webservice zu erreichen.

5. Problemstellungen

In diesem Kapitel soll der Fokus auf die aufgetretenen Probleme und Schwierigkeiten gelegt werden. Die Problemstellungen reichen hierbei von generellen Schwierigkeiten bis zu ganz speziellen Problemen, die gegebenenfalls nur in dieser Konstellation auftreten würden.

5.1 Sprachschwierigkeiten

Das generellste Problem ist offensichtlich. Es wurde zuvor nicht mit Kotlin gearbeitet, entsprechend gab es hier bereits die einen oder anderen Startschwierigkeiten. Problematischer als die Eigenheiten von Kotlin gestaltet sich aber, dass nach wie vor die meisten Apps in Java geschrieben werden. Entsprechend sieht es auch bei den Dokumentationen zu Bibliotheken aus. Die meisten Beispiele sind schlicht in Java geschrieben.

Android Studio als Entwicklungsumgebung bietet zwar die Möglichkeit Java in Kotlin Code umzuwandeln, doch klappt das nicht immer perfekt. Speziell bei den Features bezüglich Null-Safety treten im umgewandelten Code Probleme auf. Entweder weil es schlicht nicht berücksichtigt wurde, dann ist der Code nicht ohne Weiteres kompilierbar, oder weil Variablen, die möglicherweise den Wert ”null”haben könnten, werden hinsichtlich dieser Eigenschaft ignoriert. D. h. es wird einfach auf die Variable zugegriffen und im Worst-Case wird eine NullPointerException ausgeworfen.

5.2 Auswahl der Persistierungsart

Da in BOSCall auch Daten gespeichert werden müssen, z. B. die Einheiten, Alarmierungen und Einstellungen, ist es notwendig die verschiedenen Arten der Persistierung zu evaluieren.

Der erste Anwendungsfall für die Persistierung war die Speicherung der registrierten Einheiten. Zunächst wurde darüber nachgedacht, die Daten in einer SQLite Datenbank abzulegen. Zu diesem Zeitpunkt erschien es aber zunächst zu komplex und es war wichtiger die Alarmierungsfunktionalität zu realisieren. Aus diesem

Grund wurde die Entscheidung getroffen, die Einheiten im JSON Format als .json-Datei auf dem internen Dateisystem abzulegen. Zur effektiven Verarbeitung von JSON Strings wurde im Rahmen der Anwendung die Google GSON Bibliothek[4] verwendet.

Mit zunehmender Anzahl an persistenten Daten fiel die Entscheidung darauf, für alle weiteren Daten auf SQLite zu setzen. Da die SQLite Funktionalität von Android aber etwas komplizierter für ausschweifende Arbeiten ist und auch keine native Unterstützung für die Abbildung der Daten in der Datenbank auf Objekte bietet, wurde Room verwendet. Die Verwendung von Room ist auch die Empfehlung von Google.[5]

Room bietet den großen Vorteil des Objekt Mappings. D. h. es wird von SQL abstrahiert. Man spart sich damit also Klassen zu schreiben, die sich nur mit der Datenbank Kommunikation beschäftigen. Der Verbindungsauf- und -abbau und die Auswertung von Daten zu komplexen Objekten wird durch Room erledigt.

5.3 Kommunikation mit dem Backend

Die Kommunikation mit dem Backend erfolgt über HTTP im REST Stil. Auch hier bringt Android bzw. Java Möglichkeiten mit, um mit dem Service zu kommunizieren. Ebenfalls erfolgt mit diesen kein automatisches Mapping der Daten (vorliegend in JSON) zu komplexen Objekten.

Es gäbe es die Möglichkeit das Mapping manuell selbst zu schreiben, was aber aufgrund des zusätzlichen Aufwands und des hohen Fehlerpotentials ausgeschlossen wurde. Stattdessen wurde auf eine getestete Bibliothek zurückgegriffen, die diese und weitere Funktionalitäten mitbringt. Aufgrund der guten Dokumentation und der bereits nicht zu verachtenden Gemeinschaft um das Projekt, fiel die Entscheidung auf Retrofit.[6] Die kurze und verständliche Dokumentation, sowie die umfangreichen Tutorials im Internet, machten diese Entscheidung recht leicht.

5.4 FCM - Notification Message oder Data Message

Diese Problematik tritt in Verbindung mit Firebase Cloud Messaging (FCM) auf, das in diesem Projekt für die Push Nachrichten erforderlich ist. Die Wahl fiel auf insbesondere deshalb auf Firebase, weil das Projekt von Google betrieben wird und die Implementierung in BOSCall entsprechend leicht zu realisieren war. Die Integration wird zusätzlich durch Android Studio unterstützt. Nicht zuletzt war aber der größte Vorteil, dass die Nutzung von Firebase Cloud Messaging, im Gegensatz zu anderen Push Dienstleistern kostenlos ist.

Das Hauptproblem bei der Nutzung von FCM entstand, wenn man prüft, wie sich die App im Hintergrund verhält, wenn z.B. das Gerät gesperrt ist. Dabei fiel auf, dass die Nachrichten nur als Notification ausgeliefert wurden, aber nicht die programmierten Empfangsroutinen durchlaufen wurden. Wenn man die Notification anschließend angeklickt hat um die App zu öffnen, kam man auch nur an die Daten dieser einzelnen Notification, die anderen wurden sofort gelöscht. Damit gehen effektiv Alarmierungsdaten verloren, die für das alarmierte Personal essenziell sein könnten. Ein Beispiel: Es wird zunächst normal zu einem Gebäudebrand alarmiert. Bei der Alarmierung werden die Adressdaten der Einsatzstelle übertragen. Im Anschluss wird ein zweites Mal alarmiert um darin mitzuteilen, dass die Anfahrt aufgrund einer Straßensperrung über einen anderen Straßenzug erfolgt. Klickt der Alarmierte nun den neuesten, also den zweiten, Alarm an, fehlt ihm die Adresse um die Einsatzstelle überhaupt anfahren zu können.

Eine Lösung des Problems wäre, das alle ausgelösten Alarmierungen auch auf dem virtuellen Server gespeichert werden. Sobald jemand die App öffnet werden alle Alarmierungen abgerufen, die noch nicht auf dem Endgerät gespeichert wurden. Das hat aber einen gravierenden Nachteil. Denn dazu müssten alle Alarmierungen, inklusive sensibler Daten, auf dem virtuellen Server gespeichert werden. Aus Datenschutzgründen sollte das vermieden werden.

Nach einer intensiven Recherche war klar, dass bei FCM zwischen zwei Arten einer Push Nachricht unterschieden wird. Zunächst die Notification Message. Diese hat die oben genannte Problematik, wenn die App sich im Hintergrund befindet. Zusätzlich gibt es aber auch die Möglichkeit Data Messages zu versenden. Um eine solche Nachricht zu verschicken, kann man nicht die Admin Console von Firebase verwenden, was bereits für einige Verwirrung sorgte, da zum Testen bis dahin immer die Admin Console verwendet wurde. Es ist erforderlich eine der angebotenen Schnittstellen zu verwenden. Für das BOSCall Projekt bedeutete das eine weitere Verzögerung, da zunächst ein Alarmprogramm unter Nutzung der FCM Admin SDK entwickelt werden musste. Dazu gibt es diverse Tutorials, die aber für unterschiedliche Versionen von FCM geschrieben wurden und entsprechend im Laufe der Zeit nicht mehr in einem funktionsfähigen Programm endeten. Nachdem diese Schwierigkeiten überwunden waren, stellte sich schnell heraus, dass an der App keine weiteren Änderungen er-

forderlich sind. Denn bei Empfang einer Data Message durchläuft unabhängig vom aktuellen Zustand der App (Vorder- oder Hintergrund) jede Nachricht die bereits zuvor geschriebenen Routine.

6. Installationsvorbereitungen

In diesem Kapitel werden die Schritte erklärt, die durchgeführt werden müssen, bevor man die .apk-Datei erzeugen lassen kann.

6.1 Erforderliche Schnittstellen

Zunächst muss ein Backend System verfügbar sein. Dieses muss einige Schnittstellen bieten, die nachfolgend erläutert werden. Zusätzlich ist es erforderlich, dass alle Anfragen in JSON empfangen werden können und alle Antworten in JSON vorliegen.

6.1.1 Registrationsschnittstelle

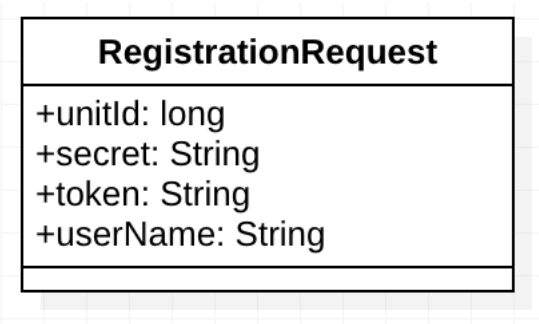

Abbildung 4. RegistrationRequest

Die Registrationsschittstelle muss unter dem Endpunkt */registration* per POST-Methode erreichbar sein. Im RequestBody wird ein sog. *RegistrationRequest* mitgeschickt, der in Abbildung 4 zu sehen ist. Die Daten um die Anfrage zu befüllen stammen aus dem QR-Code (UnitID, Secret) beziehungsweise aus der App des Benutzers (Token, Name).

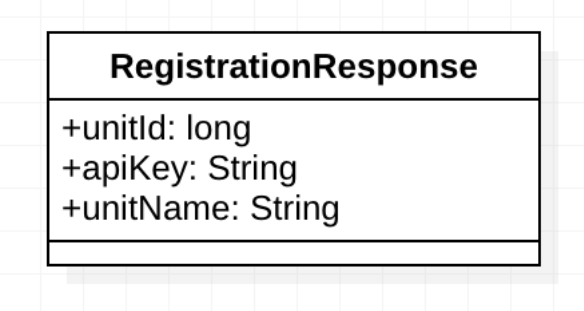

Abbildung 5. RegistrationResponse

Als Ergebnis auf die Anfrage erwartet die App eine *RegistrationResponse*, die in Abbildung 5 dargestellt ist. In diesem Objekt liegt ein generierter API Schlüssel, der quasi das Passwort des Benutzers darstellt, um sich beispielsweise abzumelden.

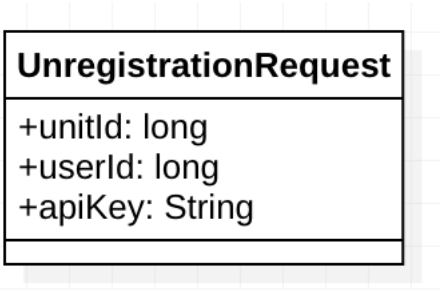

Abbildung 6. UnregistrationResponse

Um diese Abmeldung zu realisieren, muss die Schnittstelle eine weitere Funktionalität bieten. Diese erfolgt ebenfalls auf dem Endpunkt */registration*, nur ist dieses Mal die DELETE Methode erforderlich. Zusätzlich wird wieder im RequestBody ein Objekt mitgeschickt, das wie in Abbildung 6 aussehen muss. Als Antwort wird ein Http-Status 204 (No-content) zurückgeliefert, wenn die Registration gelöscht wurde. Falls nicht, kann ein beliebiger anderer Status zurückgegeben werden. Um aber möglichst nahe an REST zu bleiben, bieten sich insbesondere die Statuscodes der Bereiche 4xx (Client Fehler) und 5xx (Server Fehler) an.

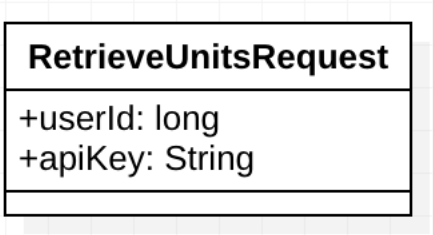

Abbildung 7. RetrieveUnitsRequest

Zuletzt muss diese Schnittstelle auch einen Endpunkt bieten, um alle aktuell registrierten Einheiten eines Benutzers auszulesen. Diese muss */units* mit POST angeboten werden. Das ist zwar nicht absolut REST-konform, aber die GET-Methode bietet keine Möglichkeit einen RequestBody anzuhängen. Parameter, die direkt in der URL mitgegeben werden, können unter Umständen abgehört werden. Im Body liegt ein RetrieveUnitsRequest, der in Abbildung 7 zu sehen ist.

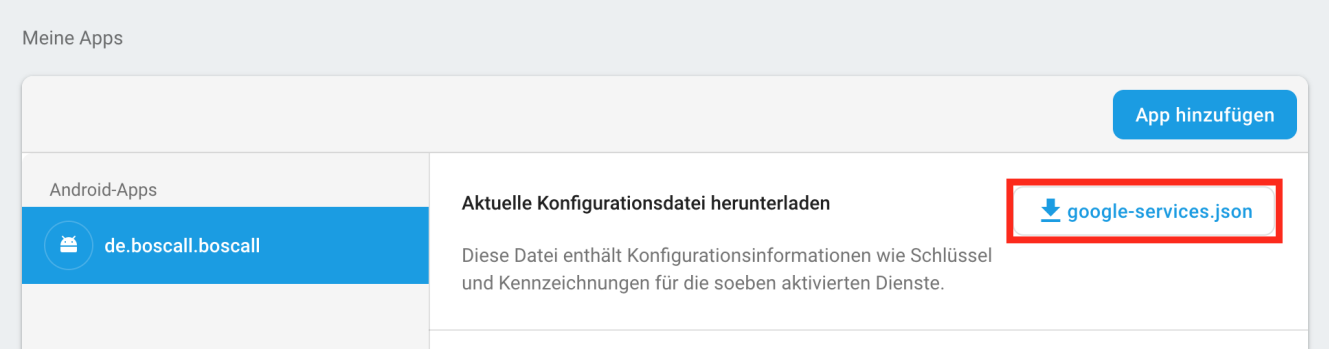

Abbildung 8. google-services.json Download

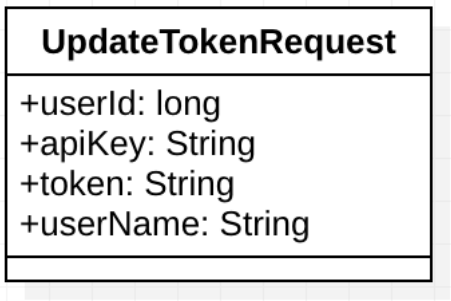

Abbildung 9. UpdateTokenRequest

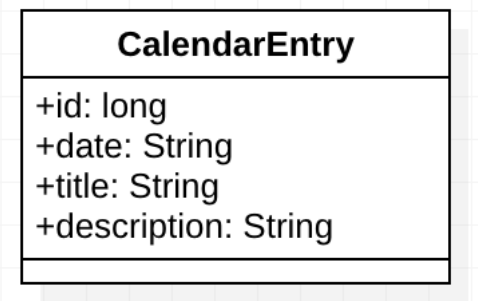

Abbildung 10. CalendarEntry

6.1.2 Tokenschnittstelle

Die Tokenschnittstelle dient der App dazu, bei einer Änderung eines Tokens eines Benutzers diesen in der Datenbank des Backends zu aktualisieren. Da der Token die einzige Möglichkeit ist, das Endgerät mittels Push Nachrichten zu adressieren sind solche Anfragen enorm wichtig. Zusätzlich wird der Endpunkt genutzt, um bei einer lokalen Änderung eines Benutzernamens diesen auch in der Datenbank des Backends zu aktualisieren. Der Endpunkt */updateToken* muss für die POST-Methode angeboten werden. Es wird hierbei ein UpdateTokenRequest, zu sehen in Abbildung 9, mitgeschickt. Als positives Ergebnis wird der Statuscode 200 (OK) erwartet.

6.1.3 Kalenderschnittstelle

Die letzte erforderliche Schnittstelle ist die Kalenderschnittstelle, die bei Anfrage alle Termine eines Benutzers zurückliefert. Dieser Endpunkt ist unter */calendar* mittels GET erreichbar und es wurde sich deutlich mehr an die Vorgaben des REST Stils gehalten. Hier gilt es in Zukunft zu evaluieren, ob die Sicherheit des API Schlüssels in den Parametern der URL gewährleistet ist. Jedenfalls wird die UserId und der API Schlüssel als Parameter in der URL übertragen. Ein Beispiel mit der UserId 42 und dem API Schlüssel 1234567890: */calendar/userId=42&apiKey=1234567890*.

Als Antwort erwartet die App eine Liste (in JSON ein Array - „[ ]“) aus CalendarEntry-Objekten. Diese sind aufgebaut wie in Abbildung 10 dargestellt.

6.2 Erforderliche Konfiguration der App

Damit die App später auch das richtige Backend anspricht, ist es erforderlich dessen Adresse anzupassen. Die Anpassung erfolgt in der Klasse *de.boscall.constants.ServiceConfiguration*. Die dortigen Anpassungen sind dann selbsterklärend.

Zuletzt muss dann in das „app“-Verzeichnis eine sog. „google-services.json“-Datei gelegt werden. Diese bekommt man in der Firebase Console. In den Einstellungen des Projekts befindet sich direkt auf der Startseite ein Button „google-servies.json“, mit dem man die Datei herunterladen kann. Der Button ist auch in Abbildung 8 farblich hervorgehoben. In der Datei sind Daten hinterlegt um mit Firebase kommunizieren zu können. Insbesondere der API Schlüssel und die Projektkennung.

7. Fazit

Zunächst lässt sich als Fazit sagen, dass die App funktioniert und aufgrund des extremen Kostenvorteils bereits in Kürze in den Testbetrieb bei der Feuerwehr Kusel genommen wird. Somit lässt sich grundlegend schluss-

folgern, dass das Projekt auf den ersten Blick erfolgreich war.

Die Probleme, die bei der Entwicklung auftraten, konnten wirksam behoben werden, dennoch sind einige unsaubere Lösungen im Code, die in späteren Iterationen entfernt werden sollen. Ein Beispiel wäre hier die Speicherung der Einheiten, die zurzeit noch als .json-Datei im Dateisystem erfolgt. Besser wäre es, wenn auch diese Funktionalität in Zukunft mittels Room gelöst wird.

Viel wichtiger als diese noch vorhandenen Notlösungen, die aber nach wie vor problemlos funktionieren, sind einige ungelöste Sicherheitsfragen. Beispielsweise sollten in naher Zukunft die Nachrichten die zwischen dem Alarmserver und dem Endgerät mittels Push ausgetauscht werden Ende-zu-Ende verschlüsselt werden. Dazu könnte man als Secret auch den API Key oder eben das Secret der Einheit nutzen, das im QR-Code bei der Registration hinterlegt ist. So kann einerseits sichergestellt werden, dass der Betreiber des Push Dienstes keinen Zugriff auf stark vertrauliche Daten (Einsatzdaten) hat, aber auch, dass die Daten bei einem Man-in-the-Middle Angriff nicht in falsche Hände gelangen. Alleine aus Gründen des Datenschutzes sollte das verhindert werden.

Ebenfalls sollten zukünftig alle Kommunikationen mit dem Backend nur noch auf Basis von SSL verschlüsselten HTTP Zugriffen (HTTPS) erfolgen. Da es sich hierbei aber zunächst um eine Probeanwendung handelt und auch noch keine SSL Zertifikate zur Verfügung standen, sollte zunächst die Kommunikation über HTTP erfolgen. Das ist bedeutend einfacher zu realisieren und für einen ersten Test auch völlig ausreichend. Für den späteren Produktivbetrieb sollte aber definitiv eine Umstellung auf HTTPS erfolgen.

Abbildungsverzeichnis

Literatur

[1] Alamos GmbH. firEmergency 2. [Online]. https://www.alamos-gmbh.com/service/fe2/ (Letzter Zugriff: 2018-06-19).

[2] A. GmbH. FE2 serverseitige Lizenzierung - Alamos GmbH. [Online]. https://www.alamos-gmbh.com/produkt/fe-2-serverseitige-lizenzierung/ (Letzter Zugriff: 2018-06-20).

[3] A. GmbH. Systemvoraussetzungen - Handbuch - Alamos GmbH. [Online]. http://docu.alamos-gmbh.com/display/documentation/Systemvoraussetzungen (Letzter Zugriff: 2018-06-20).

[4] Google GSON. google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back. [Online]. https://github.com/google/gson (Letzter Zugriff: 2018-06-23).

[5] A. Developers. Data and file storage overview — Android Developers. [Online]. https://developer.android.com/guide/topics/data/data-storage#db (Letzter Zugriff: 2018-06-23).

[6] Square. Retrofit - A type-safe HTTP client for Android and Java. [Online]. http://square.github.io/retrofit/ (Letzter Zugriff: 2018-06-23).