

Pförtner - Das digitale Türschild

Technische Dokumentation

Gruppe 26: Marc Oliver Arnold <marc.arnold@stud.tu-darmstadt.de>
Jonas Johannes Franz Belouadi
<jonasjohannesfranz.belouadi@stud.tu-darmstadt.de>
Anton Wolf Haubner <anton.haubner@outlook.de>
David Heck <david.heck@stud.tu-darmstadt.de>
Martin Kerscher <martin.kerscher@stud.tu-darmstadt.de>

Auftraggeber: Christian Meurisch <meurisch.tk.tu-darmstadt.de>
Sebastian Kauschke <kauschke@tk.tu-darmstadt.de>
Stefan Wullkotte <wullkotte@tk.tu-darmstadt.de>
Telecooperation Lab
Fachbereich Informatik

Abgabedatum: 19.03.2019



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Internet Praktikum Telekooperation
Fachbereich Informatik

Inhaltsverzeichnis

1. Einleitung und Motivation	2
2. Übersicht	3
2.1. Features	3
2.1.1. Büroinformationen	3
2.1.2. Mitarbeiterinformationen	3
2.1.3. Terminvereinbarungen	4
2.1.4. Spion	5
2.2. Architektur	5
2.2.1. Komponenten	5
2.2.2. Datenbanken	6
2.2.3. Kommunikation	6
3. Ausgewählte technische Aspekte	10
3.1. Initialisierungsvorgang	10
3.1.1. Motivation	10
3.1.2. Konzept	10
3.2. Synchronisierung	14
3.2.1. Motivation	14
3.2.2. Konzept	14
3.2.3. Eingesetzte Technik	14
3.2.4. Reaktive Programmierung	15
3.3. Google Kalender Einbindung	15
3.3.1. OAuth2 Authentifizierung bei Google	15
3.3.2. Push Notifications	16
A. Anhang	18

1 Einleitung und Motivation

Statische Türschilder lassen sich nur aufwendig anpassen und zeigen daher nicht immer aktuelle Informationen. Mithilfe von Pförtner lässt sich fast jedes Endverbrauchertablet¹ ab Android 7 als digitales Türschild einsetzen. Über eine Smartphone App sind die Mitarbeiter in der Lage, die Inhalte des Pförtner-Türschilds anzupassen. Neben den üblichen Informationen, welche auf Türschildern dargestellt werden, wie Raumnummer und Namen der Mitarbeiter, erlaubt die App es den Büromitarbeiter topaktuelle Informationen wie den Anwesenheitsstatus anzuzeigen.

Ist beispielsweise ein Mitarbeiter krank, so kann er dies mithilfe von Pförtner von Zuhause auf dem Türschild vermerken und Besucher sind sofort informiert.

Weiterhin transformiert Pförtner Türschilder zu einem interaktiven Medium für die Besucher: Pförtner integriert sich mit Google Calendar, sodass Besucher am Türschild sich für Termine anmelden können, welche die Mitarbeiter in ihrem Kalender angelegt haben.

¹ Notwendige Features: Frontkamera, Near Field Communication, installierte Google Play Services [16]

2 Übersicht

2.1 Features

Diese Sektion dient als Übersicht über alle Features, welche im Verlauf des Praktikums in eine Userstory gefasst und realisiert wurden. Die Features werden aus Nutzersicht motiviert und beschrieben. Die technische Realisierung besonders wichtiger Aspekte wird in Kapitel 3 beschrieben. Die Anwendung, welche auf dem Tablet ausgeführt wird, werden wir als *Panel-App* bezeichnen und die Anwendung, welche auf den Smartphones der Büromitarbeiter ausgeführt wird, als *Admin-App*.

2.1.1 Büroinformationen

Da unsere Panel-App als elektronisches Türschild etablierte Lösungen ablösen möchte, muss es zumindest den gleichen Informationsgrad bieten. Deshalb ist die Bereitstellung von Informationen über den korrespondierenden Raum eine der wichtigsten Funktionen Pförtners. Dazu gehören die verschiedenen Mitglieder eines Büros, die Raumnummer und der Verfügbarkeitsstatus. Im Gegensatz zu statischen Türschildern bietet unsere Panel-App die Möglichkeit, die angezeigten Büroinformationen jederzeit bequem zu ändern. Mitglieder des Büros können einfach und schnell über ihrer Admin-Apps neue Büroinformationen setzen (vgl. Abb. 2.1 und 2.2). Die Informationen werden auf der Panel-App dann automatisch dargestellt.

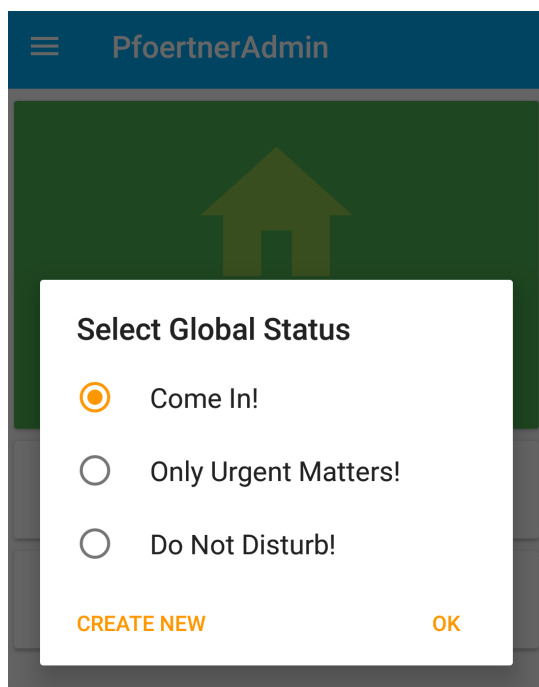


Abbildung 2.1.: Ansicht der App zum Setzen eines neuen Bürosstatus

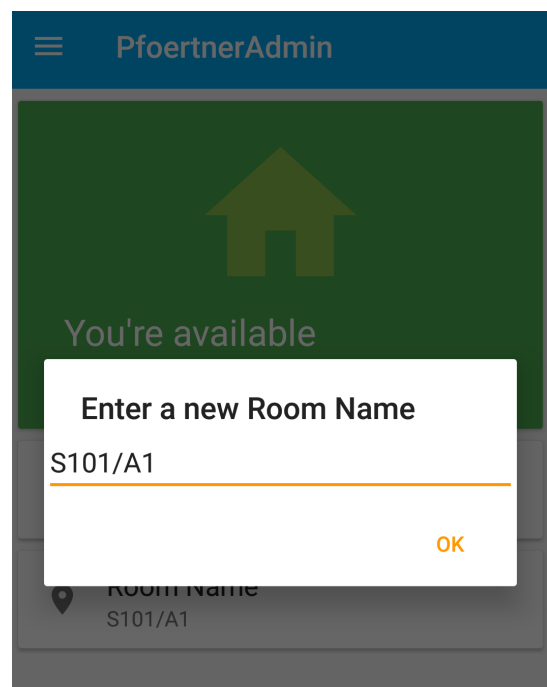


Abbildung 2.2.: Ansicht der App zum Setzen einer neuen Raumnummer

2.1.2 Mitarbeiterinformationen

Über eine Anzeige der Büroinformationen (2.1.1) und einfache Auflistung der Büromitglieder hinaus sollen natürlich auch detailliertere Informationen über diese angezeigt werden. Unsere Panel-App kann den Namen,

die Sprechzeiten und ein Foto des Mitarbeiters anzeigen. Zusätzlich ist es möglich, diese Informationen schnell und einfach mittels der Admin-App abzuändern (vgl. Abb. 2.3). Sobald Informationen neu gesetzt wurden, aktualisiert sich die Panel-App auch hier automatisch.

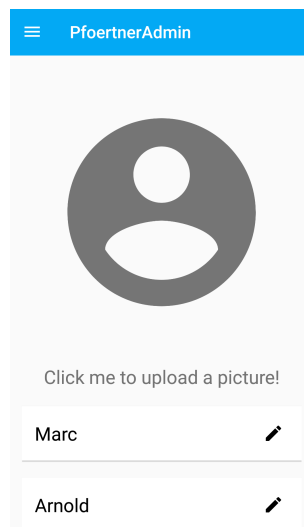


Abbildung 2.3.: Übersicht der persönlichen Einstellungen

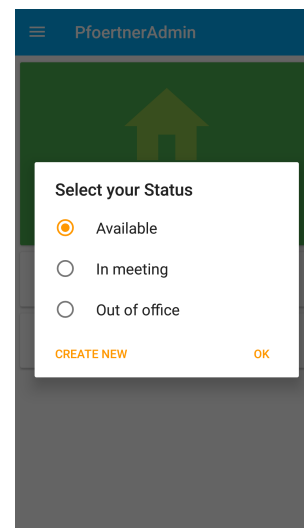


Abbildung 2.4.: Setzen des persönlichen Status

2.1.3 Terminvereinbarungen

Des Weiteren ist es möglich, über die Panel-App Terminanfragen an die Büromitglieder zu senden. Sollte ein Büromitglied nicht vor Ort antreffbar sein, so kann mit dieser Funktion für einen späteren Zeitpunkt ein Termin vereinbart werden. Die Person, welche ein Büromitglied nicht antreffen konnte, erhält mittels der Panel-App eine Liste von Terminen der gesuchten Person (vgl. Abb. 2.5). Dort kann Sie für einen oder mehrere Termine eine Terminanfrage senden (vgl. Abb. 2.6). Mittels einer Notification wird das Büromitglied, an welches die Terminanfrage gesendet wurde, auf dem eigenen Smartphone über die Terminanfrage informiert und kann dann entscheiden, ob diese angenommen wird oder nicht. Falls die anfragende Person eine Email bei der Terminanfrage angegeben hat, wird sie eine Email mit dem Ergebnis der Anfrage (Abgelehnt/Zugestimmt) erhalten. Dies eröffnet einen großen Vorteil gegenüber den etablierten Türschildern, welche keinerlei Form von Terminvereinbarung anbieten.

Room Name Not Set		Appointment times for: Marc Arnold		
13-3	Wed	16:00	-	17:00
14-3	Thu	17:00	-	18:00
15-3	Fri	18:00	-	19:00
16-3	Sat			
17-3	Sun			
18-3	Mon			
19-3	Tue			
20-3	Wed			
21-3	Thu			
22-3	Fri			
23-3	Sat			
24-3	Sun			

Abbildung 2.5.: Ansicht aller Termine für eine Terminanfrage

One last step required

Please enter your name, mail and a message.

🕒

18:00 - 19:00

👤

Enter your full name here

✉️

Enter your email address here

💬

Enter a message

REQUEST APPOINTMENT

Abbildung 2.6.: Ansicht des Terminanfrageformulars

2.1.4 Spion

Ein weiteres Feature der Panel-App ist der sogenannte „Spion“. Oftmals befindet man sich in der Situation, dass man gerade konzentriert am Arbeiten ist und es an der Tür klopft. Je nach dem welche Person vor der Tür steht, möchte man mitunter die Tür lieber nicht öffnen. Problematisch dabei ist, dass man erst in Erfahrung bringen kann, wer an der Tür ist, wenn man diese öffnet. Dieses Problem löst der Spion. Die Spion Funktion ermöglicht es, über die Admin-App eine Fotoaufnahme über die Frontkamera des Türschilds zu starten. Das aufgenommene Foto wird dann auf der Admin-App angezeigt. Auf diese Weise kann man beruhigt entscheiden, ob man die Tür öffnen möchte.

2.2 Architektur

2.2.1 Komponenten

Pförtner besteht aus 3 Komponenten:

1. Panel-App
2. Admin-App
3. Server

Apps

Wie bereits erläutert, umfasst Pförtner eine Android App, welche auf einem Tablet zu installieren ist und die als Türschild agiert, die Panel-App. Weiterhin installieren alle Büromitglieder eine Android App, welche es erlaubt, die auf dem Türschild dargestellten Informationen anzupassen, die Admin-App.

Die beiden Apps finden sich in den Quelldateien realisiert als getrennte Android Projekte in den Ordnern PfoertnerPanel und PfoertnerAdmin. Zum Öffnen und Kompilieren der Projekte empfiehlt sich die Verwendung der Software Android Studio [3].

Da sich beide Apps viele Funktionalitäten teilen, wurden diese in eine Bibliothek ausgelagert, die durch beide Apps verwendet wird. Dabei handelt es sich z. B. um eine Datenbankschnittstelle, da beide Apps zumeist auf die gleichen Daten zugreifen müssen (siehe 3.2).

Diese Bibliothek befindet sich ebenfalls als Android Projekt im Ordner PfoertnerCommon.

Server

Um einen Kommunikationsweg zwischen den beteiligten Geräten eines Büros (Panel-App und Admin-Apps der Büromitglieder) herzustellen, verwenden wir einen zentralen Server. Dieser wurde mithilfe des Node.js [9] Web Frameworks *Express* [19] entwickelt und das zugehörige Projekt befindet sich im Ordner *PfoertnerServer*. Instruktionen zur Ausführung des Servers sind in der Datei *ReadMe.md* beigelegt.

Pförtner wurde mittels dieser Technologien entwickelt, aufgrund deren ereignisgesteuerter Architektur. Diese erlaubt es, viele Anfragen nebenläufig abzuarbeiten. Dies ist notwendig, da Pförtner auch in der Lage sein soll, mehrere Bürotürschilder und deren zugehörige Büromitglieder zu verwalten. Weiterhin sind über den Node Package Manager [27] über 350.000 Softwarepakete [8] verfügbar, auf welche bei der Entwicklung von Pförtner zurückgegriffen werden kann. Für eine Liste verwendeter Bibliotheken siehe auch das Verzeichnis verwendeter Softwarebibliotheken [Link]. Auch wird Node.js aktiv in Unternehmen wie Netflix [7] oder PayPal [28] eingesetzt.

2.2.2 Datenbanken

Sowohl der Server, als auch die Apps müssen strukturierte Informationen speichern (Siehe 3.2). Der Einsatz einer relationalen Datenbank, erlaubt es Beziehungen zwischen den Entitäten der Domäne (Büros, Büromitglieder, Geräte) zu organisieren und gezielt abzufragen. Die wichtigsten Entitäten der Domäne, welche sich in der Datenbank wiederfinden und ihre Beziehungen untereinander sind im Entity-Relationship Diagramm 2.7 dargestellt.

Hierbei modelliert die Entität *Device* genau eine Installation einer Admin oder Panel App auf einem Smartphone. Sie wird z. B. benötigt, um Notifications (siehe 2.2.3) an ein ganz bestimmtes Gerät zu versenden. Die Entität *Office* entspricht einem Büro, an welchem ein Pförtner-Türschild angebracht ist. Seine Attribute speichern daher z. B. die aktuelle Statusmeldung für das Büro („Do Not Disturb“). Die *Office*-Entität steht in Beziehung zu einem ganz bestimmten *Device*, welches dem Tablet entspricht, auf dem die Panel-App für das Büro installiert wurde. Die Entität *OfficeMember* entspricht einer Person, welche im Büro arbeitet und auf dem Türschild angezeigt werden soll. Entsprechend speichern die Attribute z. B. den Namen der Person oder auch ein Bild. Bei der Beziehung zu *Device* handelt es sich um eine 1:1 Beziehung, da in der aktuellen Implementation das Türschild nur über eine Admin-App Installation je Nutzer gesteuert werden kann. Die Aufteilung in zwei Entitäten *Device* und *Officemember* erleichtert allerdings die Implementation der Unterstützung von mehreren Geräten je Nutzer, sollte dies von den Nutzern gewünscht werden. Zuletzt modelliert die Entität *AppointmentRequest* eine Terminanfrage durch einen Besucher (siehe auch 2.1.3).

Wir haben uns sowohl auf dem Server als auch auf den Android Apps zum Einsatz einer *SQLite* [18] Datenbank entschieden, da es nicht den Betrieb eines DBMS¹ Servers wie bei z. B. *MySQL* [26] erfordert.

Dies ist vor allem für den Einsatz innerhalb der Android Apps entscheidend, da die Verteilung derselben mit einem DBMS Server mit hohem Aufwand und Komplexität verbunden wäre.

Da der Server das *Sequelize* [32] ORM² für den Zugriff auf die Datenbank nutzt, welcher vom eingesetzten DBMS abstrahiert, kann auf der Seite des Servers leicht *SQLite* durch ein anderes SQL basiertes DBMS ausgetauscht werden.

Auf Seiten der Android Apps wird die Bibliothek *Room* [31] für den Zugriff auf die Datenbank eingesetzt, für weitere Details zum Hintergrund dieser Entscheidung siehe auch Abschnitt 3.2.4.

2.2.3 Kommunikation

HTTPS

Für die Übertragung von Inhalten (Bilder, Nutzerstatus, etc.) von Apps zum Server bzw. das Herunterladen von Inhalten vom Server (siehe auch 3.2) wird das HTTPS Protokoll eingesetzt. Dieses bietet uns die folgenden Vorteile:

¹ Datenbankmanagementsystem

² Object Relational Mapping

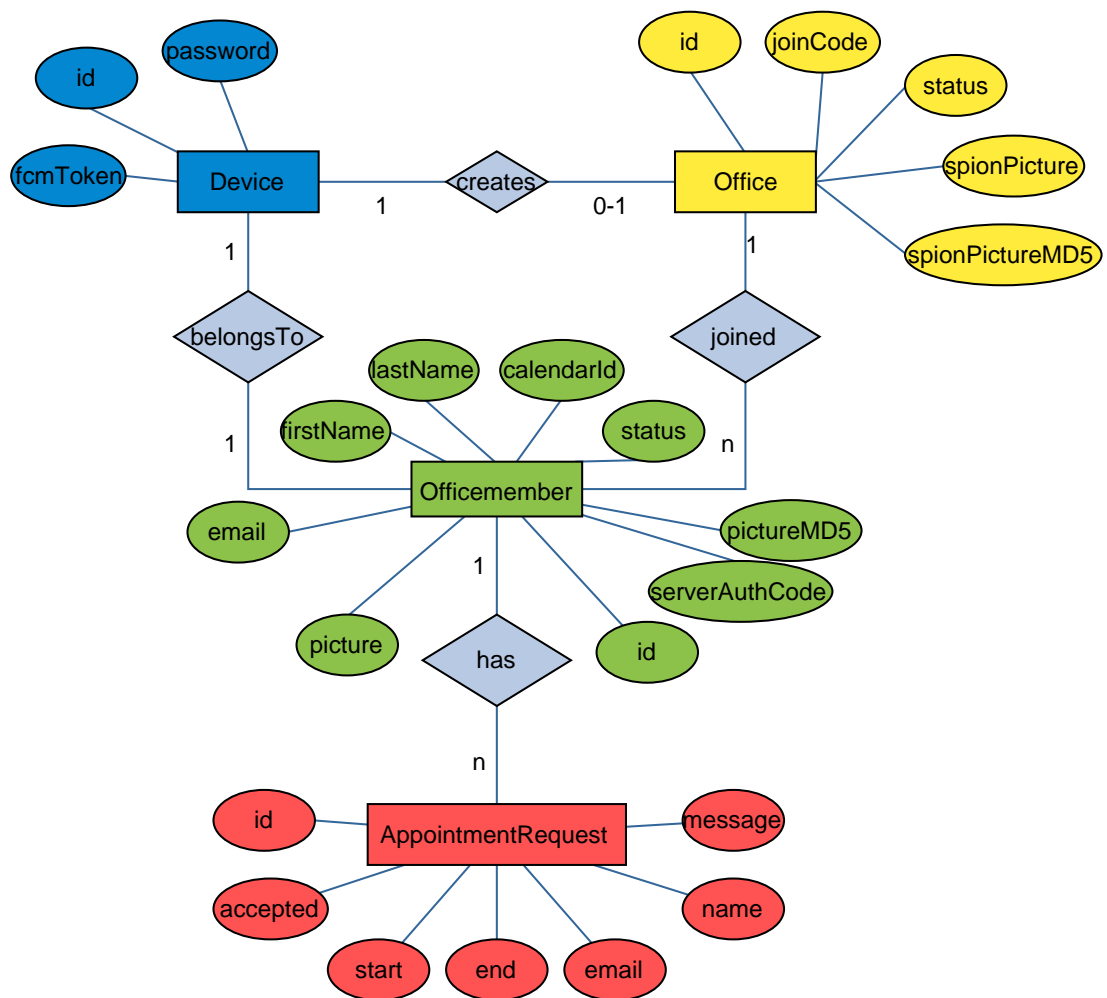


Abbildung 2.7.: ER-Diagramm des Datenmodells des Nodejs-Servers

- *Wartbarkeit*: HTTP ist als Basis des WWW weit verbreitet. Alle Pfoertner Entwickler waren bereits damit vertraut. Durch die Anlehnung der HTTP Api an das REST Programmierparadigma (siehe ??) konnte die Serverschnittstelle leicht verständlich und somit wartbarer umgesetzt werden.
- *Verfügbarkeit hervorragender Bibliotheken*: Wie bereits genannt, verwenden wir serverseitig das Express Framework (> 42.000 Sterne, > 7000 Forks auf GitHub [11]) und innerhalb der Apps den Retrofit [33] Client (> 31.000 Sterne, > 5000 Forks auf GitHub [[retrofitgithub](#)]).
- *Einfach abzusichern*: HTTPS bietet sowohl Verschlüsselung als auch Authentifizierung. Da die Apps und der Server auch Geheimnisse austauschen (siehe 3.1) ist eine Absicherung notwendig.

Das für den Einsatz von HTTPS notwendige Zertifikat, welches der Servers zur Authentifizierung beim Client nutzt, empfehlen wir von der Zertifizierungsstelle Let's Encrypt [23] zu beziehen. Diese Möglichkeit wurde auch während der Entwicklung von Pfoertner genutzt.

Im Gegensatz zur Nutzung eines selbst signierten Zertifikats bietet die Nutzung eines kostenfreien Zertifikats von Let's Encrypt verschiedene Vorteile. Das Zertifikat muss nicht zuvor an die Endgeräte verteilt und den vertrauten Zertifikaten hinzugefügt werden. Selbst wenn die Pfoertner Apps das Zertifikat beinhalten und selbst überprüfen würden, so könnte man auch nicht auf die Vorteile der PKI³ der Zertifizierungsstellen zurückgreifen. Ein Beispiel eines solchen Vorteils wäre die automatisierte Überprüfung, ob das Zertifikat zurückgezogen wurde.

Firebase Cloud Messaging

Firebase Cloud Messaging [24] (im Folgenden *FCM*) ist eine Cloud-basierte Lösung für den Versand von Nachrichten und Notifications zwischen Android, iOS und Webanwendungen.

Pfoertner setzt FCM ein, um die Android Apps über Ereignisse (siehe 3.2), wie Terminanfragen (siehe 2.1.3) zu informieren, ohne, dass der Server deren IP-Adressen kennen muss oder diese eine dauerhafte Verbindung zum Server aufrechterhalten müssen. Da die Google Play Services [16], welche auf fast jedem Android Endgerät installiert sind, automatisiert den Empfang von Nachrichten und das Starten der Empfängerapp übernehmen, muss also auf diese Weise für Pfoertner keine eigene Implementation von Push Notifications [22, 6] vorgenommen werden.

Google Kalender

Die Kommunikation mit dem Google Kalender erfolgt auf 2 Wegen: Einerseits mittels des Google Calendar Provider [13] und andererseits über die Google Calendar API [12]. Erstere erfordert es, dass die Google Kalender App installiert ist, erlaubt es aber, sehr einfach einen Kalender mittels der Android Programmierschnittstellen zu verändern. Daher verwenden wir den Calendar Provider, um Termine in die Kalender von Büromitgliedern mittels der Admin App einzutragen (siehe auch 2.1.3).

Im Falle der Panel-App, welche auf dem Türschild ausgeführt wird, ist die Verwendung des Calendar Provider daher allerdings keine Option. Nicht alle Büromitarbeiter sollen sich in der Google Calendar App auf dem Türschild anmelden müssen. Daher erfolgt hier der Abruf von z. B. Sprechstundenzeiten aus den Kalendern der Büromitglieder über die Nutzung der Google Calendar API, welche in Sektion 3.3 näher erläutert wird.

Das Diagramm 2.8 bietet einen Überblick über alle beteiligten Komponenten und Akteure der Architektur Pfoertners und auch der Kommunikationswege zwischen diesen.

³ Public Key Infrastructure

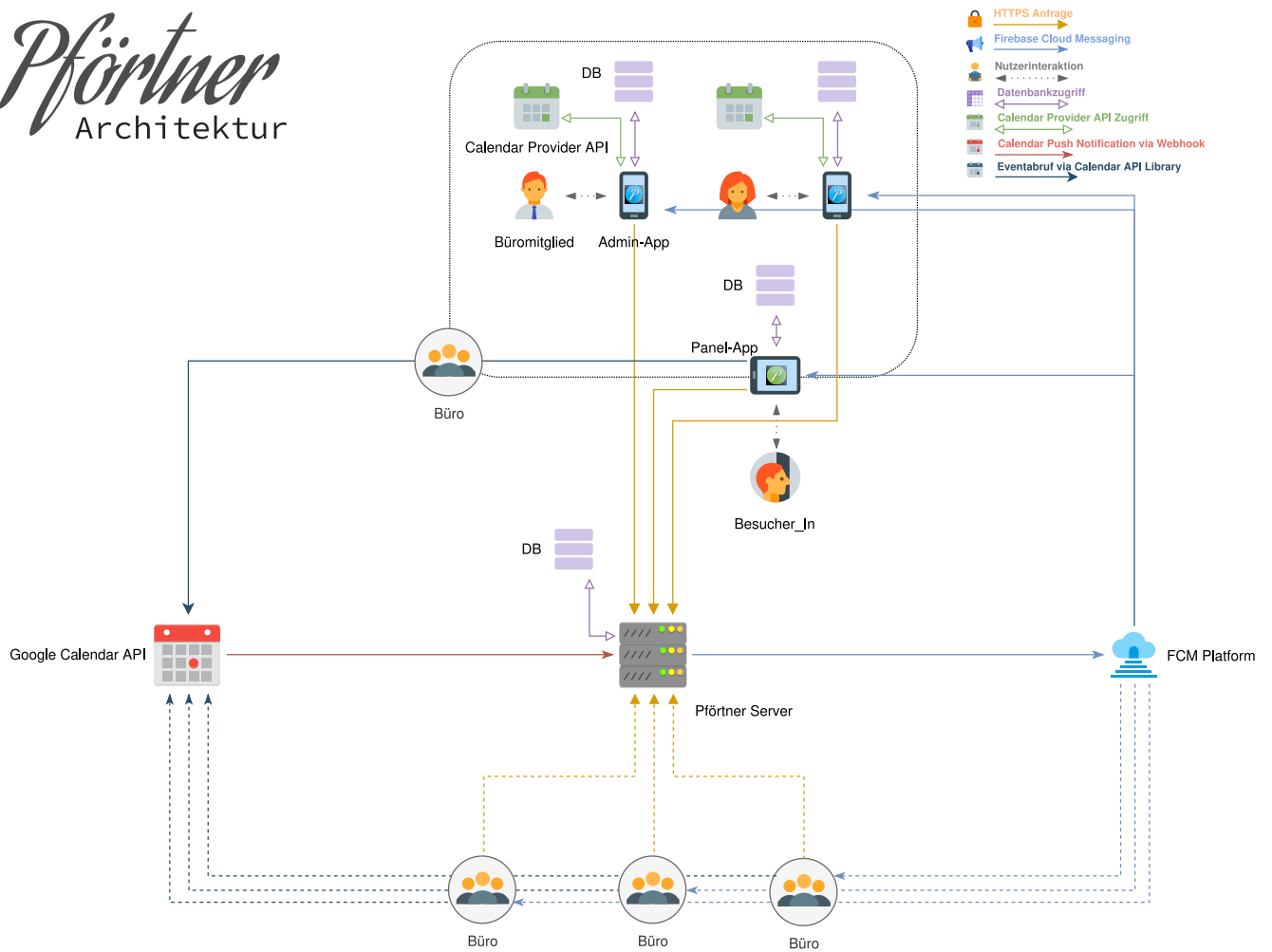


Abbildung 2.8.: Architektordiagramm, Icons entnommen aus [25, 34]

3 Ausgewählte technische Aspekte

3.1 Initialisierungsvorgang

3.1.1 Motivation

Damit Personen mit der Admin-App einem Büro beitreten und angezeigte Daten mittels der App verändern können, müssen die Admin-Apps und die Panel-App mit einem Büro assoziiert werden. Dieser Vorgang stellt somit die Grundlage aller weiteren Funktionalitäten dar und wird daher in diesem Abschnitt detailliert betrachtet.

3.1.2 Konzept

1. Nach dem Starten generieren die Apps zuerst ein zufälliges, persönliches Passwort, mit welchem sie sich beim Server unter einer eindeutigen Id registrieren (siehe die `Device`-Entität im Abschnitt 2.2.2), welche der Server vergibt. Dies erlaubt es dem Server, einzelne Geräte zu identifizieren, um diese Büromitgliedern zuzuordnen und gezielt Informationen an diese zu versenden. Weiterhin soll es nicht möglich sein, die Daten von Büromitgliedern zu verändern, wenn man nicht Zugriff auf deren Smartphone besitzt, was die Verwendung eines Geheimnisses (Passwort) erfordert.
2. Mithilfe des Passworts und der Id fordern die Apps Authentifizierungstokens beim Server an. Alle weiteren Anfragen beim Server erfolgen mittels dieser Tokens, sodass der Server feststellen kann, ob das Gerät autorisiert ist, die Anfrage durchzuführen.

Das Sequenzendiagramm 3.3 zeigt detailliert, wie dieser erste Abschnitt der Initialisierung implementiert wurde. Zu beachten ist hier, dass der dargestellte Ablauf bei jedem Start einer App erfolgt, da die geladenen Daten, wie das Authentifizierungstoken, für fast jede weitere Funktion benötigt werden. Daher überprüfen die dedizierten Datenklassen wie `Authentication`, ob die benötigten Daten nicht bereits generiert wurden und laden sie in diesem Fall vom lokalen Speicher.

(Hinweis: Die dargestellten Teilnehmer `Authentication`, `Password`, `User` und `Office` sind z. T. noch in Klassen außerhalb des in 3.2.4 beschriebenen Repository-Patterns implementiert, da sie bereits vor dessen Einführung fest in den Initialisierungsvorgang integriert waren und für dessen Funktion bedeutend sind. Wir empfehlen jedoch, bei Weiterführung des Pförtner-Projekts, auch diese Klassen dem Pattern anzupassen. Dies ist z. B. für `Office` bereits erfolgt, die alte Klasse wurde jedoch noch nicht aus allen Verwendungen entfernt.)

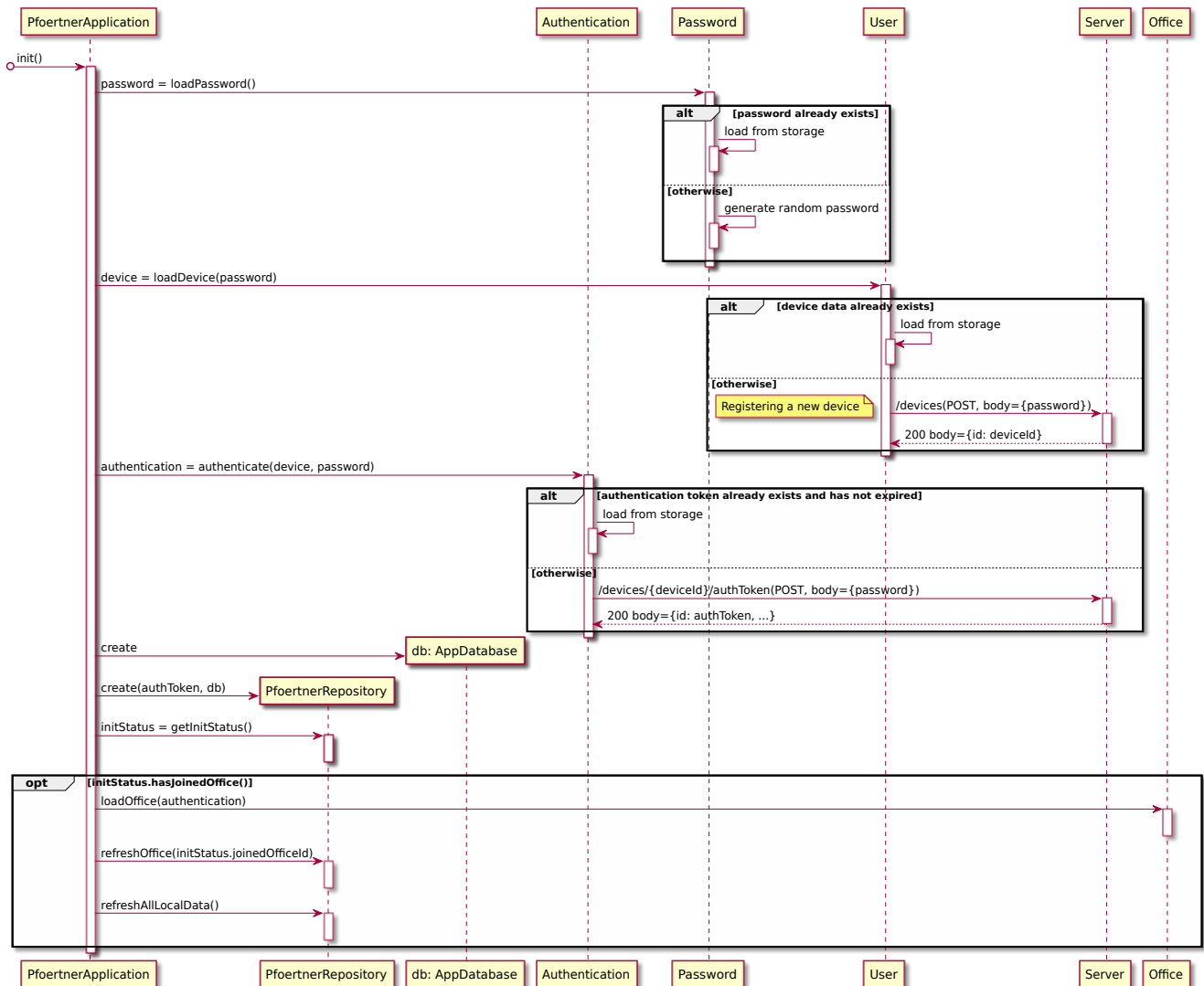


Abbildung 3.1.: Detaillierterer Ablauf des ersten Initialisierungsabschnitts, welcher in beiden Apps gleich ist. Die Implementierung befindet sich daher in PfoertnerCommon, siehe 2.2.1

Die nächsten Schritte unterscheiden sich, je nachdem ob es sich um die Initialisierung aus der Sicht der Panel App oder der Admin App handelt:

Panel-App

1. Die Panel-App registriert mittels einer HTTPS Anfrage beim Server ein neues Büro und erhält vom Server eine Identifikationsnummer für das Büro (*Büro Id*), sowie eine zufällige Zeichenfolge (*Join Code*). Diese Daten werden von der *Admin-App* benötigt, sodass sie sich einem Büro zuordnen kann, dessen Panel-App sie steuert.
2. Die Panel-App kodiert die Büro Id und den Join Code in einen QR Code [21] und zeigt diesen an (vgl. Abb 3.2). Weitere Interaktion ist mit der Panel-App erst möglich, sobald eine Admin-App sich mittels der Büro Id und dem Join Code registriert hat.
3. Sobald eine Admin-App sich registriert hat, wird statt dem QR Code der Hauptbildschirm der Panel-App angezeigt, welcher neben den Büroinformationen die Mitgliedsinformationen der beigetretenen Admin-Apps anzeigt.

Please scan this code using the **admin app**



Abbildung 3.2.: Startbildschirm der Panel-App

Admin-App

1. Die Admin-App bietet dem Nutzer das Scannen des oben genannten QR Codes mittels der Kamera des Smartphones an.
2. Nach dem Scannen des Code muss der Nutzer noch seinen Namen angeben
3. Die Admin-App tritt dem Office mittels der gescannten Daten durch eine HTTPS-Anfrage beim Server bei. Der Hauptbildschirm der Admin-App wird nun angezeigt.

Eine detailliertes Sequenzendiagramm der Implementation des geschilderten, Admin-App spezifischen Initialisierungsvorgangs ist in Abbildung 3.4 gegeben. Abbildung 3.3 zeigt eine Übersicht, welche Teile der Initialisierung in welcher App oder in beiden implementiert sind.

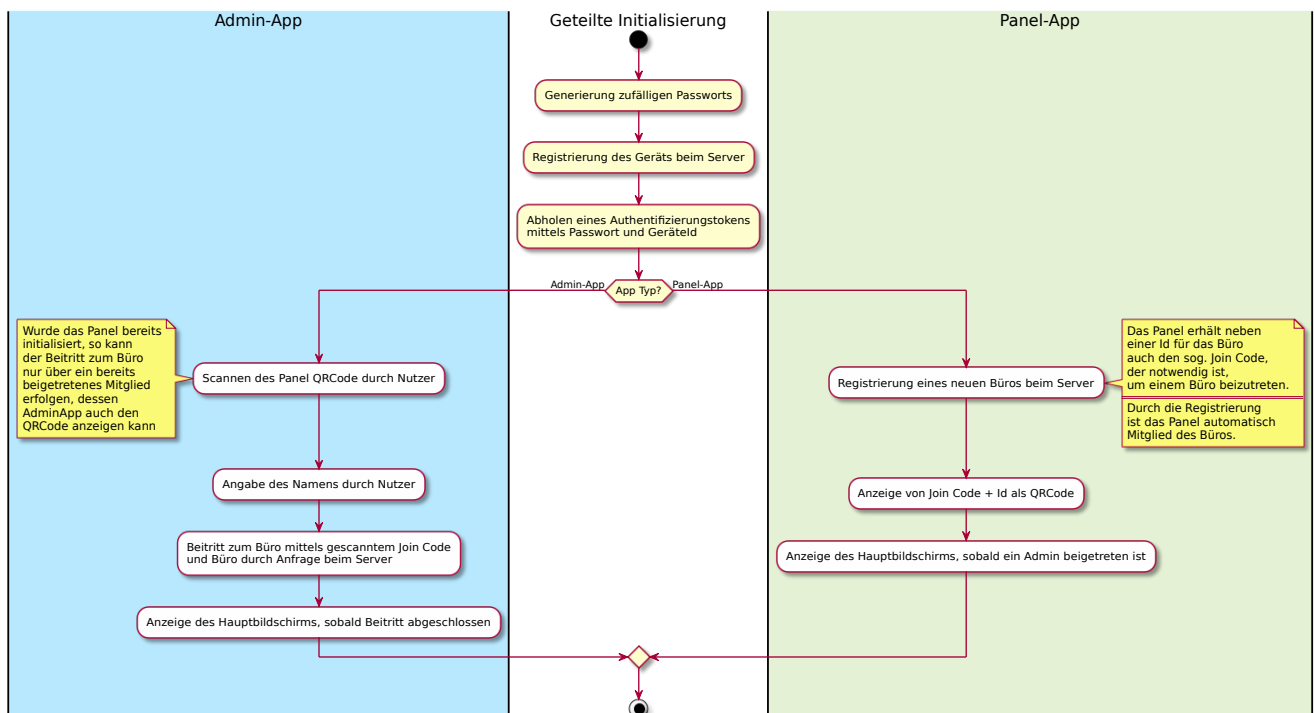


Abbildung 3.3.: Grobe Übersicht des Initialisierungsvorgangs der Apps

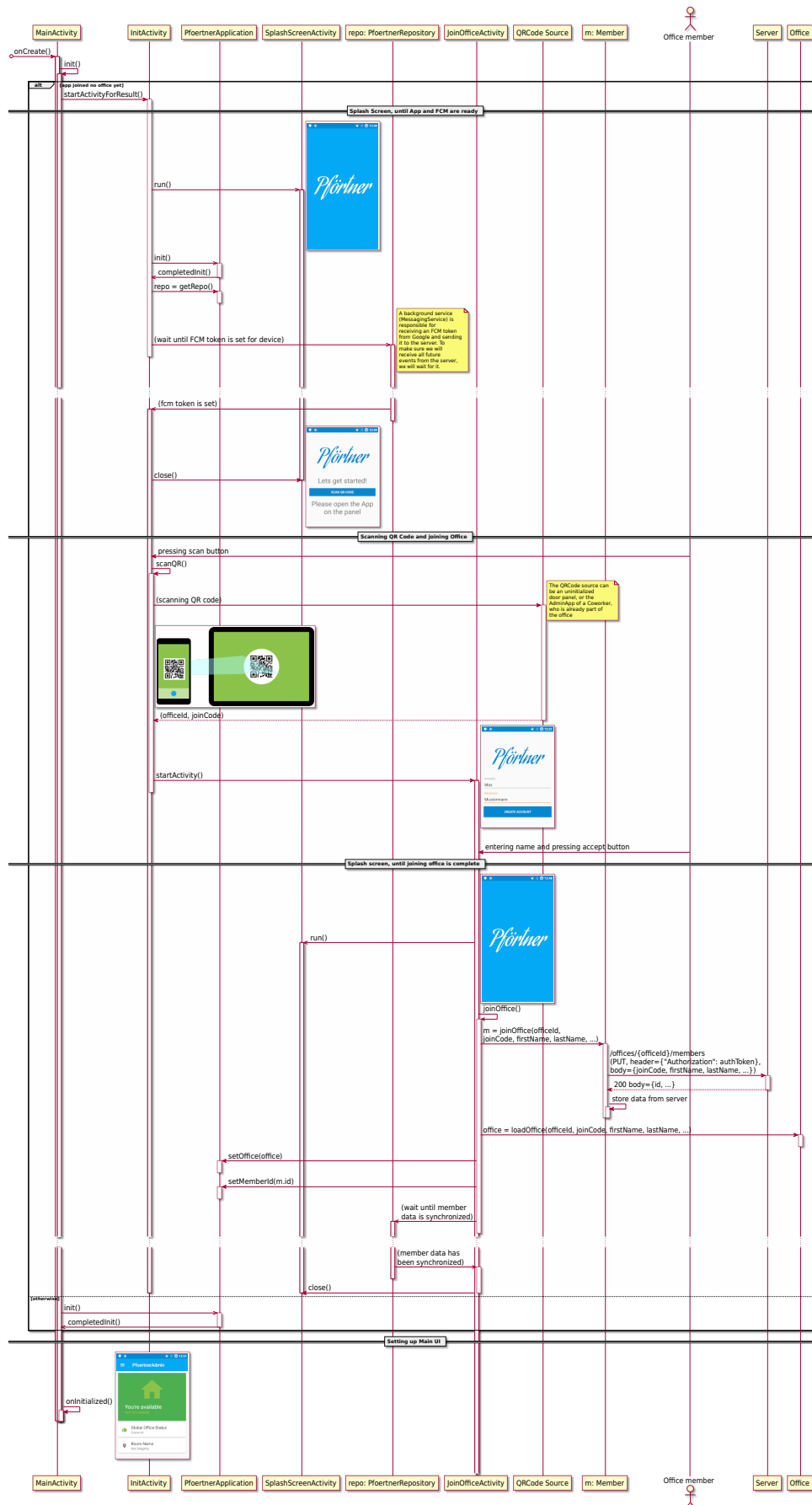


Abbildung 3.4.: Sequenzdiagramm der Implementierung des Initialisierungsvorgangs auf der Admin-App, einige Details wurden ausgelassen oder vereinfacht.

3.2 Synchronisierung

3.2.1 Motivation

Setzt ein Büromitglied den Status des Büros auf „Do not disturb“, so soll diese Information sowohl auf dem Türschild angezeigt werden, als auch auf den Admin-Apps der anderen Büromitglieder einsehbar sein.

Umgekehrt soll ein Büromitglied durch die Admin-App darüber informiert werden, wenn am Türschild durch Studierende Termine vereinbart wurden.

Es müssen also Informationen immerzu auf allen beteiligten Geräten (also Admin Apps und Türschild) verfügbar sein. Bei diesen Informationen handelt es sich unter anderem um die Namen der Mitarbeiter, ihren persönlichen Status, ihr Profilbild oder den Status des Büros. Die Gesamtheit dieser Informationen bezeichnen wir als den *Zustand* des Büros. Weiterhin sollen alle Geräte nach Austausch dazu notwendiger Nachrichten den gleichen Zustand einnehmen.

Wir bezeichnen diese Funktionalität im Folgenden als *Synchronisierung*.

3.2.2 Konzept

Um für alle Geräte letztendlich den gleichen Zustand herstellen zu können, stellt sich zunächst die Frage, welcher Zustand zu jedem Zeitpunkt durch die Geräte eingenommen werden soll. Wir bezeichnen diesen Zustand als den *korrekten* Zustand.

Wir haben uns dazu entschieden, den Zustand des Servers als den korrekten Zustand anzusehen. Diese Wahl wurde getroffen, da sie nicht die Implementierung von Protokollen und Datenstrukturen erfordert, um einen korrekten Zustand zwischen den Geräten regelmäßig auszuhandeln.

Eine Anpassung des korrekten Zustands erfolgt also, indem ein Gerät beim Server eine Änderung an dessen Zustand anfragt (*Request*). Wurde diese vom Server akzeptiert, so informiert er alle betroffenen Geräte über die Änderung. Wir bezeichnen diese Informationen als *Events*. Die Geräte laden dann den Teil des neuen Zustands vom Server herunter, welcher sich verändert hat (*Download*). Welcher Teil des Zustands sich verändert hat, erfahren die Geräte durch den Inhalt des Events.

Unter der Bedingung verlustfreier Kommunikation (Events gehen nicht verloren, Download des Zustands scheitert nicht) garantiert dieses Vorgehen, dass alle Geräte den gleichen Zustand wie der Server, also den korrekten Zustand abbilden, sobald Request, Events und Downloads erfolgt sind.

3.2.3 Eingesetzte Technik

Die Datenbank eines Geräts bildet das Datenbankschema des Servers ab (siehe auch 2.2.2). Hierbei werden allerdings Daten ausgelassen, welche das Gerät in der aktuellen Umsetzung in keinem Fall benötigt, z. B. die Büro- und Nutzerinformationen eines anderen Büros, dem es nicht angehört.

Eine Synchronisierung erfolgt folgendermaßen:

1. Ein Gerät führt über die HTTPS API des Servers 2.2.3 einen Request durch.
2. Der Server vermerkt die Änderung in seiner Datenbank.
3. Der Server bestimmt anhand der Art der Änderung, welche Geräte durch diese betroffen sind. Beispielsweise muss bei der Änderung des Status eines Büros jedes Gerät informiert werden, das Teil dieses Büros ist, aber keine Geräte, die Teil anderer Büros sind.
4. Der Server versendet Events mittels FCM an die identifizierten Geräte.
5. Nach Empfang eines Events lädt das Gerät den veränderten Teil des Zustands mittels der HTTP API herunter

-
6. Das Gerät pflegt die Zustandskomponente in seine lokale Datenbank ein.
 7. Sobald die Änderung in der lokalen Datenbank erfolgt ist, werden betroffene UI Elemente mittels reaktiver Programmiermethoden (siehe 3.2.4) aktualisiert.

3.2.4 Reaktive Programmierung

Der Zustand soll nach jeder Aktualisierung in der UI der Apps sichtbar werden. Wie in den Schritten 1 und 5 aus Sektion 3.2.3 erläutert, ist es weiterhin oft notwendig, Netzwerkzugriffe innerhalb der Apps zu tätigen. Da Verzögerungen während deren Ausführung auftreten können, müssen sie auf einem separaten Thread nebenläufig ausgeführt werden. Andernfalls wären keine Interaktionen mit der Nutzeroberfläche während eines Netzwerkzugriffs möglich.

Es zeichnet sich also folgendes Muster ab: Es existieren Datenquellen, welche Daten zu einem oder mehreren unbekannten Zeitpunkten bereitstellen und es existieren Datensinken, welche von diesen abhängen.

Reaktive Programmierung bezeichnet ein Programmierparadigma, welches es unter anderem erlaubt, Abhängigkeiten zwischen zeitveränderlichen Werten und Berechnungen, welche diese verwenden, zu formulieren. Änderungen der Werte werden hierbei automatisch durch das verwendete, reaktive Framework an die Berechnungen weitergeleitet und diese ausgeführt [5].

Bei Datenbankaktualisierungen und Netzwerkanfragen dedizierte Logik für z. B. die Behandlung jedes UI Elements zu schreiben führt zu einer starken Kopplung zwischen den UI Komponenten und der Datenbanklogik, also zu unübersichtlichem, schwer wartbarem Code. Wir haben uns daher für den Einsatz reaktiver Programmierung entschieden, da es uns erlaubt, unabhängige Module für Datenquellen und UI zu entwickeln. Weiterhin bietet das Android Open Source Project [2] im Rahmen der Android Architecture Components [1] mit den *Room* [31] und *LiveData* [30] Bibliotheken bereits eine Implementierung des Paradigmas für SQLite Datenbanken als Quellen und Android UI als Senken.

Auch beachtet die *LiveData* Bibliothek automatisiert den Android Lifecycle [29], d.h. nachdem eine Activity-Instanz beispielsweise zerstört wurde, werden die *LiveData* Objekte, welche die UI der Activity aktuell halten, automatisch inaktiv, andernfalls kann es beispielsweise zu Memory-Leaks kommen, da noch Referenzen auf nicht mehr verwendete Objekte existieren.

Für Abläufe, welche Transformationen der Daten aus der Quelle und z. B. weitere Netzwerkzugriffe als Reaktion auf eine Datenänderung erfordern, setzen wir weiterhin die *RxJava* [10] Bibliothek ein, welche die Formulierung besonders komplexer, reaktiver Vorgänge erleichtert und auch kompatibel mit *Room* ist. Ein Beispiel für einen solchen Vorgang wäre die Einlösung eines Google OAuth2 Authorization Codes, sobald dieser verfügbar ist (siehe 3.3).

3.3 Google Kalender Einbindung

3.3.1 OAuth2 Authentifizierung bei Google

Um die Kalenderdaten unserer Nutzer abzufragen, nutzen wir die Google Calendar API [12], deren Verwendung mithilfe von OAuth 2.0 [20] autorisiert wird. OAuth 2.0 ist ein Protokoll, das es Dritten (auch als *Client* bezeichnet) erlaubt, im Namen eines autorisierten Nutzers mit einem Webserver zu interagieren und auf Daten zuzugreifen. In unserem Fall möchte die Panel-App im Namen eines Büromitglieds mit einem Google Server kommunizieren.

Die Panel-App ist hier der Client. Sie wurde in der API Console [4] der Google Cloud Platform [15] angemeldet. Anfragen auf Grants ¹ enthalten eine Client Id und ein Client Secret, die den Client als Pförtner, wie in der API Console angemeldet, identifizieren. Client Id und Client Secret lassen sich innerhalb der API Console erhalten und wurden in der Panel App hinterlegt.

¹ Methoden, Zugriff auf einen bestimmten API Endpunkt zu erhalten

Als Erstes muss der Nutzer dem Zugriff zustimmen. Dies erfolgt in der Admin App über den Aufruf eines User Interface, welches durch die Google Sign-In Library bereitgestellt wird. Nach der Zustimmung erhält die Admin App Daten von Google, die wir an unseren Server schicken. Dazu definiert OAuth 2.0 verschiedene Arten von Grants (Grant Types) für verschiedene Anwendungsfälle:

- **Authorization Code** - Der Client erhält einen Code, den er gegen ein *Authorization Token* für den Zugriff auf die Daten des Nutzers einlösen kann. Es kann auch für ein **Refresh Token** und weitere tokens eingelöst werden.
- **Implicit** - Der Client erhält direkt ein Authorization Token mit dem er Zugriffe durchführen kann.
- **Password** - Ein Passwort wird direkt gegen ein Authorization Token eingelöst. Dieser Typ ist für Drittparteien ungeeignet, da er das Speichern des Passworts erfordert.
- **Refresh Token** - Der Client löst ein Refresh Token gegen ein Authorization Token ein.

Wir nutzen den Grant Type Authorization Code, da wir so keine sensiblen Daten über unsere eigenen Verbindungen schicken müssen. Der Authorization Code ist nur einmal gültig und kann so ohne Bedenken versandt werden. Wenn in diesem Schritt der Header `access_type=offline` gesetzt ist, kann der erhaltene Authorization Code gegen ein Refresh Token eingelöst werden. Der Login Screen informiert den Nutzer dann zusätzlich, dass Offline Zugriff angefordert wurde. Refresh Tokens haben kein Ablaufdatum und können immer wieder gegen neue Authorization Tokens eingelöst werden.

Wir verwenden daher Refresh Tokens, da das Türschild über einen längeren Zeitraum genutzt wird und wir nicht Nutzer häufig um Zugriffserlaubnis bitten müssen.

3.3.2 Push Notifications

Wenn sich im Kalender eines Nutzers etwas ändert, soll die Panel-App ein Event (siehe 3.2) erhalten und gegebenenfalls ihre UI aktualisieren. Push Notifications können für den Google Kalender [14] über sogenannte *Webhooks* umgesetzt werden.

Ein Webhook ist eine URL, die von Google aufgerufen werden kann, um einen Server über Änderungen an Daten zu informieren. Die mit OAuth 2.0 authentifizierte Application muss zuerst beweisen, dass sie die URL besitzt. Dazu mussten wir in der Google Search Console [17] eine HTML Seite herunterladen und diese auf einer versteckten URL unseres Servers hosten.

Die Panel-App fordert Notifications durch Anfragen im folgenden Format an:

```
POST https://www.googleapis.com/calendar/v3/calendars/<Kalender Id>/events/watch
Authorization: Bearer <Authentication Token>
Content-Type: application/json
{
  "id": "<Hier: Zufällige UUID>", // Channel Id.
  "type": "web_hook",
  "address": "https://<domain>:3000/notifications", // Empfängerurl
}
```


Wir schicken die angefragte Channel Id an den Server und speichern sie in der Server Datenbank im OfficeMember Modell ab (siehe 2.2.2). Sobald sich im spezifizierten Kalender etwas ändert, spricht Google unseren Server an. Die Anfragen haben folgendes Format:

```
POST https://<domain>:3000/notifications
X-Goog-Channel-ID: <Channel Id>
```

X-Goog-Channel-Expiration: <String Ablaufdatum des Channels>
X-Goog-Resource-ID: <Kalender Id>
X-Goog-Resource-URI: <Kalender Url>
X-Goog-Resource-State: sync/ exists/ not_exists
X-Goog-Message-Number: <Inkrementierte Notification Id>

Der Server identifiziert anhand der Channel Id die Panel-App, an der der OfficeMember registriert ist und schickt ein Event mit dem Inhalt:

```
{  
  event: "CalendarUpdated",  
  payload: "<Office Member Id>"  
}
```



A Anhang

Verwendete Softwarebibliotheken

- [9] Node.js Contributors. *Node.js JavaScript runtime*. <https://nodejs.org/en/>. Zugriffen: 13.03.2019, Lizenz: MIT. 2009–2019.
- [10] RxJava Contributors. *RxJava library*. <https://github.com/ReactiveX/RxJava>. Zugriffen: 12.03.2019, Lizenz: Apache 2.0. 2016–2019.
- [18] Hipp, Wyrick & Company, Inc. *SQLite*. <https://sqlite.org/download.html>. Zugriffen: 12.03.2019, Lizenz: Public Domain. 2000–2019.
- [19] TJ Holowaychuk u. a. *Express web framework*. <https://expressjs.com/>. Zugriffen: 13.03.2019, Lizenz: MIT Lizenz. 2009–2019.
- [24] Google LLC. *Firebase Cloud Messaging*. <https://firebase.google.com/docs/cloud-messaging/>. Zugriffen: 13.03.2019, Lizenz: Android Software Development Kit License. 2014–2019.
- [30] The Android Open Source Project. *LiveData Library*. <https://developer.android.com/topic/libraries/architecture/livedata>. Version 1.1.1. Zugriffen: 12.03.2019, Lizenz: Apache 2.0. unknown–2019.
- [31] The Android Open Source Project. *Room Persistence Library*. <https://developer.android.com/topic/libraries/architecture/room>. Version 1.1.1. Zugriffen: 12.03.2019, Lizenz: Apache 2.0. 2018–2019.
- [32] Sequelize Contributors. *Sequelize Object Relational Mapping*. <http://docs.sequelizejs.com/>. Zugriffen: 12.03.2019, Lizenz: MIT. 2014–2019.
- [33] Retrofit Contributors Square, Inc. *Retrofit HTTP client*. <http://square.github.io/retrofit/>. Zugriffen: 13.03.2019, Lizenz: Apache 2.0 Lizenz. 2013–2019.

Verweise auf Webauftritte

- [1] *Android Architecture Components*. Zugriffen: 18.03.2019. URL: <https://developer.android.com/topic/libraries/architecture/>.
- [2] *Android Open Source Project*. Zugriffen: 18.03.2019. URL: <https://source.android.com/>.
- [3] *Android Studio Entwicklungsumgebung*. Zugriffen: 18.03.2019. URL: <https://developer.android.com/studio/>.
- [4] *API Console der Google Cloud Platform*. Zugriffen: 18.03.2019. URL: <https://console.cloud.google.com/apis>.
- [7] Netflix Technology Blog. *Node.js in Flames*. Zugriffen: 13.03.2019. 2014. URL: <https://medium.com/netflix-techblog/node-js-in-flames-ddd073803aa4>.
- [8] Paul Brown. *State of the Union: npm*. Zugriffen: 13.03.2019. 2017. URL: <https://www.linux.com/news/event/Nodejs/2016/state-union-npm>.
- [11] *Express at GitHub*. Zugriffen: 13.03.2019. URL: <http://paypal.github.io/>.
- [12] *Google Calendar API*. Zugriffen: 18.03.2019. URL: <https://developers.google.com/calendar/>.
- [13] *Google Calendar Provider*. Zugriffen: 18.03.2019. URL: <https://developer.android.com/guide/topics/providers/calendar-provider.html>.
- [14] *Google Calendar Push Notifications*. Zugriffen: 18.03.2019. URL: <https://developers.google.com/calendar/v3/push>.
- [15] *Google Cloud Platform*. Zugriffen: 18.03.2019. URL: <https://cloud.google.com/>.
- [16] *Google Play Services*. Zugriffen: 13.03.2019. URL: <https://play.google.com/store/apps/details?id=com.google.android.gms>.
- [17] *Google Search Console*. Zugriffen: 18.03.2019. URL: <https://search.google.com/search-console>.
- [20] *Informationen zu OAuth 2.0*. Zugriffen: 18.03.2019. URL: <https://oauth.net/2/>.
- [23] *Let's Encrypt Zertifizierungsstelle*. Zugriffen: 13.03.2019. URL: <https://letsencrypt.org/>.
- [25] Icons8 LLC. *Flat Color Icons*. Zugriffen: 13.03.2019, Lizenz: MIT. URL: <https://github.com/icons8/flat-color-icons>.
- [26] *MySQL Webauftritt*. Zugriffen: 13.03.2019. URL: <https://www.mysql.com/>.
- [27] *Node Package Manager*. Zugriffen: 15.03.2019. URL: <https://www.npmjs.com/>.
- [28] PayPal. *Open Source at PayPal*. Zugriffen: 13.03.2019. 2019. URL: <http://paypal.github.io/>.
- [29] Android Open Source Project. *Understanding the Activity Lifecycle*. Zugriffen: 14.03.2019. URL: <https://developer.android.com/guide/components/activities/activity-lifecycle>.
- [34] Alexey Varfolomeev, Sergei Eremenko und Papirus Contributors. *Papirus Icon Theme*. Zugriffen: 13.03.2019, Lizenz: GPL. URL: <https://git.io/papirus-icon-theme>.

Literatur

- [5] Engineer Bainomugisha u. a. „A survey on reactive programming“. In: *ACM Computing Surveys (CSUR)* 45.4 (2013), S. 52.
- [6] Kris M Bell, Darryl N Bleau und Jeffrey T Davey. *Push notification service*. US Patent 8,064,896. Nov. 2011.
- [21] *Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification*. Standard. Zugegriffen: 12.03.2019. International Organization for Standardization, Feb. 2015. URL: <https://www.iso.org/standard/62021.html>.
- [22] Erik Kay u. a. *Push notifications for web applications and browser extensions*. US Patent 8,739,249. Mai 2014.