

**Mobile Anwendungen**

**Dokumentation**

**Gruppe M - Kontaktverwaltung**

Raphael Barth      Mike Sickmüller  
Marius Cerwenetz

Mannheim, Wintersemester 20/21

**Inhaltsverzeichnis**

<b>1 Einleitung</b>	<b>1</b>
<b>2 Architektur</b>	<b>1</b>
<b>3 PDF</b>	<b>1</b>
<b>4 RoomLifecycleService</b>	<b>2</b>
<b>5 RoomTag - QR und NFC</b>	<b>3</b>
<b>6 MQTT-Service</b>	<b>5</b>
<b>7 Datenbank</b>	<b>8</b>
<b>8 Fazit</b>	<b>8</b>
<b>9 Ausblick</b>	<b>9</b>

# 1 Einleitung

In der COVID-19-Pandemie sind Veranstaltungen häufig mit mehreren Teilnehmern mit einem erhöhten Infektionsrisiko verbunden, gerade in geschlossenen Räumen. Häufig besteht an die Veranstalter eine Pflicht Ein- und Austrittszeitpunkt aller Teilnehmer individuell festzuhalten um im Falle einer nachträglich auftretenden Infektion die Kontaktverfolgung für die Gesundheitsämter möglichst effizient zu gestalten. Eine Verwaltung mit Papierformularen ist mit einem hohen Digitalisierungsaufwand verbunden. Die App Kontaktverfolgung versucht dieses Problem zu lösen.

In der App kann eine Nutzerin oder ein Nutzer die Rolle eines Hosts, also eines Gastgebers oder eines Teilnehmers einnehmen. Jeder Nutzer hat ein Profil bestehend aus: Name, E-Mail und Telefonnummer. Hosts erstellen Räume die an eine reale Adresse gebunden sind. Teilnehmer können dann per NFC-Tag oder einen QR Code in diesen Raum eintreten. Ein und Austrittszeitpunkte werden erfasst und in der Datenbank vermerkt. Die Ein- und Austrittszeitpunkte der Räume können als PDF exportiert werden. Nach 14 Tagen werden die Daten aus der Datenbank gelöscht.

## 2 Architektur

## 3 PDF

Das System soll dem Nutzer die Möglichkeit bieten Informationen über einen Raum als PDF zu teilen. Dabei gibt es zwei unterschiedliche PDF's, diese beinhalten die generellen Rauminformationen, sowie den dazugehörigen QR-Code oder eine Liste aller Teilnehmer. Das Android-Framework verfügt über eine *PdfDocument* Klasse, die zum Erstellen von PDF's genutzt wird. Für eine A4 Seite nach DIN Norm ergeben sich die Werte 846x594 Pixel. Näheres zur Berechnung der Pixelwerte ist unter [developer.android](#) zu finden. Die erstellte Seite wird mittels eines Canvas editiert. Die Formatierung der Texte und Grafikelemente lassen sich über die *Paint* Klasse einstellen. Das erstellte PdfDokument wird im externen Dateiverzeichnis der App abgespeichert. Der Nutzer besitzt bei einem fehlgeschlagenen Teilen somit trotzdem die Möglichkeit auf das erstellte PDF zuzugreifen. Um das PDF zu teilen, wird ein *FileProvider* verwendet, welcher im Manifest deklariert ist. Der *FileProvider* bietet dem Nutzer eine Auswahl der installierten Anwendungen, über welche er das PDF

teilen kann.

Für die nächste Version der App muss das PDF mit der Übersicht aller Teilnehmer überarbeitet werden. In der aktuellen Version wird die E-Mail Adresse ab einer bestimmten Länge zum Teil mit in die nächste Spalte geschrieben. Ziel ist es, dass ab einer bestimmten Länge ein automatischer Zeilenumbruch in der entsprechenden Spalte stattfindet.

## 4 RoomLifecycleService

Der RoomLifecycleService steuert, wie der Name bereits andeutet, in welchem Zustand sich Räume befinden. Der Zustand bestimmt sich maßgeblich nach Öffnungs- und Schließzeit. Diese sind abhängig von der lokalen Zeit. Ein Raum soll ab einer gewissen Uhrzeit geöffnet und zu einer gewissen Uhrzeit geschlossen werden. Der LifecycleService gleicht also immer wieder ab, ob ein Raum gerade geöffnet oder geschlossen sein sollte. Er wird im Host und Teilnehmer-Modus benötigt. Die Entscheidung fiel auf einen Service, weil die App auch im nicht-geöffneten Zustand auf Timeouts und Öffnungszeiten reagieren sollte. Beispiel: Der Host eröffnet einen Raum und wechselt die App oder sperrt das Smartphone. Wenn die Kontaktverfolgungsapp nun nicht mehr im Vordergrund ist während das Timeout ausläuft oder der Raum in die Öffnungszeit reinrutscht, dann würde er nicht geschlossen respektive geöffnet werden. Um also zu vermeiden, dass der Host über die gesamte Zeit sein Smartphone nicht verwenden kann und das Display immer angeschaltet bleiben muss, was mit einem hohen Stromverbrauch verbunden ist, fiel die Entscheidung auf einen Service.

Wird der Service gestartet, wird die Raumüberprüfungsroutine nebenläufig in einem neuen Thread gestartet. Das nimmt die Last vom Hauptthread. Der Raumstatus wird jede Sekunde mit einem Busy-Wait-Algorithmus überprüft. Das sollte man nicht in den Hauptthread auslagern, da er diesen unnötig blockiert. Der Datenbankzugriff ist also nicht die Ursache für die Designentscheidung, denn der Zugriff wird im Repository ohnehin nebenläufig realisiert. Im Thread wird überprüft ob aktuell geschlossene Räume geöffnet und aktuell geöffnete Räume geschlossen werden sollten. Nebenher wird im Mainthread versucht den MQTT-Service zu einzubinden. Denn der Host soll natürlich auf die Topics aller Räume hören die offen sind bzw. aufhören auf alle Räume zu hören die geschlossen sind. Außerdem muss der Host den Teilnehmern die Raumeigenschaften und den Öffnungsstatus beim Öffnen mitteilen. Der

MQTT-Service wird hier also gebraucht um den Teilnehmern Raumstatusänderungen mitzuteilen. Dabei ruft der RoomLifecycleService die Methoden des MQTT Services nicht direkt auf. Die gesamte Überprüfung der Raumzustände erfolgt nebenläufig in einem anderen Thread. Es kann also vorkommen, dass der Thread schon über den MQTT Service Raumänderungen versendet obwohl er eigentlich noch nicht gebunden ist. In dem Fall kommt es zu einer Exception. Deshalb haben wir einen Handler eingeführt. Der Thread schreibt in eine threadsichere Liste die Räume rein, die ihren Zustand geändert haben und über MQTT versendet werden sollen. Ist der MQTT-Service im Hauptthread gebunden, dann arbeitet besagter Handler mit der Methode *postPendingRooms()* diese Räume im Hauptthread ab. Dadurch ist stets sichergestellt, dass der MQTT-Service nur Räume erhält wenn er gebunden ist. Das hätte man einfach vermeiden können, indem man statt zwei, einen einzigen Service genommen hätte. So hätte man gar nicht auf das Binden warten müssen.

Zurück zum LifeCycleService. Ein Raum hat drei mögliche Zustände:

1. ROOMISCLOSE
2. ROOMISOPEN
3. ROOMWILLOPEN

Jede Sekunde wird also überprüft ob Räume die geschlossen sind geöffnet werden sollen, ob Räume die sich öffnen werden geöffnet werden können und ob Räume die geöffnet sind geschlossen werden sollten. Falls noch Teilnehmer in zu schließenden Räumen eingebucht sind werden Sie rausgeworfen. In der Datenbank wird dann für alle Teilnehmer als Austrittszeitpunkt die Timeout-Zeit des Raums eingetragen.

## 5 RoomTag - QR und NFC

Damit Räume distinktiv sind besitzen Sie einen Identifier. Dieser nennt sich Room-Tag und ist zusammengesetzt aus dem Raumnamen, der Email-Adresse des Hosts und der Id des Raums in der Datenbank der Hostdatei. Beispiel für einen Room-Tags: Vorlesung 1/dozent@hs-mannheim.de/0. Teilnehmer benötigen diesen Tag um dem Raum beizutreten. Dies können Sie über NFC oder über einen QR-Code durchführen. Eine manuelle Eingabe ist nicht implementiert.

Um den Tag weiterzugeben kann der Veranstalter nach dem erstellen des Raumes ein PDF generieren und per E-Mail versenden. Das passiert in der

Methode `shareRoom()` in der Klasse `Activity_22_RoomHostDetail`. In einem solchen PDF ist neben Raum-Metadaten wie Titel, Start und End-Zeitpunkt auch ein QR-Code über den die Teilnehmer beitreten können. Für die Erstellung wird die externe Bibliothek `zxing` von `journeyapps` benutzt. Die Anbindung erfolgt die Klasse `BarcodeEncoder` in der Klasse `QRCodeManager`.

Ein Schreiben des NFC Codes auf einen NFC Sticker ist nicht implementiert. Apps wie NFC Tools des Unternehmens `wakdev` aus dem PlayStore schaffen hier aber Abhilfe. Dort kann der Tag in Plaintext-Format auf den NFC Tag geschrieben werden. Eine URI sollte nicht vergeben werden, da der Scanner nicht drauf reagieren würde.

Im Laufe der Entwicklung definierten wurde fest definiert, dass ein Veranstalter seinen eigenen Raum nicht betreten dürfte. Außerdem sollte ein Teilnehmer, nachdem er eine Sitzung verlassen hatte, nicht wieder erneut teilnehmen dürfen. Damit wurden irreführende Einträge in der Auswertung vermieden. Dieses Verhalten wird mit der Methode `checkEnterPeriomiission` in der `Activity_11_EnterViaQrNfc` abgefangen.

### 5.1 Betreten über QR

Wünscht der Teilnehmer über QR einzutreten, dann wird in der `Activity_11_EnterViaQrNfc` die Funktion `callScanner()` aus der `QRCodeManager`-Klasse gerufen. Dort wird die Scanner-Activity aus `zxing` gestartet, die einen Rückgabewert liefert. Es ist auch möglich den Scanner zu vorkonfigurieren, so, dass z.B. immer der Kamerablitzz angeschaltet ist um auch bei schlechten Licht-Bedingungen den QR-Code gut zu erkennen. Außerdem kann ein Timeout eingestellt werden, falls der Nutzer den QR Code in einer gewissen Zeit nicht einscanned. In unserem Projekt wurde sich aber wegen der Benutzerfreundlichkeit gegen diese Konfiguration entschieden. Der Scanner wird über einen Intent gestartet und liefert wenn er fertig ist ein Ergebnis zurück. Dieses wird in der Methode `onActivityResult` verarbeitet. Wenn der Scan geklappt hat wird die Methode `enterRoom()` aufgerufen. Dort werden die Daten des Room-Tags in die Datenbank eingetragen und die `Activity_14_RoomParticipantDetail` gestartet. Der Teilnehmer hört dort dann auf das korrekte MQTT Topic. Er hat den Raum betreten.

## 5.2 Betreten über NFC

Wünscht der Teilnehmer über einen NFC-Tag einzutreten, dann wird in der *Activity\_11\_EnterViaQrNfc* die Funktion *armNFCAdapter()* aufgerufen. Es wird überprüft ob das Smartphone überhaupt einen NFC Sensor hat. Falls das erfüllt ist wird dieser auf events des Types *ACTION\_NDEF\_DISCOVERED* scharfgeschaltet, also wenn ein neues Tag detektiert wurde. Man sagt auch er ist im Foreground-Dispatch-Modus. Das bedeutet dass, wenn ein neues NFC Tag detektiert wurde, ein gewisser Intent gestartet wird. Falls das der Fall ist, wird die gleiche *Activity\_11\_EnterViaQrNfc* noch einmal gestartet allerdings mit der Action *NFC\_INTENT\_ACTION*. In der *onResume()* wird überprüft ob diese Action gesetzt ist, denn nur dann kehrt die Anwendung ja aus dem NFC-Intent zurück. In den Extras des NFC-Intents befindet sich dann er abgelesene Room-Tag. Der Ablauf danach ist wie beim Betreten über QR. *enterRoom()* wird gerufen und *Activity\_14\_RoomParticipantDetail* wird gestartet.

## 6 MQTT-Service

Der MQTT-Service stellt die MQTT-Funktionalität bereit, über welche die App mit anderen Smartphones kommuniziert. Das Starten und Stoppen des MQTT-Service findet in der *Activity\_00\_StartActivity* statt. Alle anderen Activities binden lediglich den MQTT-Service. Das Binden des MQTT-Service findet in *Activity\_11\_EnterViaNfcQr*, *Activity\_14\_RoomParticipantsDetails* und *Activity\_22\_RoomHostDetails* statt. Des Weiteren wird der Service in dem *RoomLifecycle-Service* gebunden. Näheres hierzu unter [Abschnitt 4](#). Beim Verwenden des MQTT-Service wird in jeder Activity geprüft, ob dieser schon gebunden ist (not null). Ist dies nicht der Fall, wird das zu sendende Objekt zwischengespeichert und bei erfolgreichem Binden verarbeitet. Die im MQTT-Service implementierten Methoden sind ausführlich mittels [JavaDoc](#) erklärt.

Die Kommunikation zwischen dem Host und den Teilnehmern verläuft beim MQTT-Protokoll über sogenannte Topics. Für einen Raum werden, wie [Abbildung 1](#) zeigt, immer zwei Topics benötigt. Der Teilnehmer kann einem Raum über die Topic *moagm/Room-Tag* beitreten. Der Host sendet die nötigen Informationen für die Teilnehmer über die Topic *moagm/Room-Tag/public* an alle Teilnehmer. Durch die Trennung der Kanäle wird verhindert, dass Teilnehmer die Nachrichten, welche von anderen Teilnehmern an den Host gesendet werden, empfangen können.

Insgesamt gibt es vier verschiedene Nachrichtentypen, welche zwischen den

Teilnehmern und dem Host ausgetauscht werden. Alle Nachrichten werden im JSON-Format versendet. Dabei lassen sich die Nachrichten in zwei Gruppen unterteilen. Die Nachrichten *anmeldung.json* und *abmeldung.json* werden vom Teilnehmer an den Host gesendet. *teilnehmer.json* und *rauminfo.json* sendet der Host an alle Teilnehmer. Mittels eines Converters werden die entsprechenden JSON-Objekte erzeugt und wieder in verwendbare Datentypen zurück gewandelt. Zum Identifizieren eines Nachrichtentyps wird (warum auch immer..) nicht das Typfeld benutzt. Das Identifizieren wird mit den folgenden Schlüsselworten durchgeführt.

- ENTERTIME (*anmeldung.json*)
- EXITTIME (*abmeldung.json*)
- RAUM (*rauminfo.json*)
- TEILNEHMERLIST (*teilnehmer.json*)

Abhängig von dem empfangen Datentyp werden unterschiedliche Aktionen durchgeführt.

Mit der in [Listing 1](#) dargestellten *anmeldung.json* meldet sich ein Teilnehmer beim Host an und stellt eine Verbindung zu der entsprechenden Topic her. In der Nachricht sind alle relevanten Informationen des Teilnehmers enthalten sowie die Uhrzeit wann er den Raum betreten hat. Der MQTT-Service konvertiert die erhaltene JSON-Nachricht mittels der *AdapterJsonMqtt* Klasse in den passenden Datentyp. Das erhaltene *ParticipantItem* Objekt wird in der Datenbanktabelle *dbParticipant* mit der entsprechenden RaumId abgespeichert. Die entsprechende RaumId kann der Topic entnommen werden, über welche die JSON-Nachricht empfangen wurde. Im Anschluss wird an alle Teilnehmer ein *teilnehmer.json* sowie *rauminfo.json* gesendet. Die Nachricht *abmeldung.json*, dargestellt in [Listing 2](#), sendet der Teilnehmer an den Host, sobald er den Raum verlässt. Dabei meldet sich der Teilnehmer ebenfalls von der entsprechenden Topic der MQTT-Verbindung ab. Analog zum *anmelden.json* enthält diese Nachricht alle wichtigen Informationen über den Teilnehmer sowie die Uhrzeit zu welcher er den Raum verlässt. Im MQTT-Service wird mithilfe des erzeugten *ParticipantItem* Objekt und der RaumId die Exittime des dazugehörenden Teilnehmers aktualisiert.

[Listing 3](#) stellt die Nachricht vom Typ *rauminfo.json* da. Diese Nachricht enthält sämtliche Informationen über einen Raum. Jede vom Teilnehmer empfangene *rauminfo.json* wird in ein *RoomItem* Objekt konvertiert und in die Datenbanktabelle *dbRoom* eingefügt bzw. aktualisiert den bereits vorhandenen Eintrag. Wenn der Host eine *rauminfo.json* versendet, welche den Teilnehmern

## 6 MQTT-Service

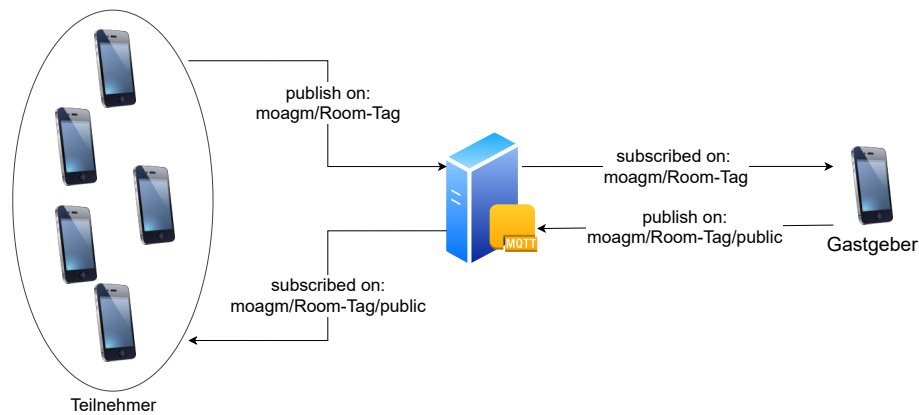


Abbildung 1: Darstellung des grundlegenden Kommunikationsablaufes mit MQTT-Topics.

mitteilt, dass der Raum nun geschlossen ist, meldet der MQTT-Service den Host automatisch von der entsprechenden MQTT-Topic ab.

Der in **Listing 4** dargestellte Nachrichtentyp *teilnehmer.json* beinhaltet eine Liste aller Teilnehmer eines Raumes. Bei Empfangen der Nachricht wird das darin enthaltene JSON-Array in eine Liste von *ParticipantItems* konvertiert. Die *ParticipantItems* der Liste werden mit der entsprechenden RaumId, welche über die Topic ermittelbar ist, in die Datenbanktabelle *dbParticipants* eingefügt. Um zu verhindern, dass Teilnehmer mehrfach eingetragen werden, wird die Anzahl der bereits eingetragenen Teilnehmer des Raumes ermittelt. Aus der Liste der erhaltenen Teilnehmer werden nur *ParticipantItems* eingefügt, welche einen mindestens gleich großen Index besitzen wie die ermittelte Anzahl. Diese Variante funktioniert, da sich die Reihenfolge der Teilnehmer in den Datenbanktabellen von Host und Teilnehmer nie ändert.

Listing 1: anmeldung.json

```

1 {
2   "TYPE": "LOGIN",
3   "TEILNEHMER": {
4     "NAME": "",
5     "EXTRA": "",
6     "EMAIL": "",
7     "PHONE": ""},
8   "EXITTIME": 0
9 }
  
```

Listing 2: abmeldung.json

```

1 {
2   "TYPE": "LOGIN",
3   "TEILNEHMER": {
4     "NAME": "",
5     "EXTRA": "",
6     "EMAIL": "",
7     "PHONE": ""},
8   "ENTERTIME": "0"
9 }
  
```



Tabelle 1: Struktur der Datenbanktabelle dbParticipant

id	name	extra	eMail	phone	roomId	enterTime	exitTime
1	Max Mustermann	null	mm@mail.de	123456789	1	0	0

Tabelle 2: Struktur der Datenbanktabelle dbRoom

id	fremdId	roomName	status	host	eMail	phone	place	address	extra	enterTime	exitTime
1	null	MusterRaum	1 (ROOMISCLOSE)	Max Mustermann	mm@mail.de	+123456789	12345 Musterstadt	Musterstr. 1	null	0	0

Listing 3: rauminfo.json

```

1 {
2   "TYPE": "RAUMINFO",
3   "RAUM": {
4     "ID": 0,
5     "ROOMNAME": "",
6     "STATUS": "",
7     "HOST": "",
8     "EMAIL": "",
9     "PHONE": "",
10    "PLACE": "",
11    "ADDRESS": "",
12    "EXTRA": "",
13    "ROOMSTARTTIME": 0,
14    "ROOMENDTIME": 0}
15 }
```

Listing 4: teilnehmer.json

```

1 {
2   "TYPE": "TEILNEHMER",
3   "TEILNEHMERLIST": [
4     {
5       "NAME": "",
6       "EXTRA": "",
7       "EMAIL": "",
8       "PHONE": "",
9       "ENTERTIME": 0,
10      "EXITTIME": 0},
11     ...
12   ]
13 }
```

## 7 Datenbank

## 8 Fazit

Die Anwendung ermöglicht eine effiziente, einfache Möglichkeit dezentral und mit wenig Ressourcen eine Kontaktverfolgung umzusetzen. Sie erfüllt die Grundsätze der Datenminimierung, Zweckbindung, Richtigkeit und Speicherbegrenzung der DSGVO. Nutzerdaten werden nur auf den Endgeräten gespeichert. Sie werden nur gespeichert, sollte ein Teilnehmer oder ein Host auch wirklich Kontakt mit diesem Nutzer gehabt haben. Des weiteren werden alle

Einträge die älter als 14 Tage sind aus der Datenbank gelöscht.

In der Entwicklung konnten die in der Vorlesung gelehrt Inhalte noch einmal selbstständig angewandt werden. Auch verschiedene Nutzungsweisen des Versionsverwaltungstools git konnten getestet werden. Am Anfang wurde hauptsächlich auf einem Branch entwickelt, später wurde auf ein Feature-Based-Setup gewechselt in dem jeder Entwickler einen eigenen Branch pro Feature hatten. Zum Schluss wurde sich für einen dev-branch für experimentelle Änderungen und einen master-branch für stabile Versionen entschieden, da dies gerade für die Demos sehr von Vorteil war.

Das Entwickler-Team war stets motiviert. Aufgaben wurden termingerecht und zuverlässig erledigt. Fragen und Probleme wurden im Plenum behandelt. Bei sehr komplizierten Problemen wurde der Dozent konsultiert. Teambesprechungen erfolgten meist über Jitsi. Für die Aufgabenverwaltung wurde eine Mischung aus dem "Issue"-Aufgabenverwaltungstool von Gogs und Todos im Code verwendet.

## 9 Ausblick

Mögliche Features im Bereich NFC wären die Beschreibung eines Tags aus der App heraus, um so auf Drittanbieter-Apps verzichten zu können. Außerdem wäre eine manuelle Uri sinnvoll. So würde der NFC Scannern nicht auf jeden Tag reagieren, sondern nur auf solche die er auch beschrieben hat. Ein weiteres Feature wäre es manuell den Room-Tag eines Raumes einzugeben um ihn zu betreten. Diese Möglichkeit gibt es momentan noch nicht.

Ein anderes Feature wäre die Implementierung eines optionalen Profilbildes im Nutzerprofil.

Im Falle einer Infektion werden Nutzer bisher noch nicht benachrichtigt. Auch das wäre wünschenswert.

Das MQTT-Protokoll war für den Nachrichtenaustausch etwas suboptimal. Es kam vereinzelt vor dass Nachrichten bei manchen Teilnehmern nicht ankamen. Auch kamen manche Nachrichten vom Host nicht bei Teilnehmern an. Die Implementierung einer Art Empfangsbestätigung wäre von Vorteil.

Sonstiges Die App ist in der Lage bei einem Verlust der Internetverbindung dem Nutzer ein Feedback mitzuteilen und ein hartes Abbrechen zu verhindern. Dafür wird ein BroadcastReceiver verwendet welcher Auf Netzwerkänderungen hört. Der BroadcastReceiver wird beim Erstellen des RoomLivecy-

## 9 Ausblick

clesServices an diesen gebunden. Bricht die Internetverbindung weg werden alle Services gestoppt. Des Weiteren wird dem Nutzer mitgeteilt das die Internetverbindung verloren wurde und er muss die App mit einer bestehen Internetverbindung neu starten muss.

Sämtliche benötigten Informationen über den Nutzer der App werden mittels Preferences erfasst. Ohne das vollständige Ausfüllen der Preferences ist in Verwenden der App nicht möglich. Um später einen eleganten Zugriff auf die Preferences zu erhalten ist eine MySelf Klasse implementiert.