

Raspberry PI Security Application

Thonas Herzog, Philipp Wurm

18. Juni 2017

1 Einleitung

Diese Dokument stellt die Dokumentation des Projekts *Raspberry PI Security Application*, in weiterer Folge *RPISec* genannt, dar, das für die Lehrveranstaltung *Mobile und ubiquitäre Systeme* realisiert wurde. In diesem Projekt wurde eine Heimsicherheitsanwendung mit Raspberry PI, Docker und Spring realisiert, die bei einem Sicherheitsverstoß in der Lage ist, bekannte mobile Endgeräte von registrierten Benutzern über diesen Sicherheitsverstoß zu informieren.

1.1 Problemdarstellung

Dieser Abschnitt behandelt die Problemdarstellung, welche die Grundlage für die zu implementierende *Raspberry PI Security Application* ist. Bei einem Auslösen eines Bewegungssensors in einem Haushalt sollen alle Bewohner über ihre mobilen Endgeräte wie Handy und Tablet über den Vorfall informiert werden sowie ein Foto erhalten, das den Sicherheitsbereich, zum Zeitpunkt wann der Bewegungsmelder ausgelöst wurde, zeigt. Des weiteren soll es zu jedem Zeitpunkt möglich sein sich ein aktuelles Foto des Sicherheitsbereichs über ein mobiles Endgerät zu beziehen.

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 Authentifizierungen durchzuführen. Da sich die mobilen Endgeräte in irgendwelchen Netzen ans 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 muss es erlauben, dass die Benutzerverwaltung von einem anderen Dienst übernommen werden kann, da wir diesen *Messaging* Dienst nicht vertrauen wollen und daher den *Messaging* Dienst auch nicht die Benutzerverwaltung überlassen wollen.

1.2 Funktionsweise

Dieser Abschnitt behandelt die Funktionsweise der Applikation *RPISec*. Die Abbildung 1 zeigt den Systemaufbau der *RPISec* Applikation.

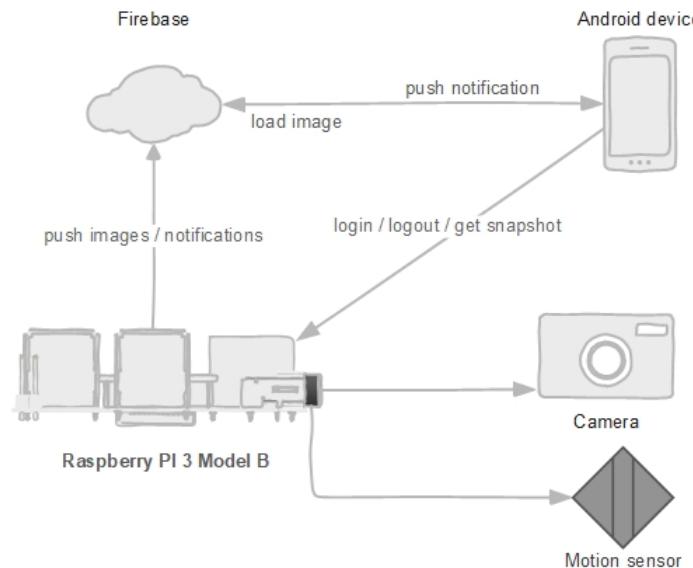


Abbildung 1: Systemaufbau der *RPISec* Applikation

1.2.1 Zugangsbestätigung eines neuen Benutzers

Beim Start des *Auth-Service* wird ein Administrator Benutzer erstellt, der über eine E-Mail dazu aufgefordert wird, seinen Zugang zu aktivieren, in dem er ein Password für den Benutzer vergeben wird.

A screenshot of a web-based password confirmation form. At the top is a field labeled "Password". Below it is another field labeled "Confirmation". At the bottom is a blue rectangular button labeled "Submit".

Abbildung 2: Zugangsaktivierung

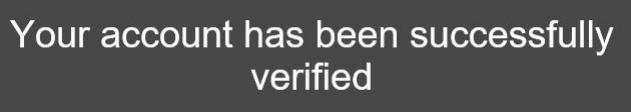


Abbildung 3: Bestätigung der Aktivierung

1.2.2 Client Login

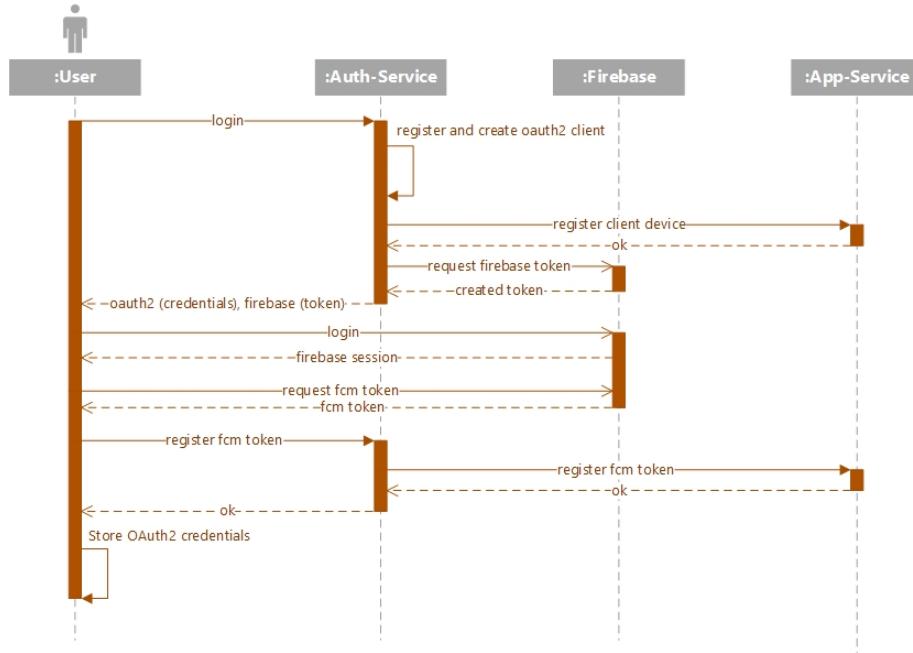


Abbildung 4: Sequenzdiagramm des Logins über einen mobilen Client

Die Abbildung 4 zeigt das Sequenzdiagramm das den erfolgreichen Ablauf des Logins eines Benutzers über ein Android Gerät beschreibt. Im Zuge des Logins wird das mobile Endgerät am Authentifizierungsservice und Applikationsservice registriert und für jeden Login ein neuer *OAuth2 Client* angelegt und gegebenenfalls der alte *OAuth2 Client* für dieses Endgerät gelöscht. Beziiglich OAuth2 wurde dieser Ansatz gewählt, da mit der Client-Applikation keine *Oauth2 Client* Zugangsdaten ausgeliefert sollen.

1.2.3 Sicherheitsvorfall melden

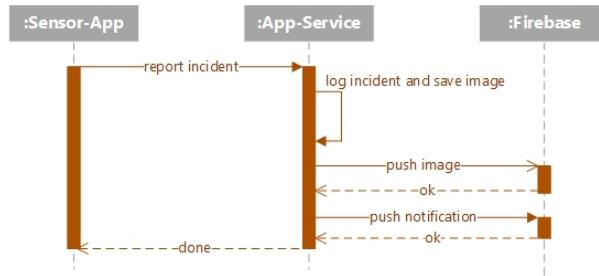


Abbildung 5: Sequenzdiagramm des Behandelns eines Sicherheitsvorfalls

Die Abbildung 5 zeigt das Sequenzdiagramm für das Behandeln eines Sicherheitsvorfalls, der von der Sensorapplikation erkannt und dem Applikationsservice mitgeteilt wurde. Der Vorfall wird über *Firebase* an die *Clients* gemeldet, wobei einerseits eine *Cloud Message* an die *Clients* versendet wird, sowie das gemachte Bild in der *Firebase* JSON-Datenbank den *Clients* zum Download zur Verfügung gestellt wird.

1.2.4 Client Nachrichtenempfang

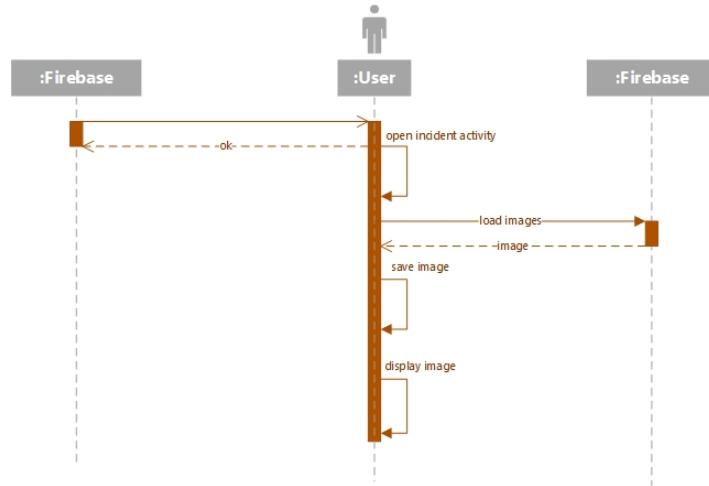


Abbildung 6: Sequenzdiagramm der Benachrichtigung eines *Client*

Die Abbildung 6 zeigt den Ablauf einer Benachrichtigung eines *Client* über den *Firebase Messaging* Dienst. Nachdem auf die Nachricht geklickt wurde, wird eine *Activity* für das Anzeigen der Bilder geöffnet, die alle bereits gespeicherten Bilder und das neu geladene Bild anzeigt.

2 *Raspberry PI*

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

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

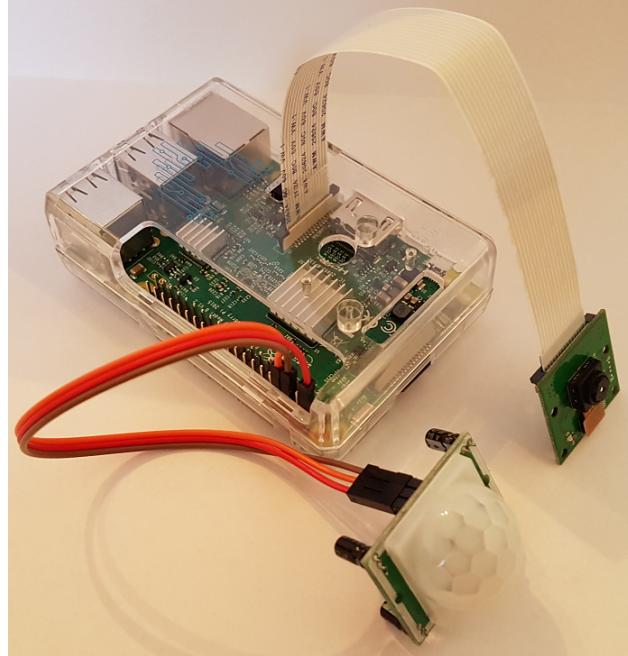


Abbildung 7: Testaufbau der Applikation

Wie in Abbildung 7 zu ersichtlich ist, wurde die Kamera über CSI (*Camera-Serial-Interface*) und der Bewegungssensor über GPIO (*General Purpose Input/Output*) an den *Raspberry PI* angeschlossen.

2.1 Betriebssysteme

Dieser Abschnitt behandelt die verwendete Betriebssysteme für den *Raspberry PI*. Die Applikation *RPISec* wurde einerseits mit dem Betriebssystem *hypriotos-rpi* und andererseits mit *Raspian* realisiert. Das Betriebssystem *hypriots* basiert auf *Debian Jessie* und wird von dem *OpenSource* Projekt *hypriot*⁴ zur Verfügung gestellt wird. Das Ziel von *hypriots* ist es ein Betriebssystem für *Raspberry PI* zur Verfügung stellen, das bereits Docker vorinstalliert und betriebsbereit hat. Mit dem Betriebssystem *Raspian* muss Docker selbst installiert, 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.

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

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

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

⁴<https://blog.hypriot.com/>

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.

3 Software

Dieser Abschnitt behandelt die verwendete bzw. implementierte Software für *RPISec*.

3.1 *Microservices* und *Cloud*

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* *rpisec-auth-service* für die Benutzerverwaltung und OAuth2 Authentifizierung und *rpisec-app-service* für die Interaktion mit der Sensorik und der Interaktion mit dem *Cloud*-Diensten implementiert, wobei der *Microservice* *rpisec-auth-service* im Zuge des Projekts für die Lehrveranstaltung *Service Engineering* implementiert wurde. Es hätte auch ausgereicht die Benutzerverwaltung in den *Microservice* *rpisec-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* *rpisec-app-service* interagiert nicht direkt mit der Sensorik, sondern bindet die Sensorapplikation beschrieben in Abschnitt ?? ein und ist für dessen Lebenszyklus verantwortlich. Nachdem Start der Sensorapplikation wird ein *Listener* registriert, der auf Statusänderungen des Bewegungssensor reagiert und diesen Sicherheitsvorfall wie in Abbildung 5 behandelt.

Die beiden *Microservices* müssen Daten persistent halten und sind daher auf eine Datenbank angewiesen, wobei im Entwicklungsbetrieb auf einen Entwicklerrechner H2 und im produktiven Betrieb auf einen *Raspberry PI* PostgreSQL verwendet wird. Die Datenbank PostgreSQL konnte verwendet werden, da PostgreSQL die ARM Architektur unterstützt.

Als *Cloud* Anbieter wurde *Google* gewählt, welcher die Plattform *Firebase* anbietet, die eine JSON-Datenbank und einen *Cloud Messaging* Dienst anbietet. Für diesen Dienst 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 bei 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* durchgeführt wird.

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*

3.3 Kamera

Bei der Kamera handelt es sich um ein eigens für den Raspberry Pi entwickeltes Modell und wird direkt an den vorhanden Kamera-Port des Raspberry Pi 3⁵ angeschlossen.

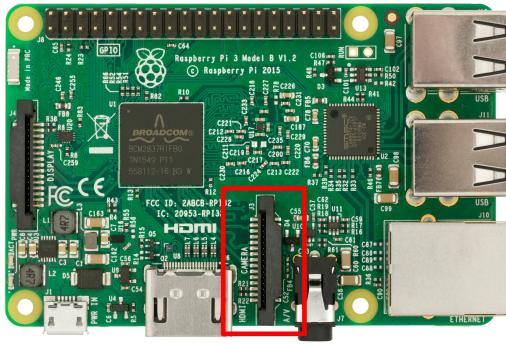


Abbildung 8: Raspberry-Pi 3

Anschließend muss die Kamera aktiviert werden. Dies geschieht über das, mit den meisten Linux-Distributionen mitgelieferte, Programm *raspi-config*⁶. Dieses Programm wird auch benutzt um andere Komponenten des Raspberry Pi zu konfigurieren.

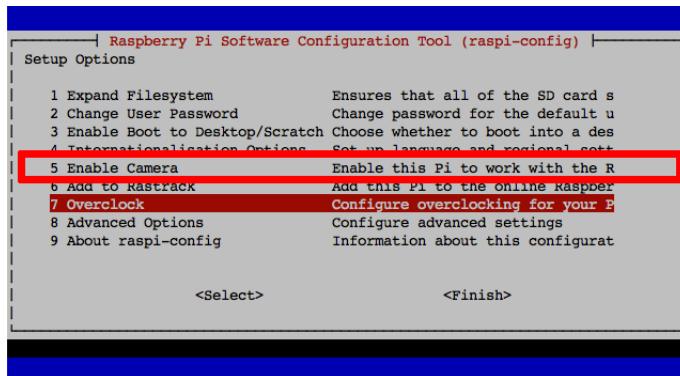


Abbildung 9: Raspi-config

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*.

⁵https://en.wikipedia.org/wiki/Raspberry_Pi#/media/File:Raspberry-Pi-3-Flat-Top.jpg

⁶<https://upload.wikimedia.org/wikipedia/commons/e/ed/Raspi-config.png>

3.4 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.

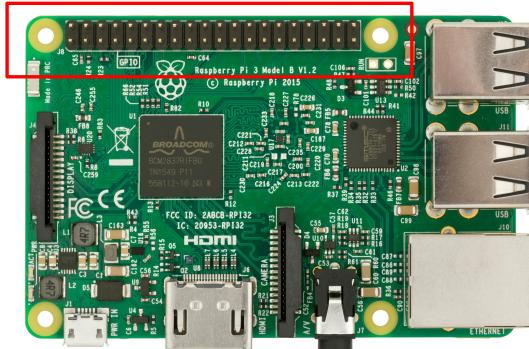


Abbildung 10: Raspberry-Pi 3

Die Pin-Belegung⁷ des HC-SR501

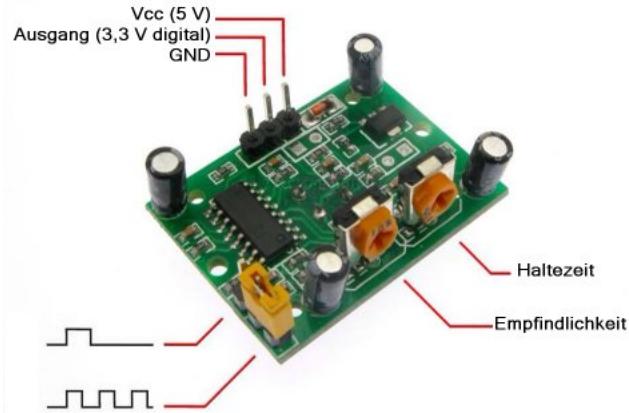


Abbildung 11: Pin-Schema HC-SR501

ist wie folgt definiert:

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

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

⁷<http://www.netzmafia.de/skripten/hardware/RasPi/Projekt-PIR/>

3.5 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 I2C, SPI und eine serielle Schnittstelle.

3.5.1 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 Darstellung⁸:

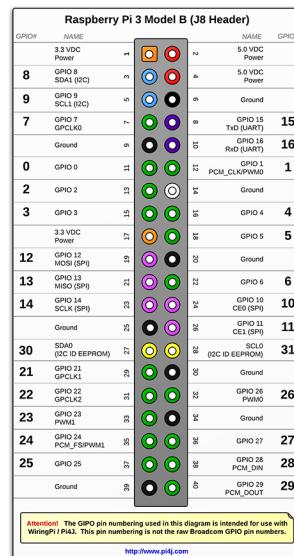


Abbildung 12: Pin-Schema

⁸<http://pi4j.com/pins/model-3b-rev1.html>

3.6 Pin-Ansteuerung mit Java

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

3.6.1 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.

3.6.2 WiringPi

WiringPi ist ein nützliches Framework 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.

Quelltext 1: gpio-controller.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                // ...
30            });
31
32            log.debug("Registered sensor listener");
33            log.debug("Started HCSR501 sensor");
34        }
35
36        // ...
37    }
```

⁹<http://pi4j.com/>

¹⁰<http://wiringpi.com/>

3.7 Mobiler Client

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

3.7.1 Login

Die Startseite der App besteht nur aus einem Login-Fenster (Abbildung 13). Der Login erfolgt in zwei Schritten.

1. OAuth 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 OAuth-Token an den OAuth-Server 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 OAuth-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     }
```

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    }
}
```

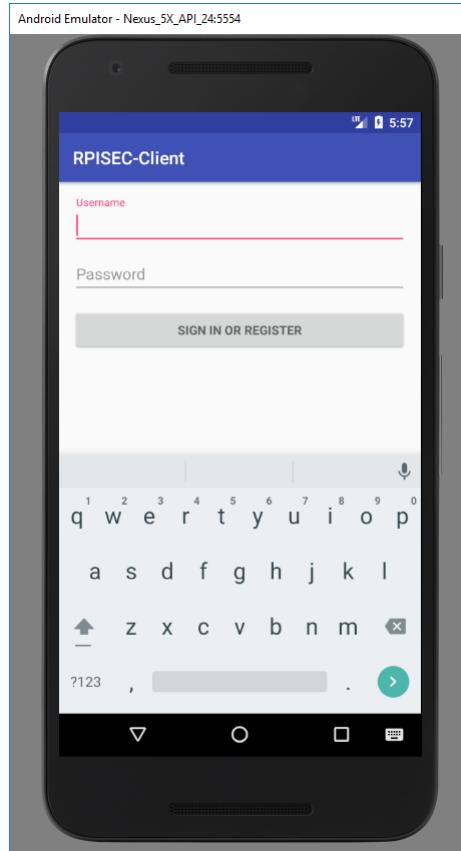


Abbildung 13: Pin-Schema

Nach einem erfolgreichen Login wird automatisch die Detailansicht (siehe Abbildung 14) gestartet.

3.7.2 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.

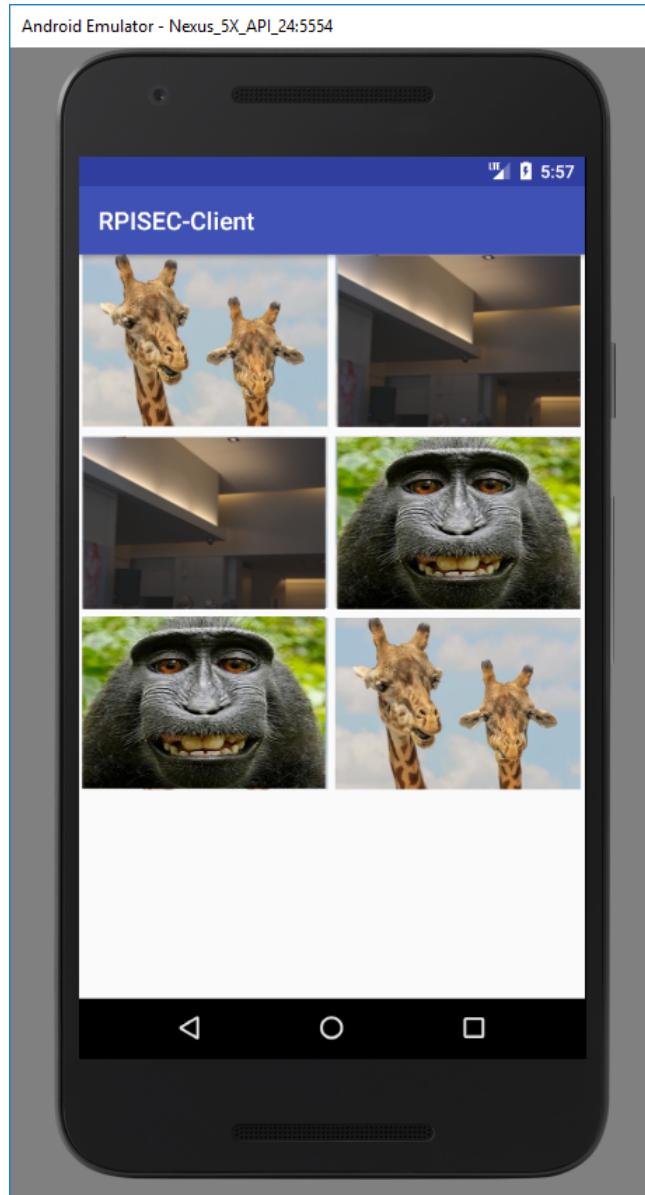


Abbildung 14: Pin-Schema

3.7.3 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 imageFileName = 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                            @Override
48                            public void onCancelled(DatabaseError databaseError) {
49
50                            }
51                        });
52                    }
53                }
54            }
55        }
56    }
57}
```

4 Docker Infrastruktur

Dieser Abschnitt behandelt die *Docker* Infrastruktur, welche die Service und deren Abhängigkeiten *hosted*. 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.

Die Definition der *Images* sowie der Aufbau der *Docker* Infrastruktur sind im Verzeichnis */host/docker/* enthalten, wobei die einzelnen *Dockerfiles* der Services in Unterverzeichnissen organisiert, die alle Abhängigkeiten, welche in die Images mitaufgenommen werden, enthalten. Die *PostgreSQL* Images stehen am *Docker Hub*¹¹ zur Verfügung.

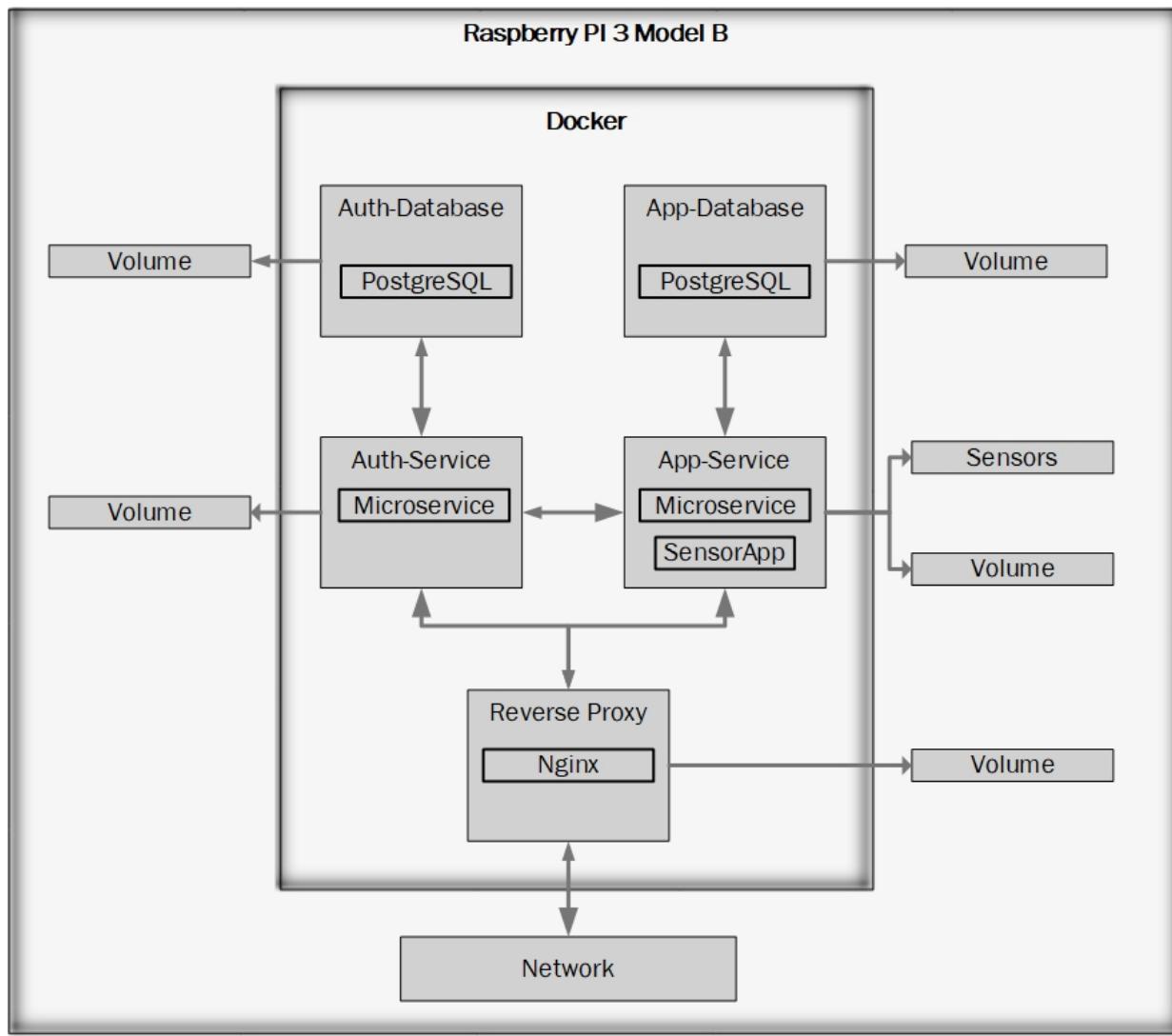


Abbildung 15: *Raspberry PI Docker* Infrastruktur

Die Abbildung 15 zeigt die *Docker* Infrastruktur, wie sie am *Raspberry PI* angewendet wird. Da die Daten persistent gehalten werden müssen, werden die Daten in den *Containern* in Verzeichnissen gehalten, die

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

auf den *Host* über ein *Volume* gebunden sind. Der *Docker Container App-Service* benötigt privilegierte Rechte damit die Sensorapplikation mit der angeschlossenen *Hardware* kommunizieren kann. Die benötigten C-Bibliotheken wie *WiringPI*¹² und die Bibliotheken für das Interagieren mit der *Raspberry Pi GPU*¹³ werden in einem Basisimage während des Bauens geladen und kompiliert. Dieses Basisimage stellt die Basis aller eigenen Images dar.

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.oeo.mus.rpisec@gmail.com -Xms128m -Xmx200m
52        mem_limit: 256m
53        cpu_shares: 4
54        depends_on:
55            - rpisec-auth-db
56            - rpisec-app

```

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

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

```

57   rpisec-nginx:
58     container_name: rpisec-nginx
59     image: rpisec-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       - rpisec-app-db
72       - rpisec-auth-db
73       - rpisec-app
74       - rpisec-auth

```

Der Quelltext 5 zeigt den Inhalt der *docker-compose.yml*, welche die *Docker* Infrastruktur für *RPIsec* 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.

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.