

# Raspberry PI Security (RPISec)

Thomas Herzog, Philipp Wurm

29. Juni 2017

# 1 Einleitung

Diese Dokument stellt die Dokumentation des Projekts *Raspberry PI Security*, in weiterer Folge *RPISec* genannt, dar, welches für die Lehrveranstaltung *Mobile und ubiquitäre Systeme* realisiert wurde. In diesem Projekt wird eine Heimsicherheitsanwendung mit einem *Raspberry PI 3 Model B*, *Docker* und *Spring Boot Microservices* umgesetzt, das bei einem Sicherheitsverstoß in der Lage sein soll, bekannte mobile Endgeräte von registrierten Benutzern über diesen Sicherheitsverstoß zu benachrichtigen.

## 1.1 Problemdarstellung

Dieser Abschnitt behandelt die Problemdarstellung, welche die Grundlage für das umzusetzende Projekt *RPISec* ist. Bei einem Auslösen eines an dem *Raspberry PI* angeschlossenen Bewegungssensors, sollen alle am System registrierten Benutzer über ihre mobilen Endgeräte wie Handy oder Tablet über den Vorfall benachrichtigt werden, sowie die Möglichkeit haben ein Bild zu erhalten, welches den Sicherheitsbereich zum Zeitpunkt des Sicherheitsverstoßes zeigt.

Da es sich um eine Sicherheitsanwendung handelt, soll die Benutzerverwaltung sowie die Authentifizierung *In-House* gehalten werden, also die Sicherheitsanwendung selbst soll in der Lage sein die Benutzer zu verwalten und die Authentifizierung der Benutzer durchzuführen. Da sich die mobilen Endgeräte in irgendwelchen Netzen an das Internet anbinden können, wie zum Beispiel über einen Mobilfunkanbieter, Internetanbieter oder öffentlichen *Hot-Spot*, wird ein *Messaging* Dienst benötigt, über den die mobilen Endgeräte erreicht werden können. Dieser *Messaging* Dienst muss es erlauben, dass die Benutzerverwaltung extern erfolgen kann, da wir diesen *Messaging* Dienst nicht vertrauen wollen und daher den *Messaging* Dienst auch nicht die Benutzerverwaltung überlassen wollen. Ebenso wird ein online Speichermedium benötigt, dass alle gemachten Bilder speichert, damit die registrierten Benutzer jederzeit darauf zugreifen können.

## 1.2 Funktionsweise

Dieser Abschnitt behandelt die Funktionsweise von *RPISec*. Die Abbildung 1 zeigt den Systemaufbau von *RPISec* mit der involvierten Hardware und den involvierten *Cloud* Dienst.

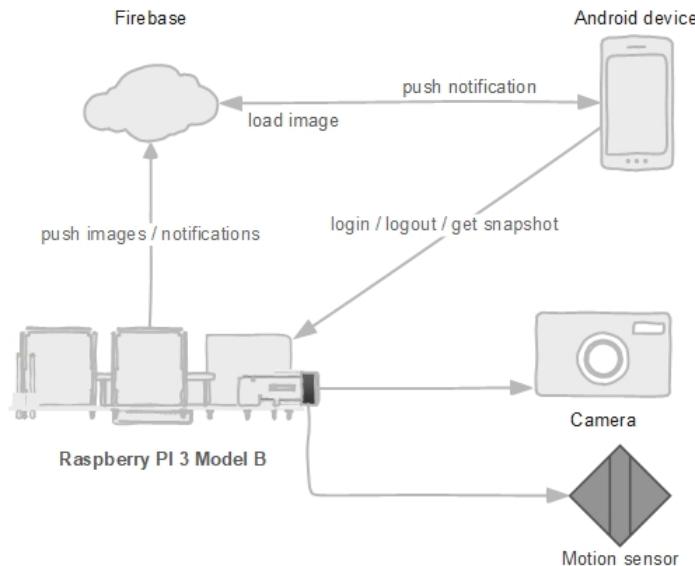


Abbildung 1: Systemaufbau der *RPISec* Applikation

### 1.2.1 Zugangsverifikation

Beim Start des Systems wird vom Authentifizierungsservice (*Auth-Service*) ein Administrator Benutzer erstellt, wenn dieser nicht bereits existiert. Ein neu angelegter Benutzer wird über eine E-Mail dazu aufgefordert, seinen Zugang zu aktivieren, in dem er ein Password vergibt. Im Idealfall würde die Zugangsverifikation nur im Heimnetz möglich sein.

A screenshot of a web-based password confirmation form. It features two input fields: one for 'Password' and one for 'Confirmation'. Below the fields is a blue 'Submit' button. The entire form is set against a dark background.

Abbildung 2: Zugangsaktivierung

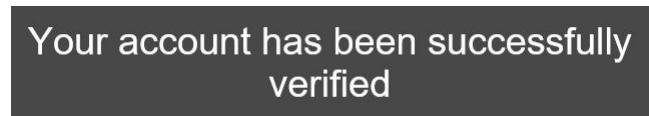


Abbildung 3: Bestätigung der Aktivierung

### 1.2.2 Client Login

Die Abbildung 4 zeigt das Sequenzdiagramm, welches den Ablauf des Logins eines registrierten Benutzers über einen mobilen *Client* beschreibt.

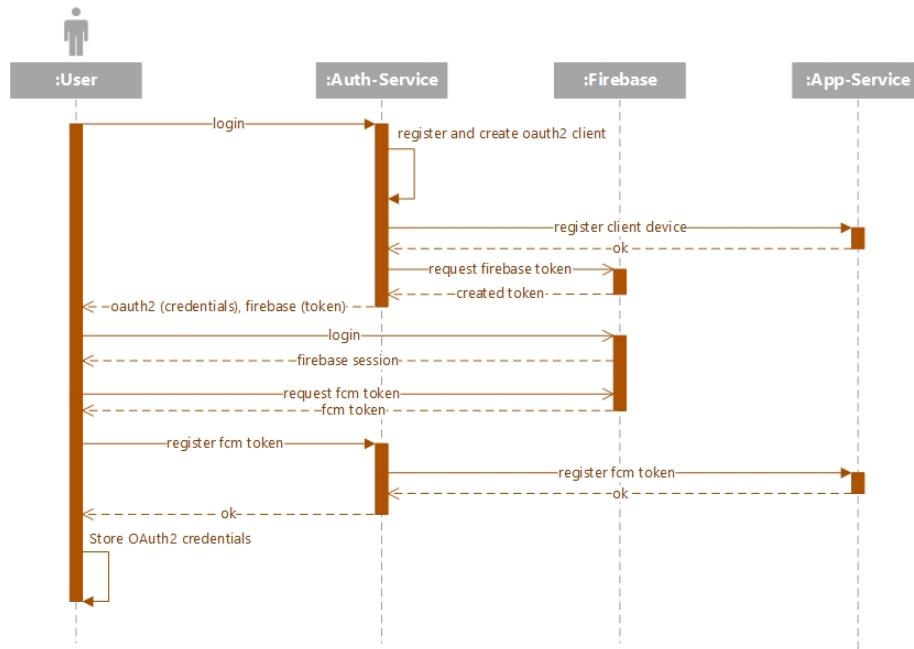


Abbildung 4: Sequenzdiagramm des Benutzerlogins

Im Zuge des Benutzerlogins wird der mobile *Client* am (*Auth-Service*), der für die Authentifizierung und die Benutzerverwaltung verantwortlich ist und am Applikationsservice (*App-Service*), der mit der Hardware und dem *Messaging* Dienst interagiert, registriert. Der *Auth-Service* erstellt bei jedem Login einen neuen *OAuth2-Client* und löscht gegebenenfalls einen bereits bestehenden *OAuth2-Client* für den aktuellen mobilen *Client*. Das Erstellen eines *OAuth2-Clients* für jeden mobilen *Client* wird durchgeführt, da mit der *Client-App* keine *Oauth2-Client* Zugangsdaten an die mobilen *Clients* mit ausgeliefert werden sollen.

Dem mobilen *Client* wird bei einem Login ein Authentifizierungstoken für den *Cloud* Dienst übermittelt, mit dem sich der mobile *Client* am *Cloud* Dienst anmelden kann. Nachdem Login eines mobilen *Clients* am *Cloud* Dienst holt sich der mobile *Client* seine eindeutige Id vom *Cloud* Dienst in Form eines Tokens, der am *Auth-Service* registriert wird, der wiederum den Token am *App-Service* registriert, damit dieser in der Lage ist, Nachrichten an die registrierten mobilen *Clients* zu versenden.

### 1.2.3 Sicherheitsverstoß melden

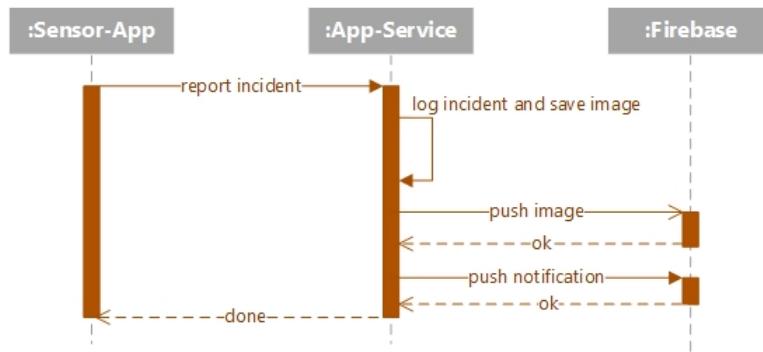


Abbildung 5: Sequenzdiagramm für das Behandeln eines Sicherheitsverstoßes

Die Abbildung 5 zeigt das Sequenzdiagramm für das Behandeln eines Sicherheitsverstoßes, der von der Sensorapplikation erkannt und dem Applikationsservice mitgeteilt wird. Der Sicherheitsverstoß wird über den *Cloud* Dienst an die mobilen *Clients* gemeldet, wobei einerseits eine Nachricht an die mobilen *Clients* versendet wird, sowie das gemachte Bild in einer Onlinedatenbank den mobilen *Clients* zum Download zur Verfügung gestellt wird. Die Benutzer können jederzeit auf die Onlinedatenbank zugreifen und sich die Bilder auf ihren jeweiligen mobilen *Clients* herunterladen.

Der Ansatz einen *Cloud* Dienst zu verwenden sorgt dafür, dass das System entlastet wird, da der Datenfluss und die Netzwerkzugriffe vom System ins Internet sowie umgekehrt minimiert werden. Die Daten müssen nur einmalig in die *Cloud* hochgeladen werden und die Benutzer können jederzeit, beliebig oft und von jedem beliebigen mobilen *Client* darauf zugreifen.

#### 1.2.4 Nachrichtenempfang am mobilen *Client*

Die Abbildung 6 zeigt den Ablauf einer Benachrichtigung eines mobilen *Clients* über den *Messaging* Dienst.

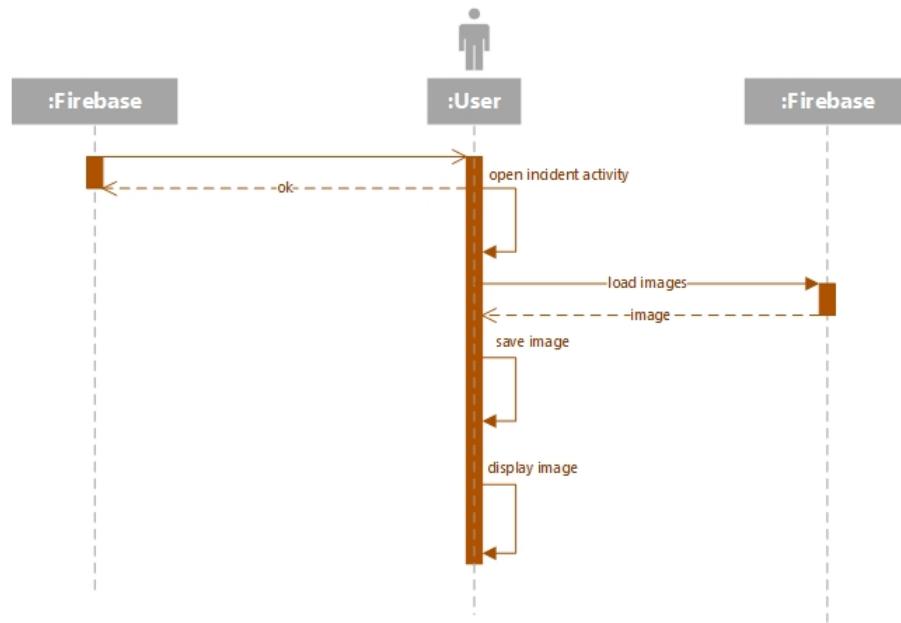


Abbildung 6: Sequenzdiagramm der Benachrichtigung eines mobilen *Clients*

Nachdem das System die mobilen *Clients* via dem *Messaging* Dienst über einen Sicherheitsverstoß benachrichtigt hat, erhalten die mobilen *Client*-Anwendungen die Nachricht von dem *Messaging* Dienst und zeigen diese an. Nachdem die Benutzer auf die Nachricht geklickt haben, wird eine *Activity* für das Anzeigen der Bilder geöffnet, die alle bereits gespeicherten Bilder und das neu geladene Bild anzeigt. Diese Daten werden von der Onlinedatenbank bereitgestellt.

## 2 *Raspberry PI*

Dieser Abschnitt behandelt den Aufbau von *RPISec* und die verwendete Hardware. Für den Testaufbau wurden folgende Hardwarekomponenten verwendet.

- Ein *Raspberry PI 3 Model B*<sup>1</sup>,
- *AZDeliveryCamRasp*<sup>2</sup> und ein
- *HC-SR501*<sup>3</sup> Bewegungssensor.

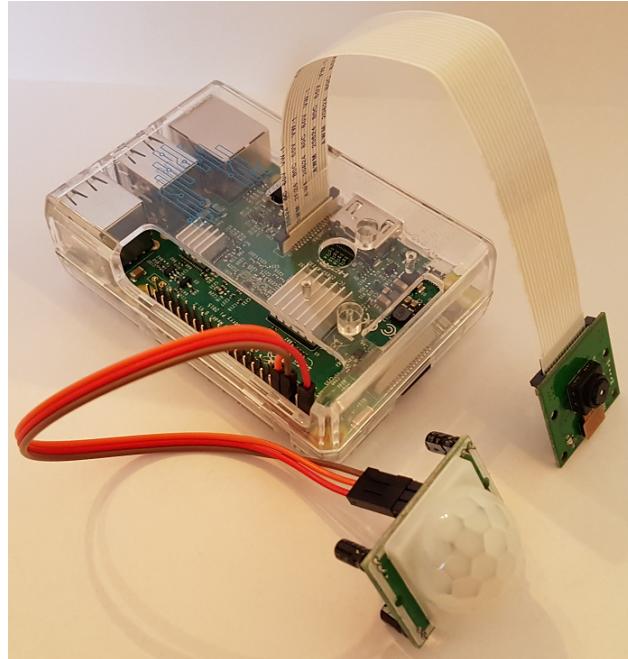


Abbildung 7: Testaufbau der Applikation

Die Abbildung 7 zeigt den verwendeten Testaufbau. Die Kamera wurde über CSI (*Camera-Serial-Interface*) und der Bewegungssensor über GPIO (*General Purpose Input/Output*) an den *Raspberry PI* angeschlossen.

### 2.1 Kamera

Bei der Kamera handelt es sich um ein eigens für den *Raspberry Pi* entwickeltes Modell und wird direkt an den vorhanden Kamera-Port (CSI) des *Raspberry Pi 3*<sup>4</sup> angeschlossen. Anschließend muss die Kamera aktiviert werden. Dies geschieht über das, mit den meisten Linux-Distributionen mitgelieferte, Programm *raspi-config*<sup>5</sup>. Dieses Programm wird auch benutzt, um andere Komponenten des *Raspberry Pi* zu konfigurieren.

<sup>1</sup><https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>

<sup>2</sup><https://az-delivery.de/products/raspberrykamerav1-3>

<sup>3</sup><https://www.mpja.com/download/31227sc.pdf>

<sup>4</sup>[https://en.wikipedia.org/wiki/Raspberry\\_Pi#/media/File:Raspberry-Pi-3-Flat-Top.jpg](https://en.wikipedia.org/wiki/Raspberry_Pi#/media/File:Raspberry-Pi-3-Flat-Top.jpg)

<sup>5</sup><https://upload.wikimedia.org/wikipedia/commons/e/ed/Raspi-config.png>

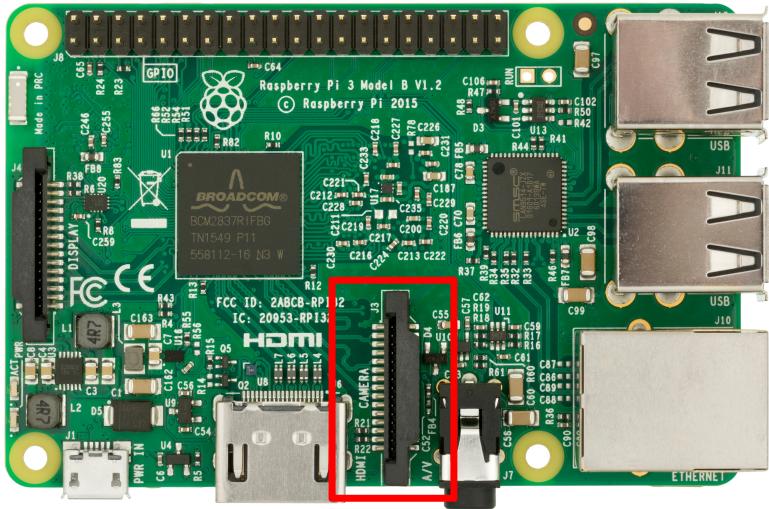


Abbildung 8: Raspberry-Pi 3

Die Abbildung 9 zeigt die Oberfläche, über welche die Kamera aktiviert werden kann.

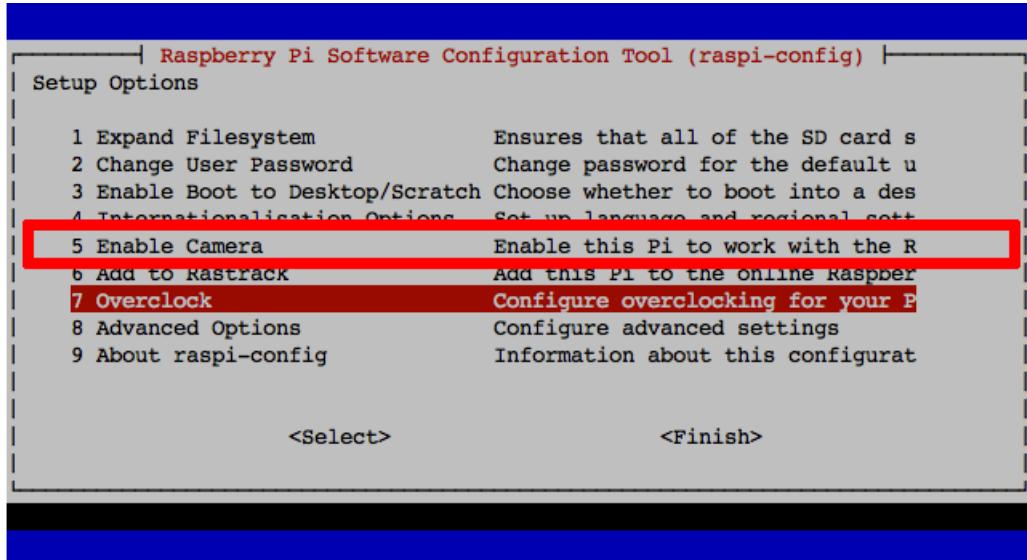


Abbildung 9: Raspi-config

Zur Ansteuerung der Kamera wird das Konsolenprogramm *raspistill* verwendet. Diese kann mit verschiedenen Parametern konfiguriert werden. Als Beispiel:

```
raspistill --width 1920 --height 1080 -o test.jpg
```

Dies erzeugt ein Bild in der Auflösung 1920 x 1080 Pixel und speichert es unter dem Dateinamen *test.jpg*. *RPIsec* wird in *Docker Container*n gehostet und die Anwendung *raspistill* wird in ein *Docker Image* installiert und muss daher nicht am Host installiert werden.

## 2.2 HC-SR501 Bewegungssensor

Der HC-SR501 Sensor wird über die *GPIO-Pins* des *Raspberry Pi* angesteuert. Hierbei werden Masse, Stromversorgung und Daten-Pin des HC-SR501 Sensors mit den *GPIO-Pins* verbunden.

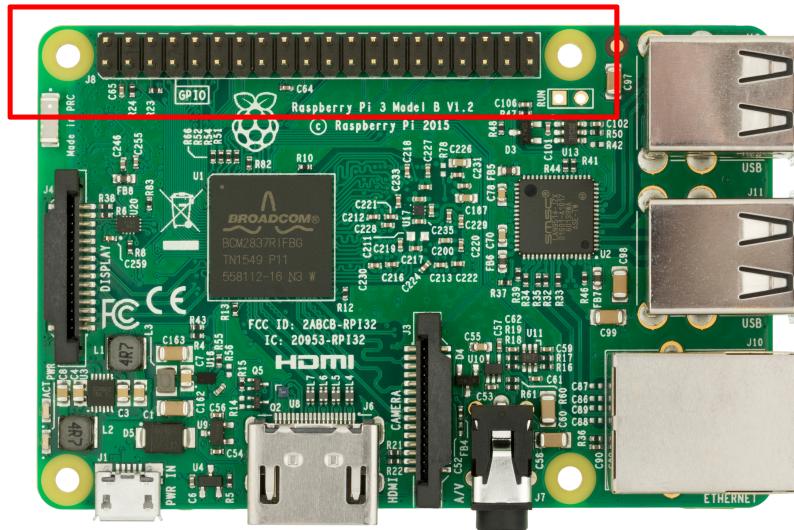


Abbildung 10: Raspberry-Pi 3

Die Pin-Belegung<sup>6</sup> des HC-SR501

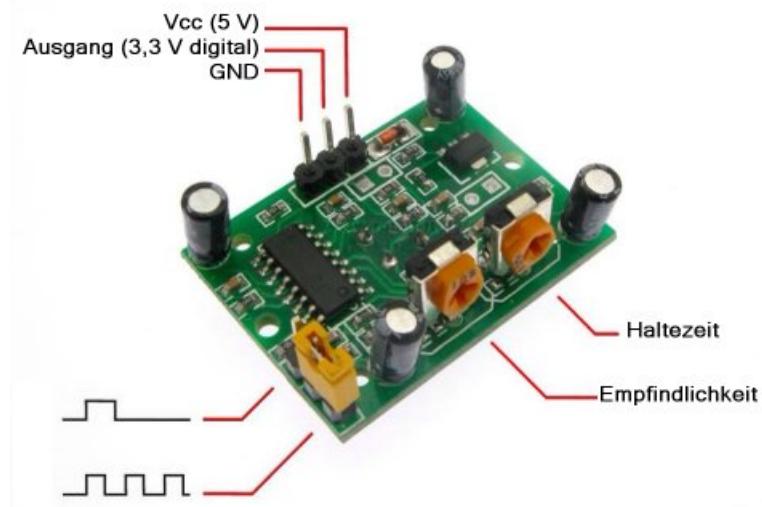


Abbildung 11: Pin-Schema HC-SR501

ist wie folgt definiert:

- Pin 1: VCC (5 Volt)
- Pin 2: Out, Data
- Pin 3: GND, Masse

<sup>6</sup><http://www.netzmafia.de/skripten/hardware/RasPi/Projekt-PIR/>

Zusätzlich kann die Empfindlichkeit und Haltezeit an den beiden Drehreglern eingestellt werden. Zusätzlich kann noch das Daten-Pin Verhalten über den Jumper konfiguriert werden.

## 2.3 GPIO

*GPIO* ist die Abkürzung für *General Purpose Input/Output*. Man bezeichnet damit programmierbare Ein- und Ausgänge für allgemeine Zwecke. Die *GPIOs* werden als Lötpunkt oder Pin in Form einer Stifteleiste herausgeführt und dienen als Schnittstelle zu anderen Systemen oder Schaltungen, um diese über den *Raspberry Pi* zu steuern. Dabei kann der *Raspberry Pi* bei entsprechender Programmierung digitale Signale von außen annehmen (*Input*) oder Signale nach außen abgeben (*Output*).

Viele der *GPIOs* erfüllen je nach Einstellung und Programmierung verschiedene Funktionen. Neben den typischen *GPIO*-Ein- und Ausgängen finden sich aber auch Pins mit der Doppelfunktion für *I<sub>2</sub>C*, *SPI* und eine serielle Schnittstelle.

### 2.3.1 HC-SR501 GPIO Verbindung

Für den HC-SR501 wird die Pin-Belegung wie folgt gewählt:

- Pin 1: VCC (5 Volt) an Pin 2
- Pin 2: Out, Data an Pin 8
- Pin 3: GND an Pin 6

Diese Pin-Belegung erfolgt aufgrund der folgenden schematischen Darstellung<sup>7</sup>:

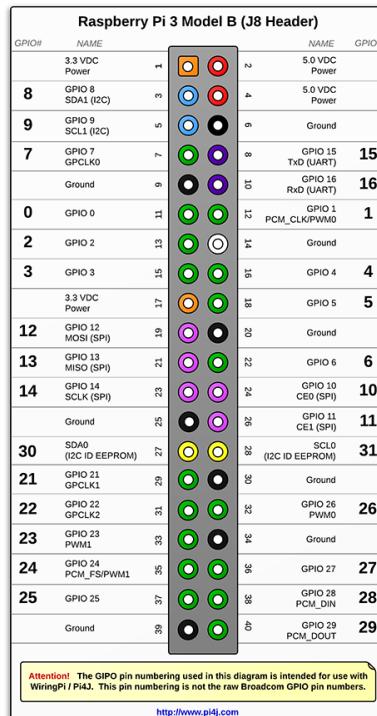


Abbildung 12: Pin-Schema

<sup>7</sup><http://pi4j.com/pins/model-3b-rev1.html>

## 2.4 Betriebssysteme

Dieser Abschnitt behandelt die verwendete Betriebssysteme für den *Raspberry PI*. Die Applikation *RPISec* wurde einerseits mit dem Betriebssystem *HypriotOS* und andererseits mit *Raspian* realisiert. Das Betriebssystem *HypriotOS* basiert auf *Debian Jessie* und wird von dem *OpenSource* Projekt *Hypriot*<sup>8</sup> zur Verfügung gestellt. Das Ziel von *HypriotOS* ist es ein Betriebssystem für den *Raspberry PI* zur Verfügung stellen, das bereits Docker vorinstalliert und betriebsbereit hat. Mit dem Betriebssystem *Raspian* muss Docker selbst installiert werden, wobei Docker als Paket im *Repository* zur Verfügung steht und daher sich die Installation als unkompliziert gestaltet.

Wenn Docker installiert und betriebsbereit ist, dann spielt es keine Rolle auf welchem Betriebssystem die Applikation *RPISec* betrieben wird, solange dieses Betriebssystem auf einer von *Docker* unterstützten Kernelversion aufbaut.

Da die Applikation *RPISec* auf eine aktive Internetverbindung angewiesen ist, muss das Betriebssystem so konfiguriert werden, dass der *Raspberry PI* entweder über *Ethernet* oder *Wlan* an ein Netzwerk angebunden ist, das Zugriff auf das Internet erlaubt. In einem produktiven Betrieb muss der *Raspberry PI* über das Internet erreichbar sein, damit die mobilen *Clients* Anfragen an die gehosteten *Microservice* absetzen können.



Abbildung 13: HypriotOS

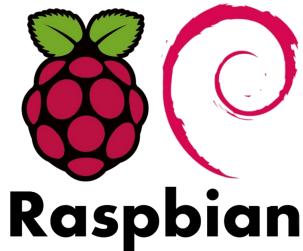


Abbildung 14: RaspianOS

---

<sup>8</sup><https://blog.hypriot.com/>

## 3 Software und *Cloud*

Dieser Abschnitt behandelt die verwendete bzw. implementierte Software und die verwendeten Dienste für die Applikation *RPISec*.

### 3.1 *Microservices*

Dieser Abschnitt behandelt die auf dem *Raspberry PI* gehosteten Services. Die Services wurden mit *Spring Boot* als *Microservices* implementiert, was möglich war, da Oracle eine ARM Implementierung der Java-JDK bereitstellt und die *Microservices* schlank implementiert wurden, sodass die zur Verfügung stehenden Ressourcen ausreichen, um diese Services auf einen *Raspberry PI* zu betreiben.

Es wurden die beiden *Microservices Auth-Service* für die Benutzerverwaltung und OAuth2 Authentifizierung und *App-Service* für die Interaktion mit der Sensorapplikation und der Interaktion mit dem *Cloud* Dienst implementiert, wobei der *Microservice Auth-service* im Zuge des Projekts für die Lehrveranstaltung *Service Engineering* implementiert wurde. Es hätte auch ausgereicht die Benutzerverwaltung in den *Microservice App-Service* zu verpacken, obwohl dann der *Microservice* für zwei Aspekte verantwortlich gewesen wäre, was im Widerspruch zu einem *Microservice* steht, der nur für einen Aspekt verantwortlich sein soll.

Der *Microservice App-Service* interagiert nicht direkt mit der Sensorik, sondern bindet die Sensorapplikation beschrieben in Abschnitt 3.2 ein und ist für dessen Lebenszyklus verantwortlich. Nachdem Start der Sensorapplikation wird ein *Listener* registriert, der auf Statusänderungen des Bewegungssensor reagiert und diesen Sicherheitsverstoß wie in Abbildung 5 behandelt.

### 3.2 Sensor Applikation

Bei den Komponenten der Sensor-Anwendung handelt es sich, wie in Abschnitt 2 erwähnt, um die beiden Bauteile:

- *AZDeliveryCamRasp Kamera*
- *HC-SR501 Bewegungssensor*

Die Ansteuerung der Pins erfolgt über die *Pi4J*<sup>9</sup>-Bibliotheken die wiederum die *WiringPi*<sup>10</sup>-Bibliothek nutzt, welche die eigentliche Ansteuerung der Pins erledigt.

#### 3.2.1 Pi4J

*Pi4J* ist eine Bibliothek für Java, die den vollen Zugriff auf die Ressourcen des *Raspberry PI* ermöglicht. Mit *Pi4J* ist es möglich Anwendungen für *Raspberry PI* zu schreiben, die nur Java benötigen. Damit können praktisch alle Bibliotheken eingesetzt werden, die für Java verfügbar sind. Einschränkungen gibt es nur bei den Ressourcen des *Raspberry PI*.

#### 3.2.2 WiringPi

*WiringPi* ist eine Bibliothek, um die *GPIO* Ein-und Ausgänge am *Raspberry Pi* zu schalten. Das Ziel dieser Bibliothek ist es, eine einzige gemeinsame Plattform und Programmierschnittstelle für den Zugriff auf die *GPIOs* des *Rapsberry Pi* für verschiedene Programmiersprachen zur Verfügung zu stellen. Im Kern ist *WiringPi* eine C-Bibliothek, aber sie steht auch in Ruby und Python zur Verfügung.

---

<sup>9</sup><http://pi4j.com/>

<sup>10</sup><http://wiringpi.com/>

Quelltext 1: IRSensor\_HCSR501.java

```

1  public class IRSensor_HCSR501 implements IRSensorDevice {
2      private GpioController gpioController;
3
4      // ...
5
6      @Override
7      public void runDevice() {
8          log.debug("Starting HCSR501 sensor");
9
10         // Already running
11         if (gpioController != null) {
12             log.debug("Starting HCSR501 sensor failed. Already started");
13             return;
14         }
15
16         // get instance of gpio controller
17         gpioController = GpioFactory.getInstance();
18
19         // set data pin to observe
20         final GpioPinDigitalInput sensor = gpioController.provisionDigitalInputPin(RaspiPin.GPIO_15,
21             "hcsr501");
22
23         // add listener to handle events
24         sensor.addListener((GpioPinListenerDigital) event -> {
25             log.debug("HCSR501 sensor at pin '{}' changed state to '{}'", event.getPin(), event.getState());
26
27             // check state of pin
28             if (PinState.HIGH.equals(event.getState())) {
29                 // ...
29             }
29
30             log.debug("Registered sensor listener");
31             log.debug("Started HCSR501 sensor");
32         }
33
34         // ...
35     }
36 }
37

```

### 3.3 Mobiler Client

Beim *RPISec Client* handelt es sich um eine native *Android App*. Diese kommuniziert über REST-Schnittstellen mit einem *Auth-Service* sowie mit *Firebase*-Diensten von Google. Die für diese App verwendeten *Firebase*-Dienste sind *Firebase-Messaging* und *Firebase-RealTimeDatabase*.

#### 3.3.1 Login

Die Startseite der App besteht nur aus einem Login-Fenster (Abbildung 15). Der Login erfolgt in drei Schritten.

1. RPISec Login

Die App generiert einen UUID-Wert (*Universally Unique Identifier*) als Identifikator für das Gerät. Zusammen mit dem eingegebenen Benutzernamen, Passwort und der UUID wird eine Anfrage für einen *Firebase*-Token an den *Auth-Service* gestellt. Sind die Zugangsdaten korrekt, wird für diese UUID ein Token, sowie *Client-Id* und *Client-Secret* erzeugt und als Antwort an die *App* übertragen. Der *Firebase*-Token wird für die Authentifizierung bei den verbunden Diensten benutzt.

### Quelltext 2: ClientLoginOAuthTask.java

```
1 private class ClientLoginOAuthTask extends AsyncTask<Void, Void, Boolean> {
2
3     @Override
4     protected Boolean doInBackground(Void... params) {
5         try {
6             if (oAuthCredentials != null) {
7
8                 try {
9                     at.rpisec.swagger.client.auth.model.TokenResponse res =
10                        → getAuthClientApi(oAuthCredentials.getUserName(),
11                        → oAuthCredentials.getPassword()).loginUsingGET(getGeneratedUUID());
12                     oAuthCredentials.setToken(res.getToken());
13                     oAuthCredentials.setClientId(res.getClientId());
14                     oAuthCredentials.setClientSecret(res.getClientSecret());
15
16                     Log.v(DEBUG_LOGIN_TAG, "[ClientId] " + res.getClientId());
17                     Log.v(DEBUG_LOGIN_TAG, "[ClientSecret] " + res.getClientSecret());
18                     return true;
19                     // thrown in case of null parameter or if required or 200 > status > 300
20                 } catch (ApiException e) {
21                     Log.v(DEBUG_LOGIN_TAG, e.getMessage());
22                 }
23             } catch (Exception e) {
24                 Log.v(DEBUG_LOGIN_TAG, e.getMessage());
25             }
26             return false;
27         }
28     }
29 }
```

## 2. Firebase Login

Im zweiten Schritt wird ein Token für *Firebase-Messaging* registriert. Dies erfolgt wiederum mit dem Benutzernamen und Passwort. Nach der Registrierung kann die *App* Nachrichten über den *Firebase-Messaging* Dienst empfangen.

### Quelltext 3: RegisterFCMTask.java

```
1 private class RegisterFCMTask extends AsyncTask<Void, Void, Boolean> {
2     private final String fcmToken = FirebaseInstanceId.getInstance().getToken();
3
4     @Override
5     protected Boolean doInBackground(Void... params) {
6         try {
7             getAuthClientApi(oAuthCredentials.getUserName(),
8                 → oAuthCredentials.getPassword()).registerFCMTokenUsingPUT(getGeneratedUUID(), fcmToken);
9             return true;
10        } catch (ApiException e) {
11            // thrown in case of null parameter or if required or 200 > status > 300
12            return false;
13        }
14    }
15 }
```

## 3. Firebase-Token Registrierung

Am Schluss wird der *Firebase*-Token am *Auth-Service* registriert.

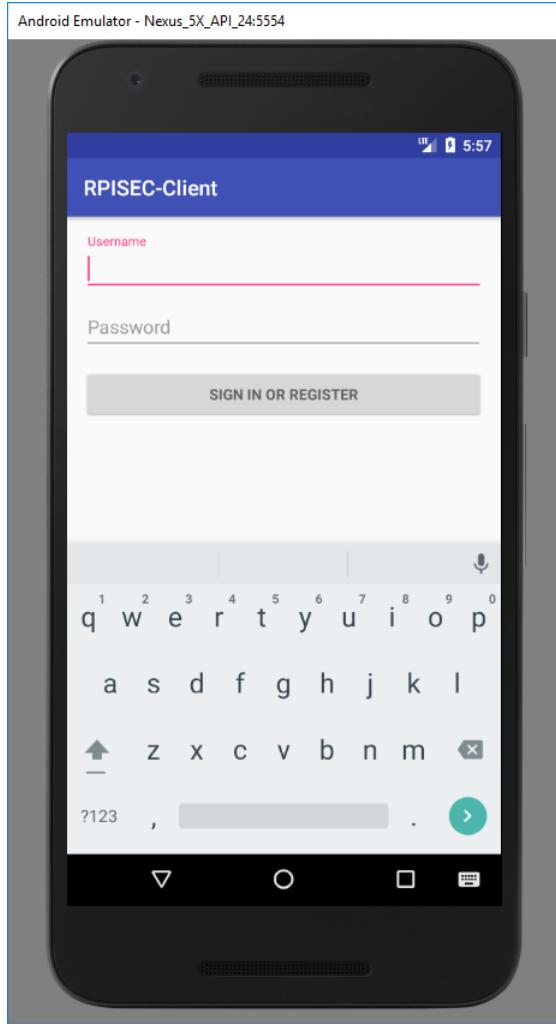


Abbildung 15: Pin-Schema

Nach einem erfolgreichen Login wird automatisch die Detailansicht (siehe Abbildung 16) gestartet.

### 3.3.2 Detailansicht

Bei der Detailansicht handelt es sich um eine Übersicht aller Bilder, chronologisch geordnet nach Aufnahmedatum, in einem Raster. Bei den angezeigten Bildern handelt es sich nur um *Thumbnails*. Die richtigen Bilder werden beim Auswählen eines Bildes angezeigt, ähnlich dem *Fotoviewer* in Android.

Die Detailansicht wird beim Eintreffen eines neuen Bildes aktualisiert. Sollte die Detailansicht nicht die aktive Anwendung sein, wird in der Infoleiste eine Notifikation angezeigt und ein Ton abgespielt. Durch tippen auf die Notifikation wird die Anwendung wieder in den Vordergrund geholt.

Mittels Swipe-Down-Geste kann die Ansicht aktualisiert werden.

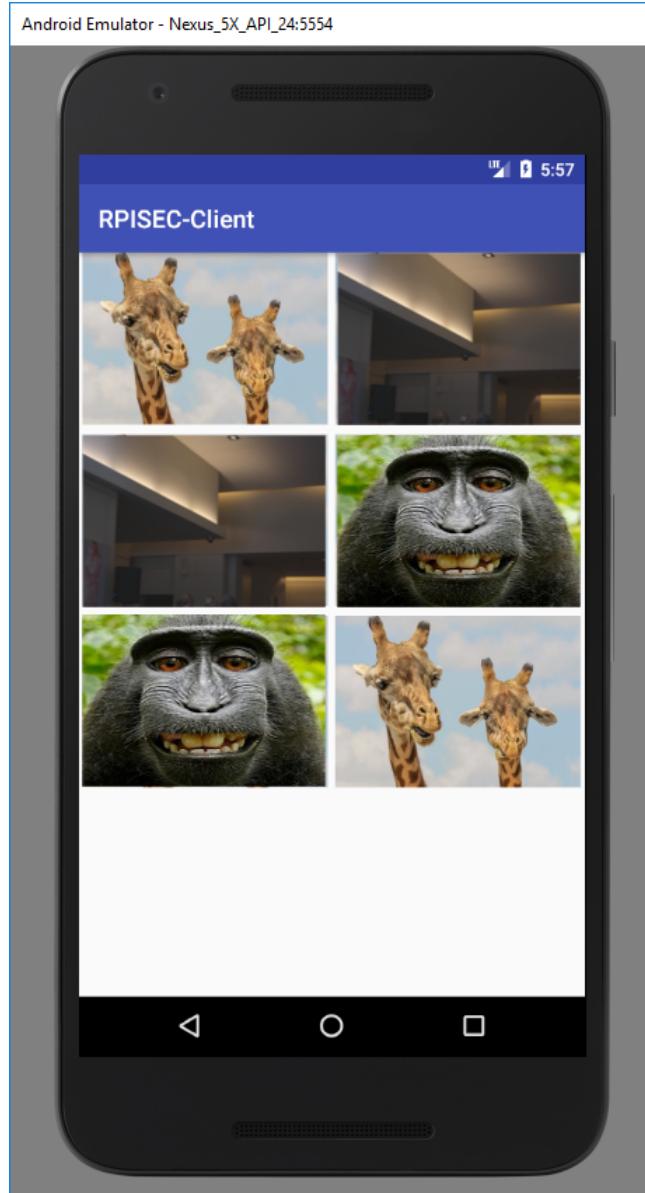


Abbildung 16: Pin-Schema

### 3.3.3 Datenabfrage

Bei jedem *Incident* der von der Server-Anwendung über Firebase-Messaging an die App gesendet wird, werden der Titel, eine Nachricht und die Id des aufgenommenen Bildes übertragen. Mit dieser Id kann das eigentliche Bild aus der Firebase-RealTimeDatabase abgerufen werden.

Quelltext 4: FirebaseMessaging.java

```
1 // incoming messages from firebase messaging service
2 public void onMessageReceived(RemoteMessage remoteMessage) {
3     if (remoteMessage.getData().size() > 0 &&
4         !lastIncidentKey.equals(remoteMessage.getData().get(FirebaseConstants.FBM INCIDENT MESSAGE KEY ID)))
5     {
6         lastIncidentKey = remoteMessage.getData().get(FirebaseConstants.FBM INCIDENT MESSAGE KEY ID);
7         processData(remoteMessage.getData());
8     }
9
10    private void processData(final Map<String, String> data) {
11        if (data.containsKey(FirebaseConstants.FBM INCIDENT MESSAGE KEY ID)) {
12            // get database instance
13            FirebaseDatabase database = FirebaseDatabase.getInstance();
14
15            if (database != null) {
16                DatabaseReference incidentDBRef = database.getReference(FirebaseConstants.DB ITEM INCIDENT);
17
18                if (incidentDBRef != null) {
19                    // get item data
20                    Query incidentQuery =
21                        incidentDBRef.child(data.get(FirebaseConstants.FBM INCIDENT MESSAGE KEY ID));
22
23                    if (incidentQuery != null) {
24                        // async listener - wait for query finished
25                        incidentQuery.addValueEventListener(new ValueEventListener() {
26
27                            @Override
28                            public void onDataChange(DataSnapshot dataSnapshot) {
29                                FirebaseDatabaseItem item = dataSnapshot.getValue(FirebaseDatabaseItem.class);
30                                if (item != null) {
31
32                                    String fileName = data.get(FirebaseConstants.FBM INCIDENT MESSAGE KEY ID) + "." +
33                                    item.getDataType();
34                                    File storageDir = getExternalFilesDir(Environment.DIRECTORY_PICTURES);
35
36                                    byte[] imageAsBytes = Base64.decode(item.getBase64Data().getBytes(), Base64.DEFAULT);
37                                    Bitmap bmp = BitmapFactory.decodeByteArray(imageAsBytes, 0, imageAsBytes.length);
38                                    // ...
39                                    // save image into directory
40                                    // ...
41
42                                    // send notification
43                                    sendNotification(msgTitle, msgBody);
44                                }
45
46                            }
47
48                            @Override
49                            public void onCancelled(DatabaseError databaseError) {
50
51                            }
52                        });
53                    }
54                }
55            }
56        }
57    }
58}
```

## 3.4 Datenbank

Dieser Abschnitt behandelt die verwendete Datenbank für die *Microservices*. Im Entwicklungsbetrieb auf einen Entwicklerrechner wird die Datenbank H2 und im produktiven Betrieb auf einen *Raspberry PI* die Datenbank PostgreSQL verwendet. Die Datenbank PostgreSQL konnte am *Raspberry PI* verwendet werden, da PostgreSQL die ARM Architektur unterstützt und sich auch mit geringen Ressourcenaufwand betreiben lässt.

## 3.5 Firebase

Dieser Abschnitt behandelt den verwendeten *Cloud* Dienst *Firebase*. *Firebase* ist ein *Cloud* Dienst von *Google*, der einen *Messaging* Dienst sowie eine Onlinedatenbank (JSON-Datenbank) bereitstellt. Bis zu einer gewissen Anzahl von *Requests* ist dieser Dienst kostenlos zu verwenden.

Für *Firebase* gibt es eine Java Implementierung das sogenannte *firebase-admin-sdk*, das eine API zum Interagieren mit der JSON-Datenbank und eine API zum Erstellen von Authentifizierungstoken für die *Client*-Authentifizierung auf *Firebase* zur Verfügung stellt. In der Java Implementierung wird zurzeit keine API für die Interaktion mit dem *Messaging* Dienst zur Verfügung gestellt, was aber kein Problem darstellt, da es sich hierbei um eine einfache Anfrage an eine *REST-API* handelt, die mit Spring *RestTemplate* realisiert wurde.

Für die Interaktion mit *Firebase* über *Android* wird ebenfalls eine Java Implementierung bereitgestellt. Diese Implementierung enthält auch eine *API* für den *Messaging* Dienst von *Firebase*.

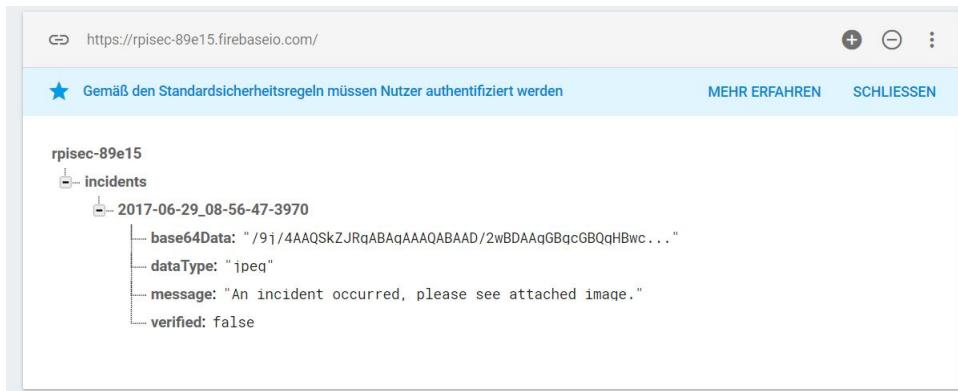


Abbildung 17: Eintrag in der *Firebase* Onlinedatenbank

### 3.6 Docker

Dieser Abschnitt behandelt die Verwendung von *Docker*, für das hosten der Services und Datenbanken für die Services. Dank dem *Raspberry PI Image*, bereitgestellt von *Hypriots*, sind bereits *Docker* und *Docker-Compose (Orchestration)* vorinstalliert und müssen nicht separat geladen und für die Zielarchitektur gebaut werden.

Da der Umgang mit *Docker* und einer umfangreicheren Infrastruktur mit viel Shell-Skripten verbunden ist, wird das Python basierte Tool *Docker-Compose* verwendet, das es erlaubt eine Infrastruktur, die aus einer Menge von untereinander abhängigen Services besteht, deklarativ über eine *YAML*-Konfigurationsdatei zu konfigurieren und zu *orchestrieren*.

Die Definition der *Images* in Form von *Dockerfiles* sowie der Aufbau der *Docker-Compose* Infrastruktur sind im Verzeichnis */host/docker/* enthalten, wobei die einzelnen *Dockerfiles* der Services in Unterverzeichnissen organisiert wurden, die alle Abhängigkeiten, welche in die Images mitaufgenommen werden müssen, enthalten. Die *PostgreSQL* Images stehen am *Docker Hub*<sup>11</sup> zur Verfügung. Es wurde ein eigenes Basisimage definiert, das die benötigten Abhängigkeiten für die Interaktion mit der Hardware beinhaltet. Es werden in diesem Basisimage alle benötigten C-Bibliotheken wie *WiringPi*<sup>12</sup> und die Bibliotheken für das Interagieren mit der *Raspberry Pi GPU*<sup>13</sup> während des Bauens des Images geladen und für die Zielplattform kompiliert. Das eigenen Basisimage wurde eingeführt, da das Bauen der Abhängigkeiten aufwendig ist und viel Zeit in Anspruch nimmt und sich das Basisimage nur selten ändert.

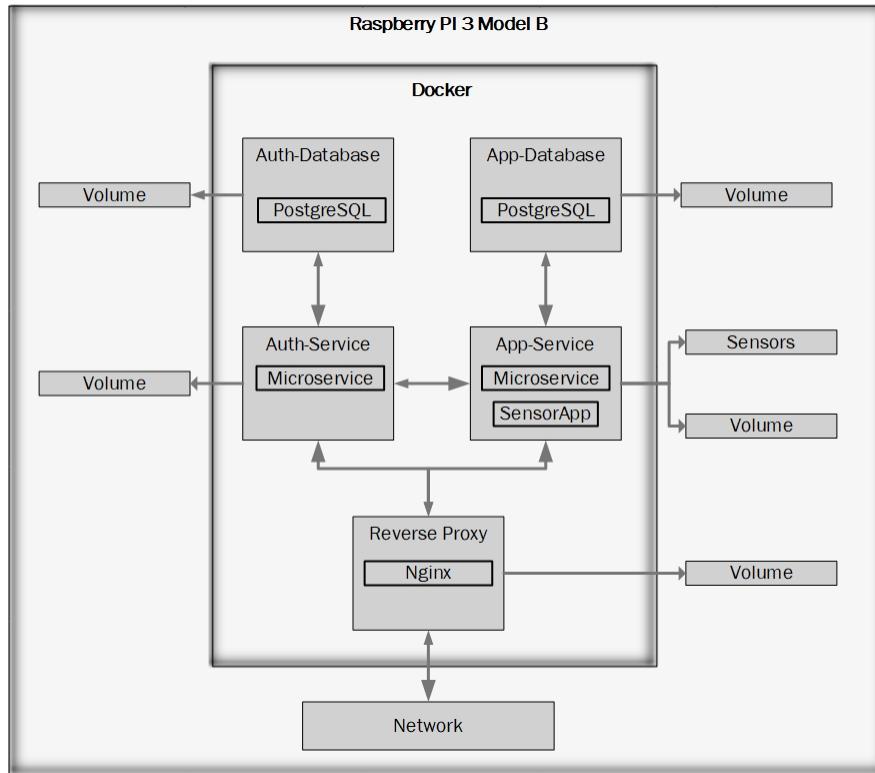


Abbildung 18: *Raspberry PI Docker* Infrastruktur

<sup>11</sup><https://hub.docker.com/r/tobi312/rpi-postgresql/>

<sup>12</sup><https://git.drogon.net/?p=wiringPi;a=summary>

<sup>13</sup><https://github.com/raspberrypi/userland>

Die Abbildung 18 zeigt die implementierte *Docker* Infrastruktur, wie sie am *Raspberry PI* gehostet wird. Da die Daten der *Docker-Container* persistent gehalten werden müssen, werden die Daten in den *Containern* in Verzeichnissen gehalten, die auf den *Host* über ein *Volume Mapping* gebunden sind. Der *Docker Container App-Service* benötigt privilegierte Rechte, damit die Sensorapplikation mit der angeschlossenen *Hardware* des *Raspberry PI* kommunizieren kann.

Der Quelltext 5 zeigt den Inhalt der *docker-compose.yml*, welche die *Docker* Infrastruktur für *RPI-Sec* am *Raspberry PI* definiert. Die in der Datei vorkommenden Textfragmente im Format \${...} stellen Variablen dar, die *Docker-Compose* entweder aus einer Datei mit dem Namen *.env*, die auf derselben Ebene wie die *docker-compose.yml* platziert werden muss, oder aus den Umgebungsvariablen des Benutzers, mit dem die Infrastruktur erstellt wird, auflöst. Sollten Variablen nicht auflösbar sein, so wird eine entsprechende Meldung auf die Konsole ausgegeben.

Quelltext 5: docker-compose.yml für RPISec am *Raspberry PI*

```

1 version: "2.1"
2 services:
3     rpisec-app-db:
4         container_name: rpisec-app-db
5         image: tobi312/rpi-postgresql:9.6
6         environment:
7             - POSTGRES_USER=rpisec-app
8             - POSTGRES_PASSWORD=rpisec-app
9             - POSTGRES_DATABASE=rpisec-app
10        mem_limit: 100m
11        cpu_shares: 2
12        volumes:
13            - ${APP_DB_VOLUME}:/var/lib/postgresql/data:rw
14    rpisec-auth-db:
15        container_name: rpisec-auth-db
16        image: tobi312/rpi-postgresql:9.6
17        environment:
18            - POSTGRES_USER=rpisec-auth
19            - POSTGRES_PASSWORD=rpisec-auth
20            - POSTGRES_DATABASE=rpisec-auth
21        mem_limit: 100m
22        cpu_shares: 2
23        volumes:
24            - ${AUTH_DB_VOLUME}:/var/lib/postgresql/data:rw
25    rpisec-app:
26        container_name: rpisec-app
27        build:
28            context: ./app
29            args:
30                - VIDEO_GUID=44
31                - UID=1000
32            volumes:
33                - ${APP_CONF_VOLUME}:/home/app/conf:ro
34                - ${APP_LOG_VOLUME}:/home/app/log:rw
35                - ${APP_IMAGE_VOLUME}:/home/app/image:rw
36            environment:
37                - APP_JAVA_OPTS=-Xms128m -Xmx200m
38        mem_limit: 256m
39        cpu_shares: 4
40        privileged: true
41        depends_on:
42            - rpisec-app-db
43    rpisec-auth:
44        container_name: rpisec-auth
45        build:
46            context: ./auth
47        volumes:
48            - ${AUTH_CONF_VOLUME}:/home/auth/conf:ro
49            - ${AUTH_LOG_VOLUME}:/home/auth/log:rw
50        environment:
```

```

51      - AUTH_JAVA_OPTS=-Dadmin.email=fh.ooe.mus.rpiseic@gmail.com -Xms128m -Xmx200m
52  mem_limit: 256m
53  cpu_shares: 4
54  depends_on:
55      - rpiseic-auth-db
56      - rpiseic-app
57 rpiseic-nginx:
58   container_name: rpiseic-nginx
59   image: rpiseic-nginx
60   build:
61     context: ./nginx
62   volumes:
63     - ${NGINX_LOG_VOLUME}:/var/log/nginx:rw
64     - ${NGINX_CERT_VOLUME}:/cert:ro
65   mem_limit: 128m
66   cpu_shares: 4
67   ports:
68     - 443:443
69     - 80:80
70   depends_on:
71     - rpiseic-app-db
72     - rpiseic-auth-db
73     - rpiseic-app
74     - rpiseic-auth

```

Das Bauen der Infrastruktur und Starten der Services dauert am *Raspberry PI* relativ lange, da nur wenig Speicher zur Verfügung steht, es sich um eine ARM-Architektur handelt und das Speichermedium eine *MicroSD* Karte ist. Ebenso ist die Performance der Services nicht herausragend, jedoch kann die Applikation auf dem *Raspberry PI* problemlos ausgeführt werden.

## 4 Erfahrungen

Dieser Abschnitt behandelt die gemachten Erfahrungen während der Umsetzung dieses Projekts. Nachdem für dem *Raspberry PI* bereits ein Betriebssystem zur Verfügung steht, das alle benötigten Bibliotheken für wie *Docker* und *Docker-Compose* bereitstellt und es auch eine *ARM* Implementierung von *Oracle-Java* gibt, hat sich die Umsetzung als relativ einfach gestaltet. Hätte es keine Oracle Implementierung für die *ARM*-Plattform gegeben, hätte man *OpenJDK* verwenden müssen, was das Laufzeitverhalten der *Microservices* negativ beeinflusst hätte.

Die verwendete *PostgreSQL* Datenbank, die in einem *Docker Container* gehostet wird, braucht beim ersten Start (Keine Datenbank vorhanden) relativ lange, weswegen der erste Start der *Docker-Compose* orchestrierten Infrastruktur beim ersten Mal fehlschlägt, da die *Microservices* gestartet sind und auf eine nicht verfügbare Datenbank zugreifen wollen, die noch im Initialisierungsprozess feststeckt. Dieses Problem wurde gelöst indem die Datenbanken initialisiert werden, bevor die gesamte Infrastruktur gestartet wird.

Die Java Bibliothek für den *Cloud*-Dienst *Firebase* ist zwar eingeschränkt verglichen mit der Implementierung für *NodeJS*, jedoch konnten alle Tasks, bis auf das Versenden der *Messages*, das mit Spring *Resttemplate* realisiert wurde, einfach implementiert werden.

Für die Umsetzung der *Microservices* mussten keine besonderen Vorkehrungen wegen dem Hosten auf einen *Raspberry PI* getroffen werden. Nichts desto trotz sollte man sich bei der Implementierung der *Microservices* bezüglich der Abhängigkeiten und verwendeten *Frameworks* bewusst sein, dass man mit einem System konfrontiert ist, das nur beschränkte Ressourcen zur Verfügung stellt.

Abschließend kann man sagen das *Docker* und Java Programme auf einen *Raspberry PI* sehr gut zu betreiben sind, solange man nicht Programme und *Docker Container* betreiben will, für welche die zur Verfügung stehenden Ressourcen nicht oder nur knapp ausreichen. Vor allem das Zeitverhalten muss berücksichtigt werden, dass aufgrund der *ARM*-Architektur sich nicht gleich verhält als auf einer *x86*-Architektur. *PI4J* bietet einen sehr gute *API* für die Interaktion mit den *GPIO* und für das Ausführen von Prozessen am Hostsystem.