HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

ins institute for networked solutions

Internet of Things mit openHAB und Microsoft Azure

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2015

Autoren:	Marco Leutenegger, Dominik Freier
Betreuer:	Prof. Hansjörg Huser
Projektpartner:	INS - Institute for Networked Solutions
Experte:	Stefan Zettel, Ascentive Zürich
Gegenleser:	Prof. Dr. Andreas Rinkel

Abstract

Unsere Arbeit befasst sich mit dem Thema «Internet of Things». Das Ziel ist der Aufbau einer Smart-Home Beispielapplikation. Es sollen wesentliche Aspekte einer Internet of Things Anwendung demonstriert werden, die in einem praktischen Szenario denkbar sind. Wir haben uns das Szenario Einbruchschutz erarbeitet und möchten zeigen, wie verschiedene Produkte kombiniert werden können, so wie es in einem Privathaushalt üblich ist. Dazu sollen verschiedene Hardwareteile, Technologien und Frameworks analysiert und in einen Versuchsaufbau integriert werden. Das Szenario umfasst auch eine Cloud-Anwendung, die mit Microsoft Azure umzusetzen ist.

Wir haben entsprechend den Anforderungen das Framework openHAB evaluiert. Aufgrund der modularen Architektur, die auf OSGi basiert, können diverse Protokolle und Features in eine bestehende Installation zur Laufzeit hinzugefügt werden. OpenHAB bezeichnet sich selbst als Platform für das Intranet of Things. Mit unserer MQTT-Verbindung in die Cloud schlagen wir die Brücke zwischen einem lokalen System und den vielseitigen Möglichkeiten der Cloud. Basierend auf der openHAB REST-API wird eine Smartphone App entwickelt, die dem Benutzer den Zugriff auf das System ermöglicht. Der Google GCM-Dienst benachrichtigt den Benutzer mit einer Push- Notification, sobald ein Alarm ausgelöst wurde.

Der Versuchsaufbau besteht aus einem Raspberry Pi als openHAB Server, zwei Philips-Hue Lampen, einem Fensterkontakt, einem Bewegungsmelder, sowie einer Webcam. Während der Entwicklung an der Android App entstand gleichzeitig ein wiederverwendbares SDK für openHAB Apps. Moderne Frameworks, reaktive Programmierung und ein ansprechendes Design runden die User Experience ab. Eine Azure Worker Role fungiert als MQTT-Client und persistiert den Event-Stream von openHAB sofort im Azure Table Storage. Zusammenfassend bietet unser System einen kostengünstigen Einbruchschutz für den Einstieg in die IoT-Smart-Home Welt.

Management Summary

Ausgangslage
Das «Internet of Things» ist eine Bezeichnung für ein Netzwerk aus intelligenten Gegenständen des Alltags. Viele dieser Things sind durch Herstellergrenzen technologisch von einander isoliert. Es gibt spezielle Softwareprodukte, damit eine heterogene Landschaft an Things zu einem Netzwerk zusammengeschlossen werden kann. Danach wird versucht eine kollektive Intelligenz im Internet of Things zu erreichen. In dieser Bachelorarbeit wollen wir nicht gleich alle Things der Welt vernetzen. Wir skalieren die Konzepte herunter und projizieren sie auf etwas vertrautes: Das eigene Zuhause – ein Smart-Home.

Vorgehen / Technologien
Nach einer Marktanalyse haben wir uns auf die Open Source Software «openHAB» festgelegt. OpenHAB definiert ein gemeinsames und abstraktes Modell eines Things und verbirgt die verwendete Technologie dahinter. Auf diesem Modell aufbauend können Regeln zur Interaktion zwischen Things beschreiben werden. OpenHAB wurde für die Verwendung in Privathaushalten entwickelt und kann keine beliebig grossen Netzwerke verwalten. Alle Geschehnisse im Haushalt werden sicher in der Cloud protokolliert.

Ergebnisse
Für das Beispielszenario «Einbruchschutz» haben wir einen Versuchsaufbau mit Things eingerichtet. Unsere Things sind ein Fensterkontakt, zwei Lampen, ein Bewegungsmelder und eine Webcam. Wir entwickelten eine Smartphone App, damit der Bewohner seine Things überwachen und steuern kann. Auf Wunsch benachrichtigt ihn das Smartphone, sobald sich ein Fenster unerwartet öffnet oder der Bewegungsmelder anschlägt. Durch unsere Cloud-Anwendung kann der Privathaushalt an einen Verbund weiterer Haushalte angeschlossen werden, womit sich das Internet of Things wieder nach oben skalieren lässt.

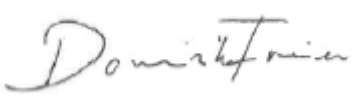
Erklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- das ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Ort, Datum:
Rapperswil, 12.06.2015

Marco Leutenegger



Dominik Freier

Danksagung

Zum Ende unserer Bachelorarbeit möchten wir uns bei allen beteiligten Personen herzlich bedanken. Speziellen Dank gebührt unserem Betreuer, Herrn Prof. Hansjörg Huser, der uns während der Arbeit begleitet hat und die Hardwarebeschaffung für den Systemaufbau ermöglicht hat.
Unser Dank gilt auch der openHAB Community und Kai Kreuzer, der dieses Projekt vorantreibt und weiterentwickelt.

Inhaltsverzeichnis

Abstract 2

Management Summary 3

Eigenständigkeitserklärung 4

Danksagung 5

Technischer Bericht 9

1. Ausgangslage . . . 9
2. Problembeschreibung . . . 10
   - 2.1. Motivation . . . 10
   - 2.2. Funktionale Anforderungen . . . 10
     - 2.2.1. Basisszenario . . . 10
     - 2.2.2. Lösungsteil (Demo-System) . . . 12
   - 2.3. Marktsituation . . . 13
3. Lösungskonzept . . . 15
   - 3.1. Evaluation der Plattform . . . 15
     - 3.1.1. Ergebnis: openHAB . . . 15
   - 3.2. Evaluation der Hardware . . . 16
   - 3.3. Einführung openHAB . . . 17
     - 3.3.1. Konnektivität . . . 17
     - 3.3.2. User Interface . . . 17
     - 3.3.3. Automatisierung . . . 18
     - 3.3.4. Persistenz . . . 18
   - 3.4. openHAB Architektur . . . 18
   - 3.5. OSGi . . . 19
     - 3.5.1. Event Bus . . . 20
     - 3.5.2. Bindings . . . 20
     - 3.5.3. Items . . . 21

3.5.4. Item Repository . . . 22
3.5.5. Rules . . . 22
3.5.6. Sitemaps . . . 22
3.6. Allgemeine Systemsicht . . . 23
3.6.1. Beispiel Use Case . . . 26
3.7. Android App . . . 27
3.7.1. Features . . . 27
3.7.2. Mockups . . . 28
3.7.3. Architektur . . . 31
3.8. Anbindung Cloud . . . 34
3.8.1. MQTT Broker Evaluation . . . 35
3.8.2. Client-Library . . . 36
3.9. Push-Notifications . . . 36
3.10. Sicherheit . . . 37
3.10.1. Cloud . . . 37
3.10.2. Android App . . . 38
3.10.3. Systemaufbau mit Sensoren/Aktoren . . . 38
3.10.4. HomeMatic Kommunikation . . . 38
3.10.5. Fazit . . . 38
4. Umsetzung . . . 39
4.1. openHAB Konfiguration . . . 39
4.1.1. Items . . . 40
4.1.2. Sitemap . . . 43
4.1.3. Rules . . . 44
4.2. MQTT . . . 48
4.2.1. Funktionsweise . . . 48
4.2.2. Broker . . . 49
4.2.3. openHAB MQTT Persistence (Publishing Client) . . . 51
4.2.4. Azure Worker Role (Subscribing Client) . . . 53
4.2.5. openHAB MQTT Action . . . 56
4.3. Android App . . . 59
4.3.1. Frameworks und Libraries . . . 59
4.3.2. Wichtige Klassen . . . 64
4.3.3. Material Design . . . 71
4.4. Push-Notifications . . . 71
4.4.1. Cloud . . . 72

4.4.2. Android-Client 74
4.5. Sicherheit 75
4.5.1. Verschlüsselung MQTT-Verbindung 75
4.5.2. Konfiguration MQTT-Broker 75
4.5.3. Verschlüsselung WLAN 76
4.6. Problematik Systemaufbau 76
5. Zusammenfassung und Ergebnisse 77
6. Persönliche Reflektion 78

Glossar **81**

A. Aufgabenstellung **82**

B. Installationsanleitungen **85**

Technischer Bericht

1. Ausgangslage

Diese Bachelorarbeit befasst sich mit einem Teilgebiet des «Internet of Things», nämlich der lokalen Vernetzung von Sensoren und Aktoren.

Gemäss der Aufgabenstellung soll eine Smart-Home Beispielapplikation erstellt werden, welche wesentliche Aspekte einer IoT-Anwendung demostriert. Das beinhaltet das Steuern von Aktoren, Lesen von Sensoren, Event-Verarbeitung, Überwachung und intelligente Abläufe steuern.
Das System soll auf einer tragbaren, erweiterbaren Architektur aufgebaut werden und Microsoft Azure als Cloud Plattform benutzen.

Die Heimautomation bzw. ein Smart-Home grenzt sich von der professionellen Gebäudeautomation in einigen Aspekten ab. Ein Smart-Home, wie wir es umsetzen, umfasst insgesamt deutlich weniger Sensoren und Aktoren, stellt dafür aber höhere Ansprüche an die Installierbarkeit, Bedienbarkeit und niedrige Anschaffungskosten. Unsere Arbeit soll zeigen, welche Überlegungen beim Einstieg in die Smart-Home-Welt angestellt werden müssen und auf was für Herausforderungen man dabei stösst.

Die Aufgabenstellung schlägt ein System zum Einbruchschutz als Beispielszenario vor.

Hinweis zur Struktur dieses Berichts:
Dieser Bericht basiert gemäss Absprache mit unserem Betreuer auf dem Stukturierungsbeispiel 2. Somit werden die Anforderungen an das System im Kapitel «Problembeschreibung», Architektur- und Design Überlegungen im Kapitel «Lösungskonzept» und Implementierungsdetails, sowie plattformabhängige Entscheidungen im Kapitel «Umsetzung» dokumentiert. Aus diesem Grund sind theoretische Überlegungen zumeist im Lösungskonzept zu finden, das praktische Pendant dazu hingegen im Umsetzungsteil.

2. Problembeschreibung

2.1. Motivation

Als SmartHome Beispielszenario soll ein System aufgesetzt werden, das einen grossen Bezug zur Realität hat. Es soll für aussenstehende Personen attraktiv und nachvollziehbar sein und einen Mehrwert mit sich bringen. In diesem Kapitel werden die funktionalen Anforderungen definiert und einige Fachbegriffe, sowie die Martksituation im Bereich Smart- Home erklärt. Der Abschnitt mit den funktionalen Anforderungen gliedert sich in ein Basisszenario und einen Lösungsteil. Das Basisszenario befasst sich mit den grundlegenden Konzepten und Anforderungen, im Lösungsteil gehen wir auf die konkreten Anforderungen für unser Beispielszenario «Einbruchschutz» ein.

2.2. Funktionale Anforderungen

2.2.1. Basisszenario

Dieser Abschnitt beschreibt die theoretischen Voraussetzungen, die für das Basisszenario gegeben sein müssen, unabhängig von der späteren Implementierung der konkreten Anwendungsfälle. In der Gebäudeautomation spricht man häufig von sogenannten Sensoren und Aktoren. Beide Begriffe stammen ursprünglich aus der Steuerungs- und Regelungstechnik. Sensoren sind Geräte, die kontinuierlich Messdaten erfassen und die Daten über eine Schnittstelle verfügbar machen. Ob die Daten aktiv vom Sensor an eine Zentrale übermittelt werden, oder ob die Zentrale den Sensor selbstständig abfragen muss, ist für die Definition eines Sensors in unserem Rahmen nebensächlich. Ein Aktor kann als Gegenstück zum Sensor betrachtet werden, da er eine aktive Rolle einnimmt. Ein Aktor empfängt Befehle und greift in das Regelungssystem ein. Einige Geräte beinhalten sowohl Sensor- als auch Aktorschnittstellen. Wenn ein Sensor eine Zustandsänderung registriert und diese dem System bekannt gibt, dann sprechen wir von einem *Event*. Als *Command* bezeichnen wir einen Befehl, der an einen Aktor gesendet wird.

Beispiele für Sensoren aus der Gebäudeautomation:

- Bewegungsmelder
- Thermometer
- Fensterkontakte
- Windkraftmesser

Beispiele für Aktoren:

- Lampe
- Heizungsregler
- Rasensprinkler
- Rollläden

Kombinierte Geräte:

- Überwachungskamera (Sensor: Bilder, Aktor: Schwenken)
- Moderner Backofen (Sensor: Temperatur, Aktor: Programmauswahl)

Für das Basisszenario benötigen wir eine Infrastruktur, über die wir alle angeschlossenen Sensoren und Aktoren erreichen können. Konkret ausgedrückt müssen wir in der Lage sein, Events von Sensoren zu empfangen und Commands an Aktoren zu senden. Darüber hinaus sollen möglichst viele Vorgänge automatisiert werden. Das bedeutet, wir benötigen eine Zentrale, um Events zu verarbeiten und gegebenenfalls mit Befehlen an Aktoren auf diese Events zu reagieren. Neben Aktoren und Sensoren existieren noch Clients. Clients sind Geräte, die von einem Benutzer des Systems verwendet werden, um Sensoren abzufragen und Commands an Aktoren zu senden. Clients greifen in der Regel nicht direkt auf Sensoren und Aktoren zu, sondern benutzen zu diesem Zweck die Zentrale. Das Netz, über das alle Sensoren, Aktoren, Clients und die Zentrale miteinander verbunden sind, Bezeichnen wir von nun an als *Bus*.

F01: Sensoren Status
Der Status eines Sensors kann über den Bus abgefragt werden. Sensoren können auch selbstständig Events auf den Bus senden.

F02: Steuern von Aktoren
Jede Komponente, die am Bus angeschlossen ist, kann Commands auf den Bus schicken. Aktoren können die Commands über den Bus empfangen und dadurch gesteuert werden.

F03: Persistieren von Events und Commands in der Cloud
Alle Events und Commands, die auf dem Bus transportiert werden, können in der Cloud gespeichert werden. In der Cloud können anhand dieser Daten umfassende Analysen angestellt werden (nicht Teil der Arbeit).

F04: Regeln
Aufgrund von Events sollen je nach Zustand des Gesamtsystems vordefinierte Aktionen ausgelöst werden. Die Aktionen bestehen in der Regel aus Commands, um wiederum Aktoren zu steuern.

F05: Auf Zentrale zugreifen
Ein Client kann auf die Zentrale zugreifen, um den Status des Systems abzufragen und Commands an Aktoren zu schicken.

2.2.2. Lösungsteil (Demo-System)

L01: Sicherheits-Status abfragen
Der Client soll den Status des Einbruchschutz abfragen können.
Status «OK»:

- Fenster ist geschlossen
- keine Bewegung detektiert

Status «NOK»:

- Fenster ist offen
- Bewegung detektiert

L02: Überwachungskamera
Der Client kann die Überwachungskamera ein- bzw. ausschalten und Livebilder anfordern.

L03: Event Kontaktsensor
Der Kontaktsensor hat permanent einen Status. Der Status ist entweder «offen» oder «geschlossen».

L04: Event Bewegungsmelder
Sobald der Bewegungsmelder eine Bewegung registriert, sendet dieser einen Event. Dieser wird nach interner Logik verarbeitet.

L05: Aktor Lampe
Der Aktor wird via Command angesteuert. Das Licht kann durch eine Regel (Zeit-

Mechanismus zur Prävention) oder durch eine Aktion des Clients ein- bzw. abgeschaltet werden.

L06: NFC Sticker
Die NFC Stickers können sehr vielfältig eingesetzt werden und dienen in erster Linie zur Entlastung des User Interface. Generell wird durch Auflegen eines NFC-fähigen Smartphones (also ein Client) eine Aktion ausgeführt. Was diese Aktion genau beinhaltet, ist offen und könnte genauso gut direkt im User Interface benutzt werden. Beispielsweise könnte ein Command zum «Scharfstellen» des Sicherheitssystems auf den Bus gesendet werden.

2.3. Marktsituation

Das Thema IoT, insbesondere SmartHome, erlebt zurzeit einen regelrechten Boom. Zwar gibt es schon seit Jahrzehnten Lösungen zur Heimautomatisierung, jedoch standen auch einfache Systeme bisher nur wenigen Privilegierten zur Verfügung, denn meistens musste man sich bereits beim Hausbau auf einen Anbieter festlegen und dessen Produkt- und Preispolitik akzeptieren. Folglich ist es aufwändig, den Anbieter nachträglich zu wechseln oder das System zu erweitern. Als Vorteile von klassischen Gesamtlösungen sind jedoch das einheitliche Bild und die nahtlose bauliche Integration zu nennen.

KNX
KNX ist ein europäischer Standard für kommerzielle Gebäudeautomation. KNX trennt Gerätesteuerung und Stromversorgung in von einander unabhängige Netze. Mit KNX können Schalter und Steuerungen relativ einfach neu zugewiesen werden, ohne dass erneut bauliche Arbeiten vorgenommen werden müssen. KNX bringt hohe Anschaffungskosten mit sich und eignet sich eher für Neubauten, da eine nachträgliche Installation noch viel teurer wäre.

digitalSTROM
Ursprünglich ein Projekt der ETH, das die nachträgliche Gebäudeautomation ermöglichen soll. Durch den Einsatz von mit einander kommunizierenden, speziellen Kabelklemmen lassen sich vorallem Beleuchtungskonzepte und Energiesparlösungen recht schnell und einfach realisieren. Das Absetzen von Befehlen oder Auslesen von Messdaten aus unterschiedlichen Protokollen ist kaum möglich. Leider sind die vielen benötigten Komponenten immer noch sehr teuer (ca. 100.- CHF für eine Klemme).

RWE Smarthome

Der Energieversorgungskonzern RWE bietet seit einigen Jahren ein nachrüstbares System für die Heimautomation. Der Fokus liegt vorallem auf der Automatisierung von Beleuchtung, Heizung, Rollläden und Stromversorgung. RWE bietet eine Zentrale, Sensoren, Aktoren sowie Software zur Steuerung und Konfiguration. Die Auswahl an Komponenten ist auf das Angebot von RWE beschränkt.

Qivicon

Qivicon ist eine Allianz verschiedener Industriepartner und wurde von der Deutschen Telekom gegründet. Im Unterschied zum RWE Smarthome können also Komponenten aller beteiligten Hersteller miteinander vernetzt werden. Die Bedienung ist etwas einfacher, dafür sind die Konfigurationsmöglichkeiten weniger umfassend als beim RWE Smarthome.

openHAB

OpenHAB steht für *open Home Automation Bus* und ist ein Open Source Projekt zur lokalen Vernetzung von IoT- und Smart-Home Zubehör unterschiedlichster Hersteller. OpenHAB fungiert als Zentrale und kann auf beliebigen Windows, Linux oder Mac OS X Geräten installiert werden. Durch die modulare Architektur können jederzeit neue Technologien integriert werden. Eine eigene Modellierungssprache erlaubt nahezu unbegrenzte Konfigurationsmöglichkeiten anhand von Regeln und Abläufen. Allerdings erfordert dies ein wenig technisches Geschick. Basierend auf openHAB 1.x wurde das Eclipse Smarthome Projekt gegründet, welches ein Framework für Smart-Home Software darstellt. OpenHAB 2 wiederum baut auf Eclipse Smarthome auf und soll die Konfiguration, insbesondere für technisch weniger versierte Benutzer, erleichtern.

3. Lösungskonzept

Dieses Kapitel beinhaltet die Beschreibung der Architektur und wichtiger Komponenten. Es werden die theoretischen Grundlagen der eingesetzten Technologien und Systeme erläutert und warum diese verwendet werden. Nicht Teil dieses Kapitels sind Erläuterungen von Konfigurationen und Code. Diese werden separat im Kapitel «Umsetzung» ab Seite 39 behandelt.

3.1. Evaluation der Plattform

Anhand der Marktsituation und den funktionalen Anforderungen haben wir die Vor- und Nachteile der jeweiligen Plattformen miteinander verglichen und darauf geachtet, welche Kriterien für unsere Lösung von Bedeutung sind.

Wichtige Kriterien:

- Nachträgliche Installation möglich
- Beliebige Szenarien realisierbar
- Herstellerunabhängige Komponenten
- Erfüllung der Anforderungen F01 - F05

Vernachlässigbare Kriterien:

- Optisch ansprechende Integration
- Installation ohne Fachkenntnisse

3.1.1. Ergebnis: openHAB

Mit openHAB haben wir eine Plattform gefunden, die allen wichtigen Kriterien entspricht und zudem kostenlos ist. Da wir openHAB sofort auf unseren Notebooks installieren konnten, war es sehr einfach zu beurteilen, ob die Plattform auch in der Praxis unsere Anforderungen erfüllt. Die mitgelieferte Demo-Konfiguration beinhaltete bereits viele anschauliche Beispiele, die später als Vorlage für unsere eigenen Anwendungsfälle dienen können.

Erfüllung der funktionalen Anforderungen
F01 - F02: Über sogenannte Items können Sensoren und Aktoren virtuell und genügend

abstrakt definiert werden. Der OSGi EventBus von openHAB ermöglicht den Transport von Events und Commands zwischen Items und der Zentrale (openHAB Runtime). Bindings mappen die Items auf tatsächliche Sensoren und Aktoren.

F03: OpenHAB kann den Verkehr auf dem EventBus über verschiedene Wege auf externen Systemen protokollieren. Zu unserem Zweck eignet sich das MQTT Persistence Modul.

F04: Über die Rule Engine von openHAB können Regeln mit Hilfe einer Java-ähnlichen DSL beschrieben werden. Regeln werden bei gewissen Events auf dem EventBus ausgeführt. Die DLS erlaubt den Zugriff auf den Zustand von Items und kann auch Commands an Items und somit an Aktoren senden.

F05: Eine REST-API bietet umfassenden Zugriff auf die openHAB Runtime. Über sogenannte Sitemaps können deskriptive User Interfaces automatisch generiert werden.

Nachteile
Ein Nachteil von openHAB ist, dass die Dokumentation grosse Lücken aufweist. Zwar sind die Konzepte leicht verständlich, jedoch fehlen Detailangaben zur DSL und genauen Konfigurationssyntax. Aus diesem Grund müssen oft Beispiele analysiert oder Benutzerforen zu Rate gezogen werden.

3.2. Evaluation der Hardware

Nachdem wir openHAB als Plattform bestimmt haben, konnten wir die Hardware für die Sensoren und Aktoren aussuchen. Dafür haben wir uns an den Vorgaben L02 - L05 aus Abschnitt 2.2.2 auf Seite 12 orientiert. Durch die Vielzahl an Protokollen, die durch openHAB unterstützt werden, hatten wir genügend Auswahl an Hardware von unterschiedlichen Herstellern. OpenHAB selbst läuft auf einem Raspberry Pi B+.

L02: Als Überwachungskamera haben wir die Edimax IC-3115W Netzwerkkamera ausgesucht. Sie ist mit einem Preis von weniger als 50 Euro relativ günstig und über das HTTP Binding von openHAB kompatibel.

L03: Beim Fensterkontaktsensor war uns ein kabelloses Modell wichtig, das unkompliziert montiert werden kann. Aus diesem Grund haben wir uns für den optischen Fensterkontakt HM-Sec-Sco von eQ-3 HomeMatic entschieden. Der Fensterkontakt erfordert jedoch eine Zentrale, die separat bestellt werden muss. Für HomeMatic existiert ein Binding seitens

openHAB.

L04: Der Funk Bewegungsmelder HM-Sec-MDIR-2, ebenfalls von eQ-3 HomeMatic, benutzt die gleiche Zentrale wie der Fensterkontaktsensor und ist für den Indoorgebrauch ausgelegt.

L05: Das Philips Hue Lux Starterkit beinhaltet zwei dimmbare LED-Birnen und eine Zentrale, die ans lokale Netzwerk angeschlossen werden muss. Ein openHAB Binding für Philips Hue ist vorhanden.

3.3. Einführung openHAB

Das System openHAB wird eingesetzt, um verschiedene Home-Automatisierungssysteme unter einen Hut zu bringen. Um dies zu erreichen, müssen Lösungen für die vier Disziplinen Konnektivität, User Interface, Automatisierung und Persistenz gefunden werden. In den nächsten Abschnitten werden diese Disziplinen kurz beschrieben.

3.3.1. Konnektivität

Mit Konnektivität ist gemeint, wie die Sensoren/Aktoren integriert werden können. Es braucht ein Konzept, um Protokolle miteinander kompatibel zu machen. Nehmen wir als Beispiel den Use Case *L03: Event Kontaktsensor*. An einem Fenster wird ein Kontaktsensor angebracht, der bei jedem Öffnen oder Schliessen den Status bekannt gibt. OpenHAB muss einerseits das verwendete Protokoll verstehen und zudem die Daten in eine interne, abstrakte Form übersetzen, sodass herstellerspezifische Details vor dem restlichen System verborgen bleiben. Ein weiteres Beispiel ist Use Case *L05: Aktor: Lampe*. Hierbei müssen keine Events gelesen, sondern Commands geschickt werden, da es sich bei der Lampe um einen Aktor handelt. Dazu muss openHAB auch den umgekehrten Fall beherrschen, nämlich aus einer internen Repräsentation des Commands in diejenige des Protokolls der Lampe zu übersetzen und letztlich auch die Lampe erreichen können. Durch Konnektivität ist es also möglich, Smart-Home Zubehör von verschiedenen Herstellern in openHAB einzubinden.

3.3.2. User Interface

Nehmen wir an, der Fensterkontakt und die Lampe aus dem vorherigen Abschnitt sind von zwei völlig verschiedenen Herstellern. Der Status des Fensterkontakts soll bei der

Verwendung ohne openHAB über eine Website im Browser ausgelesen werden können. Das Steuern der Lampe geschehe mittels einer eigens dafür vorgesehenen App. Dank dem Konzept der Konnektivität können auf beide Geräte auch über openHAB zugegriffen werden. Einen echten Vorteil hat man dadurch aber nur, wenn es auch ein User Interface dazu gibt. Denn dann hat man alle Geräte in einer Smartphone- oder Web App vereint. OpenHAB benötigt demnach eine Möglichkeit, um User Interfaces für verschiedene Clients zu gestalten.

3.3.3. Automatisierung

Durch die Konnektivität und das User Interface kann also Smart-Home Zubehör von verschiedenen Herstellern in einer einzigen Anwendung verwendet werden. Das alleine ist schon ein grosser Mehrwert. Doch es fehlt noch etwas der smarte Teil des Smart-Homes. Interessant wird es nämlich dann, wenn die verschiedenen Geräte sich gegenseitig beeinflussen sollen. Nehmen wir den Bewegungsmelder aus Use Case L04 hinzu. Sobald er eine Bewegung registriert, soll die Lampe eingeschaltet werden. Es sind aber auch wesentlich komplexere Szenarien denkbar. Damit solche automatisierten Vorgänge stattfinden können, benötigt openHAB eine Rule Engine. Die Grundlage dazu bildet die interne Repräsentation der Sensoren und Aktoren, die bereits durch die Konzepte zur Konnektivität geschaffen wurde.

3.3.4. Persistenz

Wenn Events gelesen und Commands gesendet werden, dann handelt es sich dabei um Momentaufnahmen. Im User Interface könnte man beobachten, wenn der Status des Fensterkontakts von «offen» auf «zu» wechselt. Doch was ist, wenn man wissen möchte, wann das Fenster zuletzt geöffnet wurde? Aus diesem Grund reicht es nicht, Events lediglich zu verarbeiten, sondern sie müssen auch persistiert werden. Zudem können manche Sensoren wie der Fensterkontakt möglicherweise nur immer einen Statuswechsel bekanntgeben, der aktuelle Status kann aber nicht direkt abgefragt werden. Damit trotzdem jederzeit der aktuelle Status bekannt ist, muss openHAB den Status bei jedem Wechsel speichern.

3.4. openHAB Architektur

Im Abschnitt zur Konnektivität haben wir bereits erläutert, welche Anforderungen openHAB erfüllen muss, damit verschiedene Systeme miteinander vernetzt werden können.

Die grosse Anzahl an Herstellern und die Vielfalt an Protokollen haben dazu geführt, dass openHAB sehr modular konzipiert wurde. Die Basisinstallation kann zur Laufzeit durch Add-ons erweitert werden. Das hat den Vorteil, dass openHAB selbst recht schlank bleibt und Technologien, die gar nicht eingesetzt werden, nicht im Weg sind. Ausserdem ist dadurch das spätere Einbinden weiterer Plattformen sehr einfach machbar. Technisch wurde diese modulare Architektur mit Hilfe der OSGi-Plattform umgesetzt. Die Implementierung von Protokollen geschieht innerhalb von OSGi Service Bundles, die bei openHAB Bindings genannt werden. Abbildung 1 zeigt einen Überblick der Architektur:

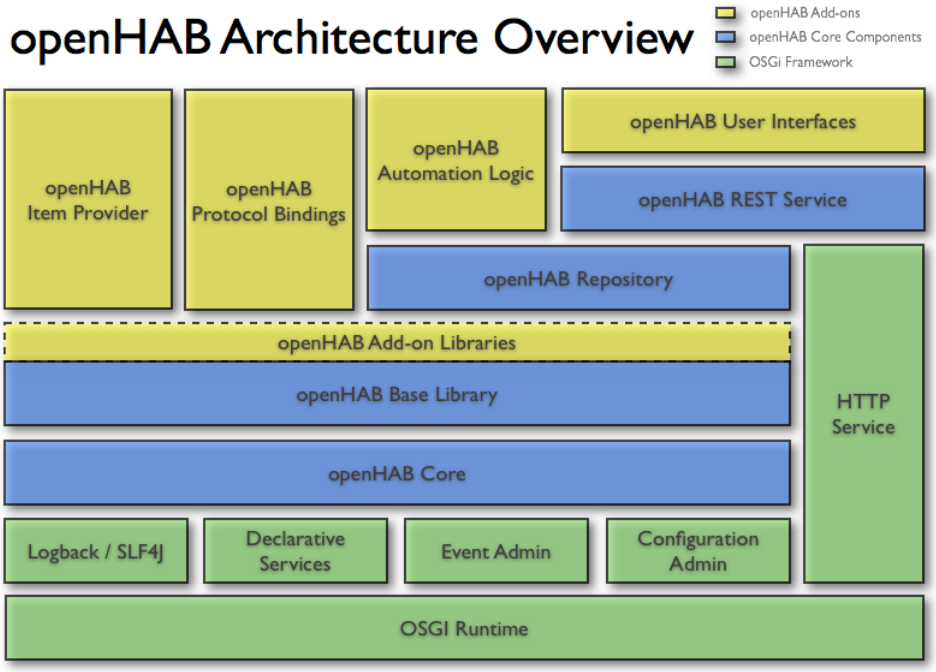


Abbildung 1.: openHAB Architektur, Quelle: `http://www.openhab.org`

3.5. OSGi

OSGi ist ein offener Standard, der eine modular aufgebaute Platform beschreibt. Es wird lediglich eine JVM (Java Virual Machine) vorausgesetzt. OSGi wird oft im Bereich der Gebäudeautomation eingesetzt, denn ein grosser Vorteil dieser Platform ist die Möglichkeit, einzelne Komponente zur Laufzeit einzubinden. Durch den modularen Aufbau ist es möglich, verschiedene Kompnenten miteinander zu verbinden, oder sie parallel und unabhängig von einander laufen zu lassen.

3.5.1. Event Bus

Der Event Bus stellt den Basisservice von openHAB dar. Über diesen Bus werden Events zwischen den verschiedenen OSGi Bundles gesendet. Die Events sind entweder Commands, welche eine Aktion ausführen, oder Status-Updates, welche Zustandsänderungen der Sensoren/Aktoren beinhalten.
Durch den Einsatz dieses Event Bus wird die Kopplung reduziert und Add-ons können somit einfach ausgetauscht werden.
Intern wurde der Event Bus mit Hilfe des OSGi Event Admin umgesetzt. Durch den Event Admin wird es möglich, dass sich weitere OSGi Bundles (also openHAB Add-ons) im Event Bus einklinken können.

3.5.2. Bindings

Bindings sind Verbindungen zwischen openHAB und den externen Systemen und bilden die Grundlage zur Konnektivität. Dadurch muss für jede Technologie ein eigenes Binding geschrieben werden. Für viele Technologien sind Bindings vorhanden, die einzeln heruntergeladen und als «Add-on» installiert werden können. Falls eine Technologie noch nicht unterstützt wird, kann man das Binding dazu selbst programmieren. Die wesentliche Aufgabe besteht darin, sich einerseits mit dem externen Gerät zu verbinden, auf dem Event Bus zu lauschen und die ausgetauschten Daten miteinander kompatibel zu machen.
Alle momentan verfügbaren Bindings sind unter folgendem Link zu finden: `https://github.com/openhab/openhab/wiki/Bindings`

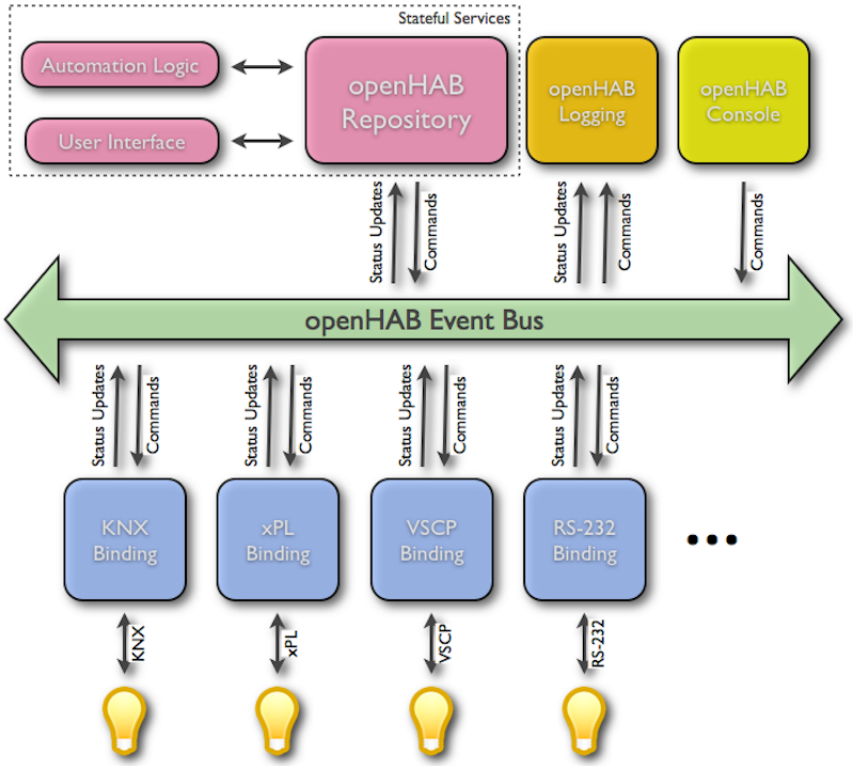


Abbildung 2.: Event Bus als Schnittstelle für Bindings, Quelle: `http://www.openhab.org`

3.5.3. Items

Wenn so viele unterschiedliche Technologien unterstützt und integriert werden sollen, dann stellt sich die Frage nach dem gemeinsamen Nenner bzw. einer einheitlichen internen Repräsentation. Aus diesem Grund wurden «Items» eingeführt, die zentrale Entität im openHAB Domainmodell. Alle Bindings implementieren ein Mapping zwischen den Daten des Sensors/Aktors und einem zugehörigen Item. Ein Item besteht aus:

- Typ
- Name
- Formatierung
- Icon
- Gruppe
- Bindingparameter

Die Eigenschaften Typ und Name sind zwingend, die Anderen sind optional. Der Typ ist auf eine vorgegebene Auswahl beschränkt, der Name dient als Identifier und muss eindeutig sein.

Je nach gewähltem Typ können Items die unterschiedlichsten Dinge repräsentieren. Ein sehr häufig verwendeter Typ ist das SwitchItem mit den beiden möglichen Werten ON und OFF. Dieses Item eignet sich aufgrund seines Wertebereichs hervorragend als Boolean. Ein SwitchItem kann stellvertretend für etwas reales, wie eine Lampe, oder für etwas abstraktes wie die Anwesenheit eines Bewohners stehen. Es ist wichtig zu verstehen, dass ein Item rein virtuell ist. Die Brücke zur echten Hardware wird erst über Bindingparameter geschlagen, denn dann verknüpft openHAB das Item mit dem entsprechenden Binding. Da die Bindingparameter eines Items aber optional sind macht openHAB keinen Unterschied zwischen virtuellen und realen Items.

3.5.4. Item Repository

Der Zustand eines Items muss die Ausführungsdauer einer openHAB Instanz überleben können, beispielsweise nach einem Neustart. Der Zustand muss ausserdem jederzeit abfragbar sein, egal ob sich der Zustand des Items gerade erst durch ein Event geändert hat oder schon über mehrere Startvorgänge hinweg genau so exisitiert. Diese Verantwortung übernimmt das Item Repository, welches permanent den Event Bus auf Status-Updates abhört und die Änderungen persistiert.

3.5.5. Rules

Rules sind die Werkzeuge zur Automatisierung. Dank einer Java-ähnlichen DSL können Aktionen ausgelöst werden, wenn gewisse Bedingungen erfüllt wurden. Das können einerseits Zustandsänderungen von Items, aber auch Systemevents oder Events von selbst angelegten Timern sein.

3.5.6. Sitemaps

Sitemaps sind Konfigurationsfiles zur deklarativen Definition von User Interfaces. Sitemaps werden ebenfalls in einer Xtext basierten DSL geschrieben und haben eine baumartige Struktur. OpenHAB liest das Konfigurationsfile und stellt die Sitemap über ein REST-API zur Verfügung. Eine mitgelieferte Webapplikation stellt das User Interface

anschliessend im Browser dar. Layouttechnisch orientiert sich die Webapplikation am iOS 6 Design mit Listendarstellung und Drill-Down Prinzip. Die nativen iOS und Android Apps von openHAB verfolgen das selbe Bedienkonzept.
Die verschiedenen Elemente, die in Sitemaps verfügbar sind, können Items direkt über deren Name referenzieren. Einige Elemente sind dynamisch und erlauben eine werteabhängige Darstellung.

3.6. Allgemeine Systemsicht

Anhand der Problembeschreibung wurde ein Plan erarbeitet, der das ganze System verständlich beschreibt. Die Abbildung 3 stellt die wichtigsten Komponenten und Entitäten aus der Problemdomäne in gegenseitiger Beziehung dar. An zentraler Stelle steht das openHAB-Framework, es verbindet alle Komponenten. Die Cloud wird über MQTT (ein Message-Protokoll) angebunden, um Events zu persistieren.
Die Sensoren/Aktoren kommunizieren durch Bindings über das Netzwerk mit openHAB um deren Status zu übertragen. Der Client kann diesen Status über ein REST-API abfragen und Commands senden.

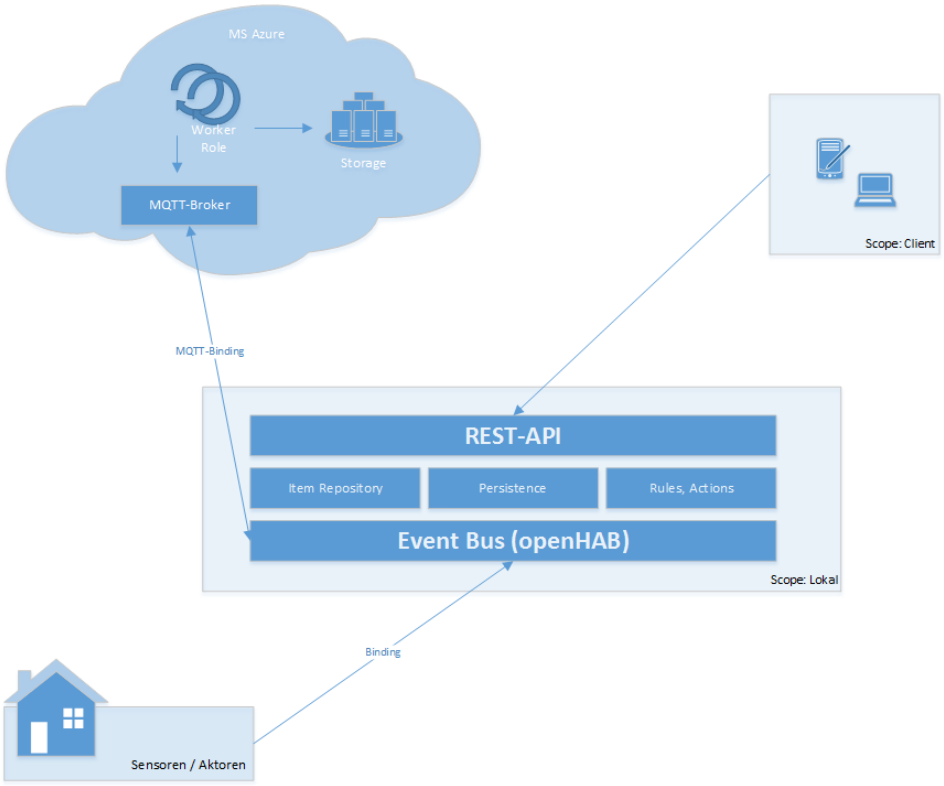


Abbildung 3.: Systemübersicht

Für das Szenario wurde eine fiktive Wohnung (Abbildung 4) gezeichnet, um den Einsatz in einer Zwei-Zimmer-Wohnung zu demonstrieren. Dabei befindet sich das openHAB auf dem Rasperry Pi in zentraler Stelle, damit die anderen Zentralen über kurze Strecken mit LAN-Kabeln vernetzt werden können.
Die Webcam ist so platziert, damit der grosse Wohnbereich mitsamt Küche überwacht werden kann und somit ein potentieller Einbrecher aufgenommen werden kann. Der Fenster Kontaktsensor wird im Zimmer angebracht und der Bewegungsmelder zwischen Küche und Wohnzimmer, um einen möglichst grosse Fläche abzudecken.Die Philips Hue Lampen befinden sich im Schlaf- und Wohnzimmer, um die ganze Wohnung zu beleuchten.

Die Türen und das Fenster im Wohnzimmer werden nicht mit Kontaktsensoren ausgestattet, da dieser Bereich durch den Bewegungsmelder erfasst wird.

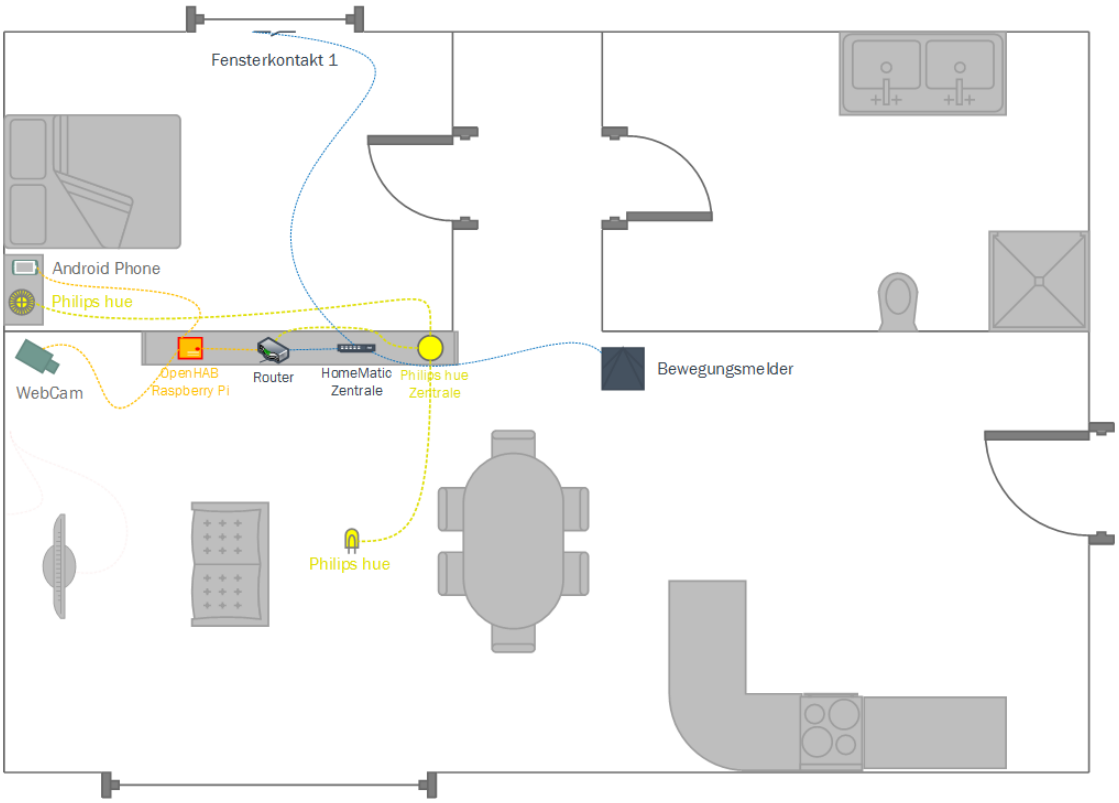


Abbildung 4.: Hausplan

Philips Hue und HomeMatic besitzen beide eine Zentrale, über die kommunizieren sie mit Ihren Gadgets. Die Zentralen werden über LAN (RJ45) an das Raspberry Pi angeschlossen.

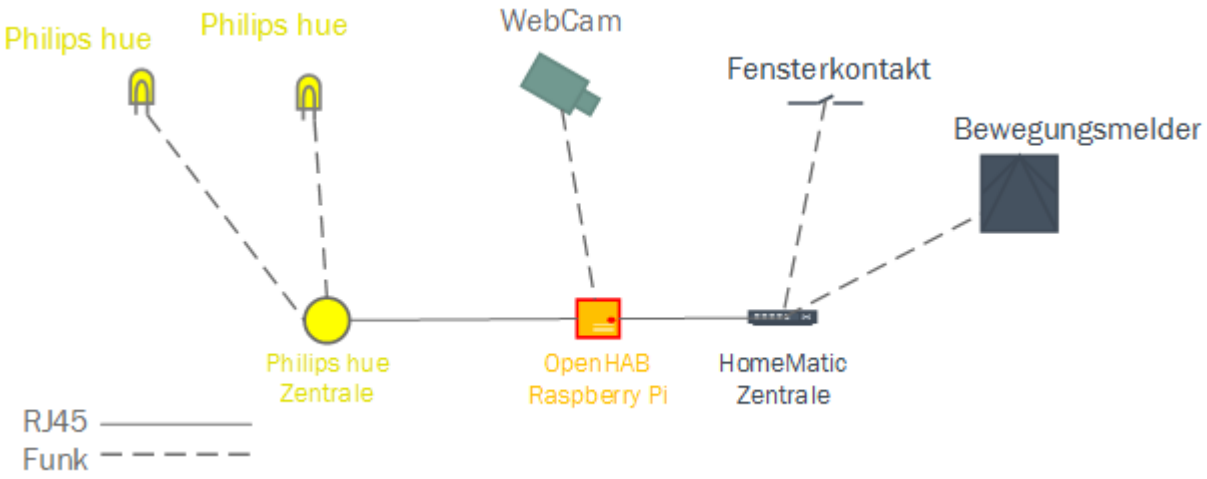


Abbildung 5.: Vernetzung der Komponenten

3.6.1. Beispiel Use Case

Das System verfügt über zwei Alarm-Modi: Scharf und Inaktiv. Wenn der Alarm scharf ist und der HomeMatic Bewegungsmelder oder der Fensterkontakt eine Veränderung registriert, dann sollen die Philips Hue Lampen eingeschaltet werden. Zeitgleich werden Aufnahmen der Webcam an die Cloud gesendet. Die Cloud schickt eine Push-Notification an das Android App um den Benutzer über den ausgelösten Alarm zu informieren. Das folgende Sequenzdiagramm veranschaulicht den Use Case:

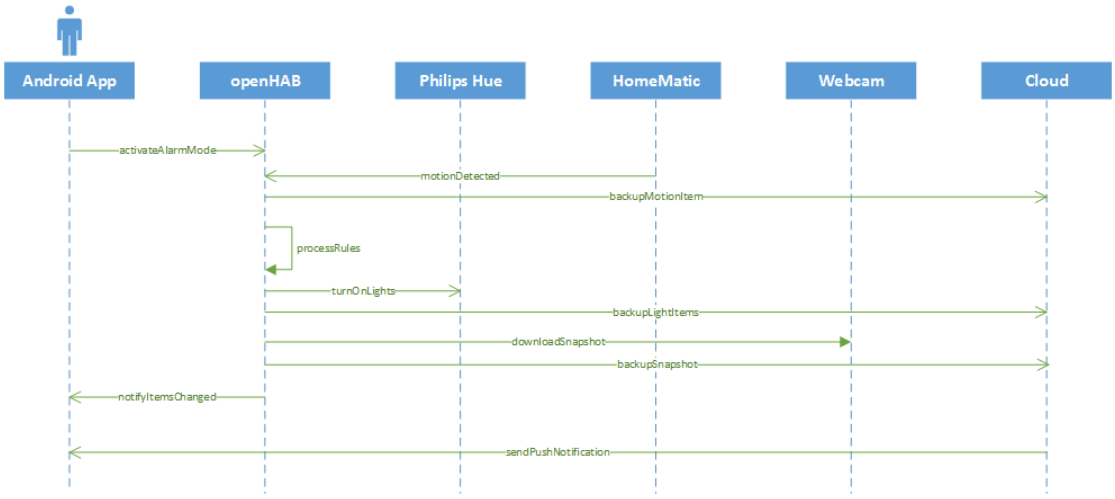


Abbildung 6.: Sequenzdiagramm Alarm

3.7. Android App

Als User Interface für unser System dient eine Android App. Zwar existieren bereits verschiedene openHAB Clients, sowohl als Web-App als auch als Android App, jedoch sind diese zu generisch. Man könnte argumentieren, dass es doch gut ist, wenn die App universell einsetzbar ist und oft hätte man damit recht. In unserem speziellen Szenario «Einbruchschutz» müssten wir aber zu viele Kompromisse eingehen. Deshalb haben wir uns dazu entschieden, die Android App selbst zu programmieren.

Jede openHAB Installation bietet eine offene REST-Schnittstelle an, damit auch Clients von Drittanbietern auf Items und Sitemaps zugreifen können. Der openHAB Server ermöglicht sowohl das explizite Abfragen der Daten, als auch den umgekehrten Weg um Clients proaktiv über Änderungen zu informieren.

3.7.1. Features

Die Features unserer App orientieren sich an den funktionalen Anforderungen zum Lösungsteil. Insbesondere sind dies:

- Übersichtsseite für den Status des Hauses
- Überwachung der Fenster
- Funktion zum Ein- und Ausschalten der Lampen

- Scharfstellen oder Deaktivieren des Alarm-Modus via NFC
- Ansicht der Überwachungskamera
- Status des Bewegungsmelders abfragen

3.7.2. Mockups

In dieser Ansicht soll der Benutzer möglichst auf einen Blick alle relevanten Informationen zum Sicherheitszustand seines Zuhauses erhalten. Von hier aus gelangt man zu den Bereichen «Fenster», «Licht» und «Überwachung». Für alle Bereiche wird eine kleine Zusammenfassung zum Status angezeigt.

Abbildung 7.: Übersichtsseite

Auf der Detailseite «Fenster» können die entsprechend ausgerüsteten Fenster überwacht werden. In unserem Versuchsaufbau ist nur ein Konaktsensor vorgesehen, sollten es aber mehrere sein, würden Fenster untereinander aufgelistet.

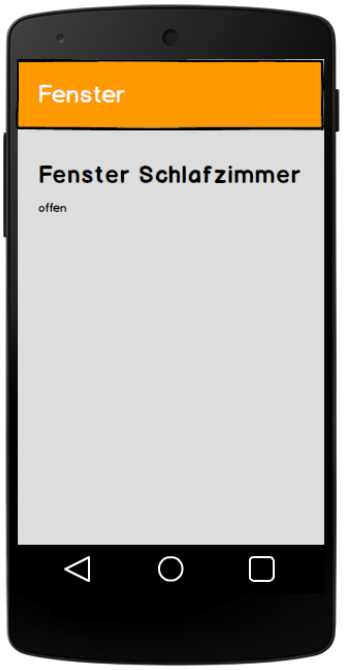


Abbildung 8.: Fenster

Auf dieser Seite sind alle Lampen aufgelistet, die in openHAB integriert wurden. Mit einem Switch können diese ein- oder ausgeschaltet werden. Auf eine Funktion zum Dimmen der Lampen wird verzichtet.

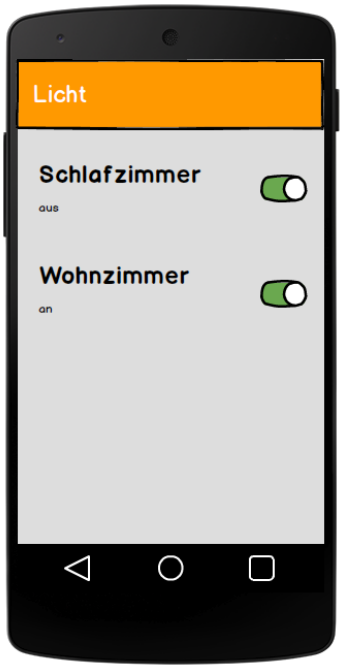


Abbildung 9.: Licht

Diese Ansicht enthält einen Schalter zum Scharfstellen oder Deaktivieren des Alarm-Modus, den Status des Bewegungsmelders und das Bild der Überwachungskamera.

Abbildung 10.: Überwachung

Wenn das Smartphone auf einen vorkonfigurierten NFC-Tag gelegt wird erscheint diese Ansicht, auch wenn die App zuvor nicht geöffnet war. Da der Alarm-Modus nur die beiden Zustände «Scharf» oder «Inaktiv» kennt, wird der Modus bei jedem Auflegen jeweils umgekehrt.

Abbildung 11.: NFC-Tag

3.7.3. Architektur

Die Architektur von Android Apps sieht meistens relativ ähnlich aus. Sie bestehen aus einem Manifest-File, Activities, BroadcastReceiver, ContentProvider und Services. Alle diese Komponenten sind als abstrakte Klassen im Android Framework enthalten. Wir verwenden in unserer App Subklassen von Activity, BroadcastReceiver und Service.

Manifest-File
In diesem XML-File werden die Rahmenbedingungen der App festgehalten. Beispielsweise welche Berechtigungen die App benötigt, welche Activities und Services sie anbietet oder verwendet.

Activities
Alle Ansichtsseiten einer Android App werden mit Hilfe von Activities umgesetzt und erben von der Framework-Klasse Activity. Eine Activity besitzt ein zugehöriges Layout (in Form eines XML-Files) und ermöglicht die Manipulation der Layoutelemente. Dazu gehört auch das Verarbeiten von Events, die von Layoutelementen gesendet wurden. Jede Ansichtsseite hat also in der Regel seine eigene Activity, die an einen Lebenszyklus gebunden ist. Das Android OS ruft bei jeder Phase des Lebenszyklus (onCreate, onResume, onPause, onDestroy etc.) die jeweilige Methode der Activity auf. Diese Methoden müssen in den eigenen Activity Subklassen überschrieben und korrekt implementiert werden. Gestartet werden Activities nicht durch Instanziierung mit dem new-Keyword, sondern über einen Intent (englisches Wort für «Absicht»).

Im Android Framework wird die Klasse Intent verwendet, um seine Absicht zum Starten einer Activity zu erklären. Dazu wird ein Intent-Objekt instanziiert und mit Key/Value Wertepaaren als Parameter für die Activity befüllt. Der Value eines Parameter kann kein beliebier Typ sein, sondern muss gewisse Kriterien erfüllen. Die primitiven Typen gehören dazu. Für Activities der selben App kann einfach die Klassenreferenz (Activity.class) beim Intent übergeben werden und Android übernimmt die Instanziierung. Im Manifest-File kann eine App seine Activities einzeln nach aussen anbieten, z.B. eine Activity zum Aufnehmen eines Fotos. Andere Apps können mit einem allgemein formulierten Intent diese Activity starten. Sollte es mehrere Apps geben, die Activities zur Fotoaufnahme anbieten, kann der Benutzer eine App aus der Liste auswählen und als Standard speichern.

Activties können ausschliesslich durch einen Intent erzeugt werden, deshalb können einer Activity keine Konstruktorargumente und dadurch auch keine Referenzen auf komplexe

Domainmodel-Objekte (View-Models, Service-Klassen) oder ähnliches übergeben werden. Alle Parameter sind serialisierte Objekte (meist Strings), die in einer Key/Value Datenstruktur über den Intent zugreifbar sind. Oft möchte man aber in mehreren Activities auf die gleichen Domainmodell-Objekte zugreifen können. Zur Lösung des Problems kann man Dependency Injection Frameworks einsetzen, die so konfiguriert werden, dass die entsprechenden Objekte automatisch den annotierten Instanzvariablen zugewiesen werden. Das lohnt sich aber nur für grössere Apps, da ein gewisser Initialaufwand damit verbunden ist. Ansonsten bleibt nur die Verwendung von Singletons bzw. statischen Methoden übrig. Streng genommen benutzen auch Dependency Injection Frameworks diese Variante.

BroadcastReceiver
Werden benötigt um auf Ereignisse von Android oder von anderen Apps zu reagieren. Sie sind Teil der App und werden im Manifest-File deklariert. Die Implementierung erfolgt durch Erben der Klasse BroadcastReceiver. Wir verwenden in unserer App einen BroadcastReceiver um Push-Notifications empfangen zu können.

Services
Sind Android-Klassen zum Ausführen von Code im Hintergrund, ohne dass eine Activity für den Benutzer sichtbar ist. Wir verwenden einen Service, um die Push-Notification, die vom BroadcastReceiver empfangen wurde, zu verarbeiten. Nicht zu verwechseln sind diese (Android-)Services mit den Service-Klassen im Schichtenmodell auf dem Business-Logic-Layer als allgemeines Programmierkonzept. Es können auch eigene Service-Klassen zum kapseln von Businesslogik verwendet werden, die nichts mit den Android-Services zu tun haben.

openHAB SDK
In unserer Android App verwenden wir das REST-API von openHAB. Dazu braucht es eigene Service-Klassen zum Ausführen der REST Calls und zum Mappen der JSON Responses in Domain-Objekte (Sitemaps, Pages, Widgets, Items). Ebenso braucht es eine Schnittstelle zum persistenten Speichern von Verbindungsinformationen. Als wir die Android App planten, wären wir sehr froh gewesen, wenn es ein openHAB SDK für Android gäbe, das all diese Aufgaben schon für uns übernehmen würde. Dann müssten wir nur noch Activities mit entsprechenden Layouts entwickeln und uns wäre ein Grossteil des Aufwands erspart geblieben. Die offizielle Android App von openHAB ist zwar Open Source, jedoch derart schlecht aufgebaut, dass nichts davon wiederzuverwenden war.

Deshalb möchten wir während der Entwicklung unserer App gleich selbst solch ein SDK aufbauen. Aus Zeitgründen werden wir uns aber auf diejenigen Funktionen beschränken, die wir tatsächlich benötigen. Die Architektur der App teilt sich deshalb auf in einen Teil SDK und einen Teil App (siehe Abbildung 12).

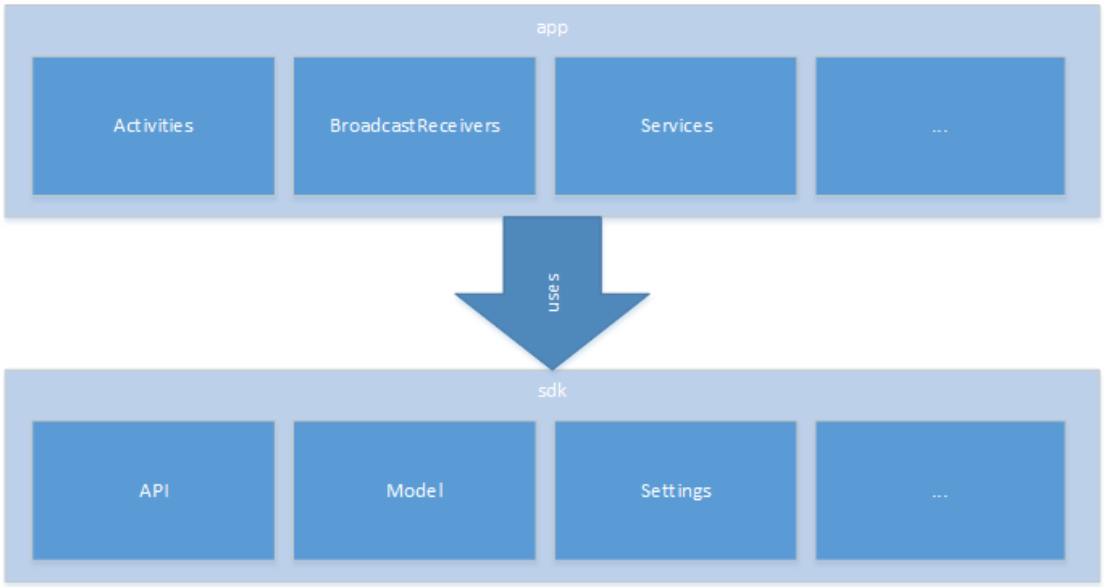


Abbildung 12.: Aufteilung SDK und App

Im SDK-Package befinden sich Klassen für die Verwendung des REST-API und Model-Klassen, um die Daten des API intern zu repräsentieren. Wir haben auf den Einsatz eines Dependency Injection Container verzichtet und verwenden ein Singleton als Einstiegspunkt ins SDK, um unnötige Komplexität zu verhindern. Das Singleton dient nur dazu, Referenzen auf die Bestandteile im SDK zur Verfügung zu stellen. Zudem werden im SDK auch diverse Hilfsklassen zu finden sein.
Das App Package besteht aus allen Klassen, die speziell für diese App so benötigt werden und nicht zwingenderweise für andere Apps wiederverwendbar sein müssen. Das betriffet hauptsächlich alle Activities und diverse UI Elemente.

Klassendiagramm
Die nachfolgende Abbildung zeigt einen vereinfachten Auszug aus dem geplanten Klassendiagramm der App. Auch hier wird wieder die Trennung zwischen SDK und App sichtbar. Eine Erklärung zu ausgewählten Klassen befindet sich im Umsetzungsteil der

Dokumentation im Abschnitt 4.3.2 auf Seite 64. Hinweis: Die dargestellten Klassen können in der tatsächlichen Implementierung in Methoden und Instanzvariablen abweichen.

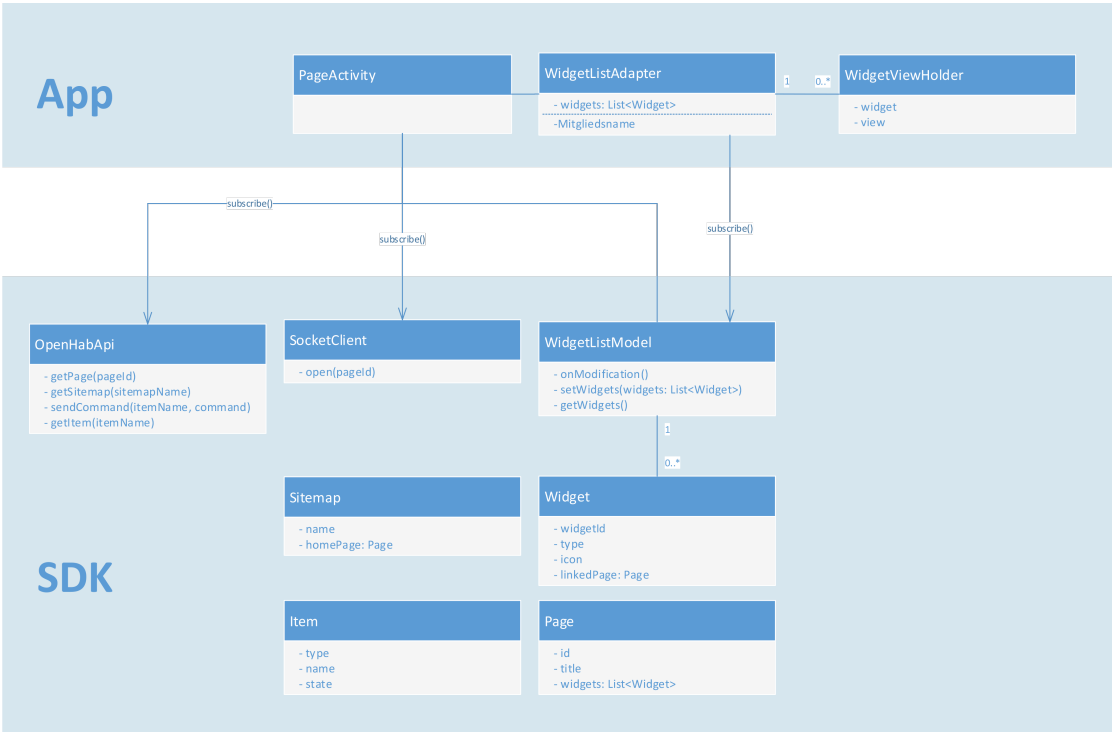


Abbildung 13.: Klassendiagramm mit PageActivity und SDK

3.8. Anbindung Cloud

In den funktionalen Anforderungen zum Basisszenario wurde festgelegt, dass Geschehnisse auf dem lokalen Bus (openHAB) an die Cloud gesendet werden müssen. Als Cloud-Plattform wurde Microsoft Azure bereits zu Beginn der Arbeit definiert. Damit die Daten von openHAB in die Cloud gelangen gibt es im Wesentlichen zwei Möglichkeiten: Entweder man entwickelt selbst ein Plugin für openHAB oder man verwendet das openHAB MQTT Persistence Plugin. MQTT ist ein Publish/Subscribe-Protokoll und besonders gut für IOT Systeme mit langsamer oder instabiler Internetverbindung geeignet. Wir haben uns für die Variante mit MQTT entschieden. OpenHAB fungiert als MQTT-Client und kann so konfiguriert werden, dass Events auf dem Bus automatisch an einen MQTT-Broker gesendet werden, der von uns auf Microsoft Azure installiert wurde. Um die Daten zu verarbeiten muss ein MQTT-Client auf Microsoft Azure laufen und die Nachrichten vom

Broker entgegennehmen. Danach steht uns frei, wie die Daten verarbeitet werden. Details zur Implementierung und technische Beschreibungen sind im Kapitel «Umsetzung» dieser Dokumentation zu finden.

3.8.1. MQTT Broker Evaluation

Für die Evaluation des Brokers haben wir eine List mit unseren Anforderungen erstellt:

1. Open Source/Freeware
2. SSL TLS Verschlüsselung
3. Benutzername & Passwort Authentifizierung
4. High throughput, Low latency
5. Cloud Ready
6. Einfache Installation
7. Qualit of Service Level: Exactly once
8. Last Will unterstützung (Message, die gesendet wird, wenn der Client die Verbindung schliesst)

HiveMQ (`http://www.hivemq.com`)
HiveMQ ist ein proprietärer MQTT-Broker und erfüllt alle unsere Kriterien bis auf das erste Kriterium. Jeder Gebrauch des Brokers muss bezahlt werden, siehe: `http://www.hivemq.com/pricing/`.

Mosquitto (`http://eclipse.org/mosquitto/`)
Mosquitto ist ein Open Source Broker, dessen Projekt von Roger Light im Jahr 2010 auf die Beine gestellt wurde. Seit der Version 1.4 läuft das Projekt unter der Eclipse Foundation.
Die aktuelle Implementation benötigt lediglich 120kB Speicherplatz und 3MB RAM bei 1000 verbundenen Clients. Ein Belastungstest von 100'000 Clients erzielte erfolgreiche und zufriedenstellende Resultate. Alle anderen Kriterien werden ebenfalls erfüllt.

CloudMQTT
CloudMQTT unterscheidet sich von den anderen Brokern, da dieser nicht selbst betrieben werden kann. Das bedeutet, man erstellt bei CloudMQTT eine Instanz und kann denn neu erstellten Broke über verschiedene APIs ansteuern.
Der Anbieter stellt verschiedene Preispläne zur Verfügung (`http://www.cloudmqtt.com/plans.html`). Ab 10 Verbindungen bzw. 10Kbit/s Bandbreite muss für die Leistung

bezahlt werden.
Ob TLS SSL Verschlüsselung unterstützt wird, ist in der spärlichen Dokumentation nicht ersichtlich.

Fazit
Aufgrund der gesammelten Daten der drei Produkte wurde entschieden, Mosquitto einzusetzen. Es ist das einzige Produkt, welches alle unsere Kriterien erfüllt. Da dieses Projekt durch die Eclipse Foundation unterstützt wird, besteht die Hoffung, dass die Zusammenarbeit mit openHAB unterstützt bzw. miteinbezogen wurde.

3.8.2. Client-Library

In unserem Systemaufbau wird es zwei MQTT-Clients geben. Einerseits durch das openHAB-Binding, da die Events auf dem EventBus über MQTT in die Azure Cloud gesendet werden soll. Andererseits wird in der Cloud eine Worker Role diese Events konsumieren und persistieren.
Das Binding von openHAB besteht bereits, daher muss nur noch eine Client-Implementation für C# gesuchtwerden. Nach kurzer Rechereche scheint die «M2Mqtt» Library sehr verbreitet zu sein.
Durch aufsetzen eines Prototyps konnte die benötigte Funktionalität erfasst werden. Sie erfüllt alle Kriterien, die auch an den Broker gestellt wurden.

3.9. Push-Notifications

Damit ein Sicherheitssystem Sinn macht, muss der Besitzer benachrichtigt werden, sobald sich ein Zustand der Sensoren ändert. Genauer gesagt soll der Benutzer informiert werden, wenn der Bewegungsmelder oder der Fensterkontakt im Alarm-Modus eine Veränderung registriert. Ansonsten müsste der Benutzer permanent die Status im Android-App überwachen. Für die Umsetzung bietet sich der Google Cloud Message Dienst (GCM) an, der auch von Microsoft empfohlen wird. Da nur die Android-Platform unterstützt werden muss, wird auf den Einsatz des Azure Notification Hubs verzichtet und die direkte Verbindung zum GCM-Dienst gesucht. Erfahren Sie im entsprechenden Kapitel des Umsetzungsteil auf Seite 71, wie die Notification Funktion cloud- und androidseitig umgesetzt wurde.

3.10. Sicherheit

In unserem Szenario, Einbruchschutz, ist das Thema Sicherheit sehr wichtig. Daher lohnt es sich dazu Gedanken zu machen. Man kann das System grob in drei Bereiche einteilen, an denen ein Angriff angesetzt werden kann:

- Cloud
- Android App
- Systemaufbau mit Sensoren/Aktoren
- Kommunikation der HomeMatic Komponenten

3.10.1. Cloud

Die Angriffsfläche in der Cloud ist ziemlich eingeschränkt. Sicherheitsrelevante Aspekte sind der Broker und die persistierten Daten im Storage. Zugriff auf diese Ressourcen erhält man über verschiedene Wege. Einerseits über das Management Web-Interface von Microsoft Azure oder über ein Connection-String. Durch die Verwendung eines Connection Strings für den Verbindungsaufbau zum Storage kein Benutzername bzw. Passwort mitgegeben werden.

Zugriff auf diese Ressourcen haben nur authorisierte Personen die ein entsprechendes Microsoft Konto besitzen und auf die Subscription zugelassen werden. Eine vernünftige Attack kann daher nur durch das Hijacken eines Accounts gefahren werden. Als Gegenmassnahme müssen ausreichend sichere Passwörter auf den persönlichen Konten gesetzt werden. Das liegt in der Verantwortung der einzelnen Personen, die auf diese Subscription Zugriff besitzen.

Auf den Storage kann ausser der Management-Oberfläche auch über ein Connection-String zugegriffen werden. Dieser String kann nur über die Management-Oberfläche eingesehen werden und ist daher sicher versteckt. Weiter könnte der Connection String aus dem Source-Code der Worker Role gelesen werden, da er dort für den Verbindungsaufbau zum Storage verwendet wird. Diese Worker Role und deren Source-Code befindet sich innerhalb der Cloud und ist somit auch gegen ein Eindringen gesichert.

3.10.2. Android App

Über die Android App kann das ganze System manipuliert werden. Daher stellt dies ein grosses Sicherheits-Risiko dar. Gesichtert wird die App durch Eingabe von Benutzerdaten, um auf das REST-API von openHAB zuzugreifen.
Auch hier wird die Person, die das App verwendet zur Verantwortung gezogen, um das Smartphone durch ein sicheres Passwort zu schützen.

3.10.3. Systemaufbau mit Sensoren/Aktoren

Wenn sich die angreifende Person bereits im Haus befindet,hat sie die entsprechenden Alarm bereits ausgelöst und wurde durch die Webcam aufgenommen.
Für dieses Szenario ist das grösste Risiko, dass die Person die Videoüberwachung bemerkt und aus diesem Grund das Raspberry Pi zerstört bzw. mitnimmt. Um die aufgezeichneten Daten zu sichern, werden die von der Kamera erzeugten Bilder über MQTT im Cloud-Storage abgelegt. Auch wenn das Raspberry Pi zerstört wird, sind die Bilder nun gesichert und können im Nachhinein vom Storage heruntergeladen werden.

3.10.4. HomeMatic Kommunikation

Da die HomeMatic Komponenten untereinander drahtlos kommunizieren besteht hier ebenfalls die Gefahr einer Attacke. HomeMatic setzt für die Übertragung auf den Funkstandard BidCos. Diese Standard sieht vor, dass das 868 MHz Band verwendet wird. Im Vergleich zu 2.4 GHz besitzt man so eine grösse Reichweite und die Gefahr von Interferenz durch andere Funkgeräte ist sehr minim.
Bezüglich Sicherheit sieht der Standard lediglich vor, die übertragenen Daten durch XOR-Operationen unleserlich zu machen. HomeMatic hat daher bei der Implentierung dieses Stanards die Verschlüsselung durch AES-128 verstärkt. Der zur Verschlüsselung verwendete Key ist in der Zentrale hinterlegt und kann bei Bedarf geändert werden.

3.10.5. Fazit

Wie aus den Ausführungen zur Sicherheit ersichtlich ist, können alle technischen Sicherheits-Risiken abgedeckt werden. Die grösste Schwachstellt ist jeweils der Mensch, der durch Social-Engineering-Techniken manipuliert werden kann und so Zugriff auf das Android App gewährt.

4. Umsetzung

In diesem Hauptkapitel werden Details zur Umsetzung dokumentiert. Hier sind Konfigurationen, Technologiebeschreibungen, Code Listings und sonstige technische Erklärungen über openHAB, MQTT und die Android App zu finden.

Zu Beginn geht es darum, die Hardware zu vernetzen und zu konfigurieren, so dass die einzelnen Komponenten im Zusammenspiel mit openHAB funktionieren. Dazu wird zuerst das Raspberry Pi konfiguriert und das openHAB Framework installiert. Als nächstes werden die beiden Zentralen von HomeMatic und Philips Hue in Betrieb genommen, sowie die Überwachungskamera. Jede Zentrale muss dabei die eigenen Geräte «anlernen», damit sie weiss, welche Sensoren und Aktoren existieren. Dazu bietet das HomeMatic ein Webinterface und von Philips gibt es ein Android bzw. iPhone App.
Damit die Zentralen mit openHAB kommunizieren können, werden sie zusammen mit dem Raspberry Pi über LAN-Kabel an den Router angeschlossen. Die Webcam kann über dessen Webinterface über WLAN eingebunden werden.

Damit openHAB mit den Komponenten kommunizieren kann, werden die Zentralen im Config-File eingetragen und die einzelnen Sensoren und Aktoren als Item definiert.
Um intelligente Abläufe zu definieren, bietet openHAB Regeln an. Über diese kann beispielsweise definiert werden, dass das Licht eingeschaltet wird, sobald das Fenster geöffnet wird.

Damit alle Events und auch die Bilder der Webcam gesichert werden können, wird MQTT (weiteres dazu im Kapitel 4.2, ab Seite 48) eingesetzt. Über dieses Protokoll werden die Daten an die Cloud gesendet und dort im Storage abgelegt. Dazu gibt es in der Azure Cloud ein Clouddienst mit einer Worker Role und einem MQTT-Broker. Der Broker nimmt die Messages entgegen, die Worker Role konsumiert diese und speichert sie ab.

Zugänglich ist das System über ein selbst geschriebenes Android App. Über dieses App können die einzelnen Glühbirnen, der Fensterkontakt und der Bewegungsmelder über openHAB angesprochen werden.

4.1. openHAB Konfiguration

Items, Rules, Sitemaps und Persistence Strategies werden in openHAB in einer eigens dafür entwickelten DSL beschrieben. Für alle diese Bereiche werden Konfigurationsdatei-

en im entsprechenden Unterverzeichnis des openHAB Konfigurationsordners abgelegt. Änderungen an den Konfigurationsdateien werden von openHAB zur Laufzeit sofort erkannt und berücksichtigt. Nebst den genannten Bereichen existiert eine globale Konfigurationsdatei, um beispielsweise IP-Adressen für Bindings einzutragen. Folgende Dateien sind für diese Arbeit relevant:

- /configurations/openhab.cfg
- /configurations/items/demo.items
- /configurations/rules/demo.rules
- /configurations/sitemaps/demo.sitemap
- /configurations/persistence/mqtt.persist
- /configurations/transform/*.map

Anhand dieser Konfigurationen werden die User Cases modelliert.

4.1.1. Items

Die Definition der Items erfolgt in der Datei `demo.items` und verwendet die openHAB Item DSL. Details zur Syntax sind auf dem openHAB Wiki (unvollständig) dokumentiert. Siehe `https://github.com/openhab/openhab/wiki/Explanation-of-items#syntax`.

Nachfolgend werden die wichtigsten Items anhand von Listing 1 erklärt.

```
1  Switch Alarm_active
2     "Alarm-Modus [MAP(alarm.map):%s]"
3     <siren>
4     (Alarms)
5
6  Group:Switch:AND(ON, OFF) Alarms
7     "Ueberwachung [MAP(alarm.map):%s]"
8     <house>
9
10 /* Bewegung */
11 String Motion_Livingroom
12    "Bewegungsmelder [MAP(motion.map):%s]"
13    {homematic="address=LEQ0797607, channel=1, parameter=MOTION"}
14
15 String Motion_Livingroom_Sabotage
```

```
16      "Sabotage"
17      {homematic="address=LEQ0797607, channel=1, parameter=ERROR"}
18
19  String Motion_Summary
20      "Bewegungsmelder [%s]"
21      <siren>
22
23  /* Lichter */
24  Switch Hue_1
25      "Schlafzimmer [MAP(switch.map):%s]"
26      (Lights)
27      {hue="1"}
28
29  Switch Hue_2
30      "Wohnzimmer [MAP(switch.map):%s]"
31      (Lights)
32      {hue="2"}
33
34  Group:Switch:OR(ON, OFF) Lights
35      "Lichter [%d an]"
36
37  /* Fenster */
38  String Window_Bedroom
39      "Schlafzimmerfenster [MAP(window.map):%s]"
40      <contact>
41      (Windows)
42      {homematic="address=LEQ1469091,channel=1,parameter=STATE"}
43
44  Group:String:OR(true, false) Windows
45      "Fenster [%d offen]"
46      <contact>
```

Listing 1: demo.items - Konfiguration der Items

Kontaktsensor, Zeile 38 - 42

Der Kontaktsensor ist ein Use Case, der in openHAB sehr oft vorkommt. Deshalb existiert ein Item Type «Contact» mit den beiden möglichen Werten OPEN und CLOSED. Das Homematic Binding liefert aber die Werte `true` oder `false` und kann deshalb nicht an ein Contact Item gebunden werden. Als Alternative haben wir ein String Item verwendet und mit Hilfe einer Transformation-Map `false` mit «geschlossen» und `true` mit «offen»

übersetzt. Das Icon <contact> kann trotzdem noch verwendet werden. Allerdings wird nun nicht mehr automatisch das Icon für das offene bzw. geschlossene Fenster angezeigt, da openHAB den Basisnamen des Icons mit dem State konkateniert. Wir haben dementsprechend die Icons umbenannt.

Auf der letzten Zeile werden die Bindingparameter für Homematic angegeben.
`address=LEQ1469091` bezieht sich auf die Seriennummer des Kontaktsensors und `parameter=STATE` gibt an, welche Eigenschaft gebunden werden soll. `channel=1` ist ein Defaultwert.

Bewegungsmelder, Zeile 11 - 21
Der Bewegungsmelder ist ebenfalls von Homematic und wird in den Bindingparametern gleich referenziert wie beim Kontaktsensor. Der Bindingparameter `MOTION` erhält einen boolschen Wert und muss mit Hilfe der Map `motion.map` in einen benutzerfreundlichen String transformiert werden. Der Parameter `ERROR` dient dem Sabotageschutz.

Leider muss bei Homematic jeder Parameter einem eigenen Item zugeordnet werden. Deshalb haben wir ein drittes String-Item `Motion_Summary` definiert, in dem die Werte der beiden anderen Items zusammengefasst werden.

Lichter, Zeile 24 - 32
Die Philips Hue Lampen können direkt an ein Switch Item gekoppelt werden. Im Bindingparameter muss lediglich die Nummer der Birne eingetragen werden. Der Status (`ON/OFF`) wird mit der Transformation-Map in «Ein» bzw. «Aus» übersetzt.

Alarm, Zeile 1 - 4
Dieses Switch Item dient zum Wechseln des Alarm-Modus. Der Status wird ähnlich wie bei den Lichtern mit einer Transformation-Map in einen sinnvollen deutschen Ausdruck übersetzt.

Aggregierte Gruppen
Die restlichen Items sind Group Items. Sie werden für die Sitemap als Zusammenfassung der zugehörigen Items verwendet. Das Item `Lights` zählt beispielsweise die Anzahl eingeschalteter Lichter.

4.1.2. Sitemap

Mit einer Sitemap wird das User Interface beschrieben, um Items für verschiedene Frontends verfügbar zu machen. In einer baumartigen Struktur können Items an sogenannte Widgets gebunden werden. Ein Widget ist einfach ausgedrückt ein Listenelement im User Interface. Je nach Widget und zugehörigem Icon ist das Element neben Icon und Text auch noch mit Controls ausgestattet. Group-Widgets führen den Benutzer auf eine neue Seite. Die Navigation ist nach dem Drill-Down Prinzip aufgebaut. Unsere Sitemap Definition ist leicht an die Bedürfnisse der Android App angepasst. Weitere Informationen zur Android App befinden sich im Abschnitt 4.3, ab Seite 59.

```
1  sitemap demo label="Ihr Haus" {
2     Group item=Windows {
3        Frame label="Nordseite" {
4           Text item=Window_Bedroom
5        }
6     }
7     Group item=Lights icon="light" {
8        Frame label="Nordseite" {
9           Switch item=Hue_1
10       }
11       Frame label="Suedseite" {
12          Switch item=Hue_2
13       }
14    }
15    Group item=Alarms {
16       Frame label="Alarm" {
17          Switch item=Alarm_active
18       }
19       Frame label="Bewegung" {
20          Text item=Motion_Summary
21       }
22       Frame label="Kamera" {
23          Webview url="http://192.168.1.15:80/jpg/1/image.jpg"
24       }
25    }
26 }
```

Listing 2: demo.sitemap - User Interface Deklaration

4.1.3. Rules

Wie im Lösungskonzept beschrieben, werden Rules zur Automatisierung von Aktionen eingesetzt. Die Regeln können im File `/configurations/rules/demo.rules` definiert werden.

Alarm

In der Grafik 14 wird der Ablauf der Alarm-Regel gezeigt. Im Wesentlichen geht es darum, beim Erkennen von Bewegung und offenem Fenster das Licht einzuschalten und die aufgenommenen Bilder der Webcam zu speichern. Dies soll jedoch nur geschehen, wenn der Alarm eingeschaltet ist.

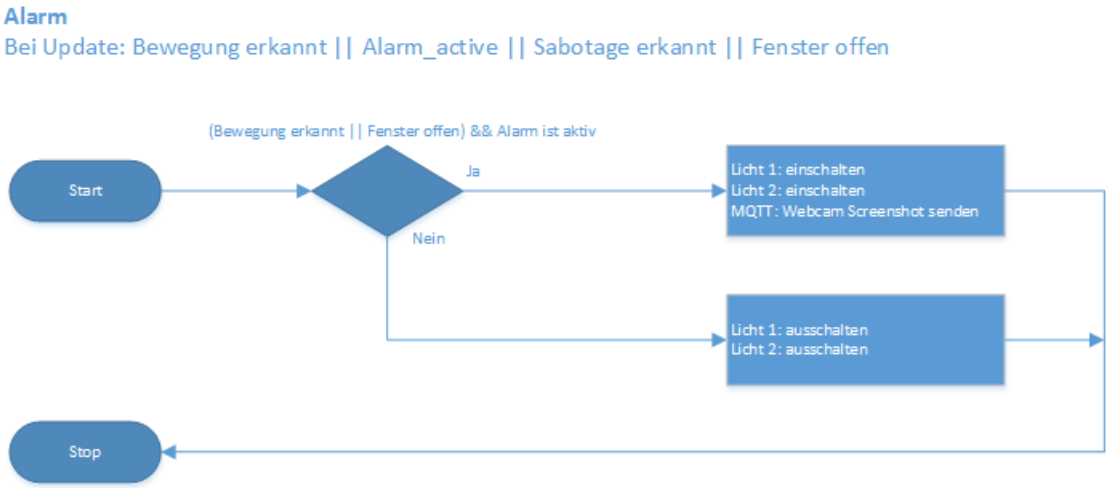


Abbildung 14.: Flowchart Alarm Rule

Umgesetzt wird diese Flowchart durch folgenden Code. Diese Rules sind in der von openHAB selbst definierten DSL (Domain Specific Language) geschrieben.

```
1  rule "Alarm"
2     when Item Motion_Livingroom received update or
3          Item Motion_Livingroom_Sabotage received update or
4          Item Window_Bedroom received update
5     then
6     if(Motion_Livingroom.state == "true" ||
7        Window_Bedroom.state == "true") {
```

```
8         sendCommand(Hue_1, ON)
9         sendCommand(Hue_2, ON)
10        sendMqttFile("openhab/blob",
11                     "http://192.168.1.15/snapshot.jpg")
12    } else {
13        sendCommand(Hue_1, OFF)
14        sendCommand(Hue_2, OFF)
15    }
16 end
```

Listing 3: demo.rules - Rule «Motion Aggregator»

Bewegungsmelder Aggregation

Diese Regel definiert die verschiedenen Status, die der Bewegungsmelder haben kann. Durch diese Regel wird der aktuelle Status ermittelt und für das Item `Motion_Summary` gesetzt.

Da der Bewegungsmelder Bewegungen und Sabotage erkennen kann (wenn die Halterung entfernt wird), ergeben sich vier mögliche Zustände: «Ruhig», «Bewegung erkannt», «Bewegung und Sabotage erkannt» und «Sabotage erkannt».

In der Abbildung 15 wird der Ablauf illustriert.

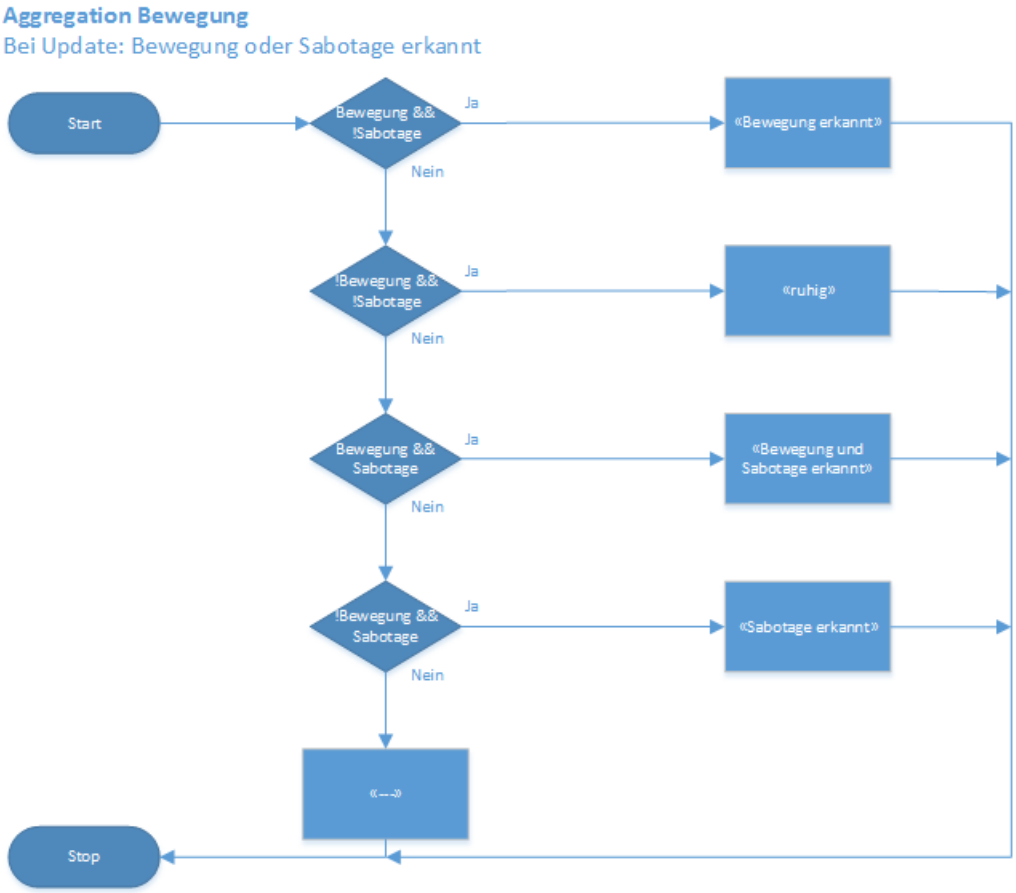


Abbildung 15.: Flowchart Motion Aggregator Rule

Nachfolgend ist die Definition der Regel in der openHAB DSL beschrieben. Falls keiner der vier möglichen Zustände zutrifft, was nur in einem Fehlerfall möglich ist, wird «—» im Item `Motion_Summary` angezeigt.

```
1  rule "Motion Aggregator"
2     when Item Motion_Livingroom received update or
3          Item Motion_Livingroom_Sabotage received update
4     then
5     if(Motion_Livingroom.state == "true" &&
6        Motion_Livingroom_Sabotage.state == "NO_ERROR") {
7        postUpdate(Motion_Summary, "Bewegung erkannt")
8     } else if(Motion_Livingroom.state == "false" &&
```

```
9               Motion_Livingroom_Sabotage.state == "NO_ERROR") {
10      postUpdate(Motion_Summary, "ruhig")
11   } else if(Motion_Livingroom.state == "true" &&
12               Motion_Livingroom_Sabotage.state == "SABOTAGE") {
13      postUpdate(Motion_Summary, "Bewegung und Sabotage!")
14   } else if(Motion_Livingroom.state == "false" &&
15               Motion_Livingroom_Sabotage.state == "SABOTAGE")  {
16      postUpdate(Motion_Summary, "Sabotage")
17   } else {
18      postUpdate(Motion_Summary, "---")
19   }
20 end
```

Listing 4: demo.rules - Rule «Motion Aggregator»

4.2. MQTT

MQTT steht für «Message Queue Telemetry Transport» und ist ein Nachrichten-Protokoll, das speziell für IoT-Anwendungen konzipiert wurde. Es setzt auf dem TCP/IP Stack auf und wird für den Nachrichtenaustausch zwischen verschiedenen, verteilten Maschinen verwendet.
Das Protokoll wurde speziell für Systeme designt, die über wenig Speicherplatz und kleine Netzwerk-Bandbreite verfügen, was bei IoT-Anwendungen meist der Fall ist.

4.2.1. Funktionsweise

MQTT folgt dem Prinzip «Publish/Subscribe», das heisst Clients können bestimmte Topics abonnieren. Wenn Messages auf dieses Topic gesendet werden, leitet der Broker diese an alle interessierten Clients weiter.
In Bezug auf das Erstellen von Topics agiert der Broker passiv. Das bedeutet, Clients können sich auf beliebige, selber definierte Topics registrieren. Wenn aber niemand auf dieses Topic publiziert, wird der Client nie eine Message erhalten.

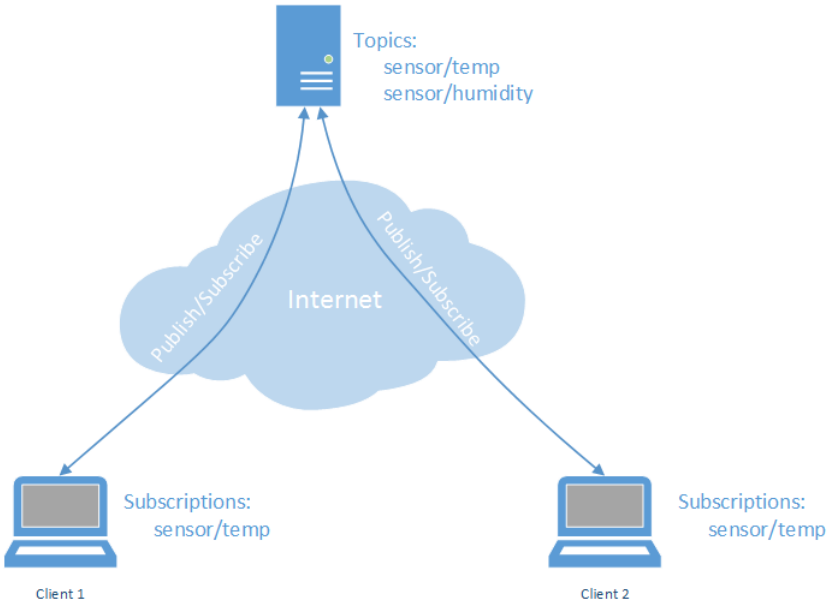


Abbildung 16.: Funktionsweise MQTT

Das Deploymentdiagramm (Abbildung 17) zeigt, wie der Cloudservice und das Smart-

Home aufgebaut sind. Der Cloudservice umfasst eine virtuelle Maschine, eine Worker Role und den Storage. Auf der virtuellen Maschine läuft der Broker. Eine Worker Role hört auf ein Topic, erhält die Nachrichten vom Broker und speicher sie in den Cloudstorage. Auf dem Raspberry Pi werden über das MQTT-Binding des openHABs die Item-Updates an den Broker, in die Cloud, gesendet.

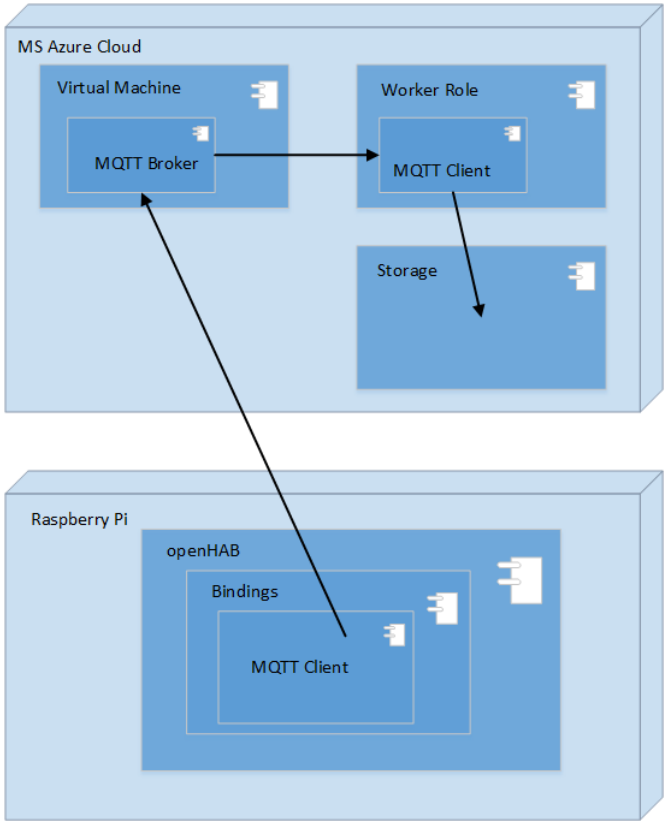


Abbildung 17.: Deployment Diagramm

4.2.2. Broker

Zertifizierungsstelle/Server-Zertifikat

Da die MQTT-Verbindung verschlüsselt werden soll, müssen verschiedene Zertifikate erstellt werden. Das erstellte Server-Zertifikat muss von einer CA (Certification Authority) signiert werden. Da ein gültiges Zertifikat nicht entgeltlich erworben werden möchte, wird eine eigene Zertifizierungsstelle erstellt. OpenSSl bringt dafür alle nötigen Mittel für die Erzeugung eines CAs mit. Dies bringt den Nachteil mit, dass Computersysteme

diesem Zertifikat nicht automatisch trauen. Daher muss das Zertifkat von Hand dem Certificate Store als «Trusted Root Certification Authority» hinzugefügt werden.

Nachdem die Zertifizierungsstelle erfolgreich generiert wurde, kann das Server Zertifikat erstellt werden. Anschliessend muss dieses Server Zertifikat von der eben erstellten Zertifizierungsstelle signiert werden.

Da sowohl die Zertifizierungsstelle, als auch das Server-Zertifikat auf dem gleichen Computer erstellt werden, muss darauf geachtet werden, dass bei der Erzeugung unterschiedliche Parameter gesetzt werden. Die betroffenen Parameter sind zum Beispiel «Locality Name», «Organizational Name», «Organizational Unit» etc. Falls hier dieselben Werte eingetragen werden, schlägt die Signierung des Serverzertifikates fehl.
Weiter muss beachtet werden, dass im Server-Zertifikat der «Common Name» dem FQDN (Fully Qualified Domain Name) des Servers entspricht, auf dem der MQTT-Broker laufen soll. Wird hier beispielsweise nur der Hostname eingetragen, schlägt die Überprüfung des Zertifikates fehl, da sich der CN vom FQDN des Servers unterscheidet.

Installation und Konfiguration des Brokers
Wie bereits im Lösungskonzept erarbeitet, wird als MQTT-Broker «Mosquitto» eingesetzt. Der Broker kann als Binary installiert und über die Commandline gestartet werden. Nach der Standard-Installation muss der Broker konfiguriert werden. Dazu wird das File «mosquitto.conf» bearbeitet.
Folgende Parameter müssen editiert werden:

Parameter	**Erklärung**
bind_address	
mqttbrokerba.cloudapp.net	IP-Adresse, an die der Default-Listener gebunden wird.
port 8883	Port, auf den der Default-Listener hören soll. Wenn er nicht speziell definiert wird, hört der Listener per Default auf den Port 1883. Da aber mit TLS verschlüsselt wird, muss dieser von Hand auf den dafür vorgesehenen Port 8883 gesetzt werden.

cafile	
`C:\OpenSSL-Win64\bin\m2mqtt_ca.crt`	Hier wird der Pfad eingetragen für das zuvor erstellte CA-Zertifikat.
certfile	
`C:\OpenSSL-Win64\bin\m2mqtt_srv.crt\path`	Hier wird der Pfad für das PEM-Encodete Server Zertifikat eingetragen.
keyfile	
`C:\OpenSSL-Win64\bin\m2mqtt_srv.key\path`	Hier wird der Pfad für das PEM-Encodete Keyfile eingetragen.
tls_version tlsv1	Diese Option definiert die zu verwendende TLS-Version. Für Openssl (Version 1.0.2) wird tlsv1 verwendet.
password_file	
`C:\ProgramFiles(x86)\mosquitto\passwords`	Das Passwords-File beinhaltet die definierten Benutzernamen und Passwörter, um sich am Broker anzumelden. Durch Setzen dieses Pfades wird dies automatisch vom Broker berücksichtigt.

Um den Broker zu starten, muss in der Konsole (cmd.exe) in das Installationsverzeichnis (`C:\ProgramFiles(x86)\mosquitto\`) navigiert werden. Dort kann der Broker mit der angepassten Konfiguration gestartet werden. Damit in der Konsole Feedback angezeigt wird, wird der Broker im «Verbose-Modus» gestartet:

```
mosquitto -c mosquitto.conf -v
```

4.2.3. openHAB MQTT Persistence (Publishing Client)

Im Lösungskonzept haben wir bereits erklärt, dass wir die Events auf dem openHAB Event Bus via MQTT an die Cloud übertragen wollen, und dass es ein MQTT Plugin

für openHAB gibt. Nachdem der Broker installiert worden ist, wollen wir nun unsere openHAB Items an den Broker senden. Dazu installieren wir das openHAB MQTT Persistence Plugin auf unserem Raspberry Pi:
`apt-get install openhab-addon-persistence-mqtt`

Danach müssen wir in der openHAB Konfiguration (openhab.cfg) angeben, wie wir uns mit dem Broker verbinden wollen:

```
1  mqtt:mosquitto.url =  ssl://mqttbrokerba.cloudapp.net:8883
2  mqtt:mosquitto.user=mosquitto
3  mqtt:mosquitto.pwd=******************
4
5  mqtt-persistence:broker=mosquitto
6  mqtt-persistence:topic=openhab/items
7  mqtt-persistence:message={
8     "name":"%1$s",
9     "state":"%3$s",
10    "date":"%4$s"
11 }
```

Listing 5: Auszug /configuration/openhab.cfg - MQTT Config

Jetzt müssen wir openHAB noch beibringen, wann genau die Items an den Broker gesendet werden sollen. Das geschieht über eine Persistence-Konfiguration. Im Ordner `configuration/persistence` erstellen wir ein File namens `mqtt.persist` mit dem folgenden Inhalt:

```
1  Strategies {
2     everyMinute : "1 * * * * ?"
3     everyHour : "0 0 * * * ?"
4     default = everyChange
5  }
6
7  Items {
8     * : strategy = everyChange, restoreOnStartup
9  }
```

Listing 6: /configuration/mqtt.persist

Damit erreichen wir, dass ein Item immer dann an den Broker gesendet wird, wenn sich der Status des Items ändert. Zudem sollen bei jedem Systemstart alle Items an den Broker geschickt werden. Jetzt fungiert openHAB als ein Publisher.

4.2.4. Azure Worker Role (Subscribing Client)

Wie die Abbildung 3 auf Seite 24 (Systemübersicht) zeigt, befindet sich nebst dem Broker auch ein Client, in Form einer Worker Role in der Cloud. Die Worker Role abonniert alle Topics und persistiert die Messages im Table bzw. Blob-Storage.

In der `OnStart()`-Methode der Worke Role werden zuerst die Referenzen zum Table- und Blob-Storage erzeugt.
Danach wird die Verbindung zum MQTT-Broker hergestellt, die Topics definiert, die er abonnieren möchte und der QoS-Level gesetzt.
Damit der Client benachrichtigt wird, wenn eine Message eintrifft, wird der Eventhandler `client_MqttMsgPublishReceived()` definiert. Die Methode muss die gleichen Parameter entgegennehmen, wie die Delegate-Methode. Das ist einerseits der Sender (Object) und andererseits die Event-Argumente. Damit der Eventhandler beim Eintreffen einer Message aufgerufen wird, muss dieser auf dem Event registriert werden:
`client.MqttMsgPublishReceived += client_MqttMsgPublishReceived;`

```
1  public override bool OnStart()
2  {
3    setupStorageConnections();
4    MqttClient client = new MqttClient(
5                          "mqttbrokerba.cloudapp.net",
6                             8883, true, null
7                            );
8    client.MqttMsgPublishReceived += client_MqttMsgPublishReceived;
9    client.Connect(Guid.NewGuid().ToString(),
10             "username",
11             "password"
12             );
13   string[] topics = { "openhab/+" };
14   byte[] qos = { MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE };
15   client.Subscribe(topics, qos);
16
17   return result;
```

```
18 }
```

Listing 7: WorkerRole.cs - MQTT Topic Subscribe

Im Eventhandler wird dann die Logik zur Persistierung eingefügt. Wenn eine Message über das Topic «openhab/blob» empfangen wird, handelt es sich um eine Fotografie der Webcam. Dieses JPG-File wird als Byte-Array übermittelt und wird so auch im Blob-Storage abgelegt.
Bei allen anderen Messages muss es sich um Text handeln, daher werden sie im Table-Storage abgelegt.

```
1  void client_MqttMsgPublishReceived(object sender,
2                                     MqttMsgPublishEventArgs e)
3  {
4     if (e.Topic.Equals("openhab/blob"))
5     {
6        CloudBlockBlob blockBlob = container.
7                       GetBlockBlobReference(Guid.NewGuid()
8                                                .ToString());
9        blockBlob.UploadFromByteArray(e.Message,
10                              0, e.Message.Length);
11     }
12    else
13    {
14       var message = System.Text.Encoding.
15                            Default.GetString(e.Message);
16       Entity entity = new Entity(message);
17       TableOperation insertOperation = TableOperation.
18                                    Insert(entity);
19       table.Execute(insertOperation);
20    }
21 }
```

Listing 8: WorkerRole.cs - EventHandler

Zertifikat

Da die Verbindung durch SSL/TLS mit einem Self-signed Zertifikat verschlüsselt wird, muss dieses Zertifikat dem Certificate Store des virtuellen Hosts als «Trusted Root

Certification Authority» hinzugefügt werden. Dies muss vorgenommen werden, bevor die Worker Role versucht eine Verbindung zum Broker herzustellen. Ansonsten terminiert die Worker Role mit einer Exception, da sie das Zertifikat nicht akzeptiert.
Da man zwischen Erzeugung des virtuellen Hosts und dem Start der Worker Role nicht auf die Maschine zugreifen kann, muss das Zertifikat programmatisch als Startup-Skript dem Store hinzugefügt werden. Für solche Aufgaben bietet Microsoft Azure die Möglichkeit, Startup Tasks zu definieren. Dies wird in der ServiceDefinition vorgenommen, durch Einfügen folgender Anweisung:

```
1 <Startup>
2    <
3       Task commandLine="startup.cmd" executionContext="elevated"
4       taskType="simple"
5    />
6 </Startup>
```

Listing 9: ServiceDefinition.csdef - Startup Task

Dieser Startup Task führt das Batchfile «startup.cmd» aus, welches das Zertifikat in den Store hinzufügt. Dazu muss das Skript und auch das Zertifikat dem Visual-Studio-Projekt als Element hinzugefügt werden. Für beide Elemente muss in den Eigenschaften der Buildvorgang als «Inhalt» deklariert werden.

```
1 certutil -addstore root m2mqtt_ca.cer
```

Listing 10: startup.cmd - Zertifikat hinzufügen

Im Sequenzdiagramm der Abbildung 18 wird aufgezeigt, wie die MQTT-Nachrichten nach dem QoS 2 versendet werden.

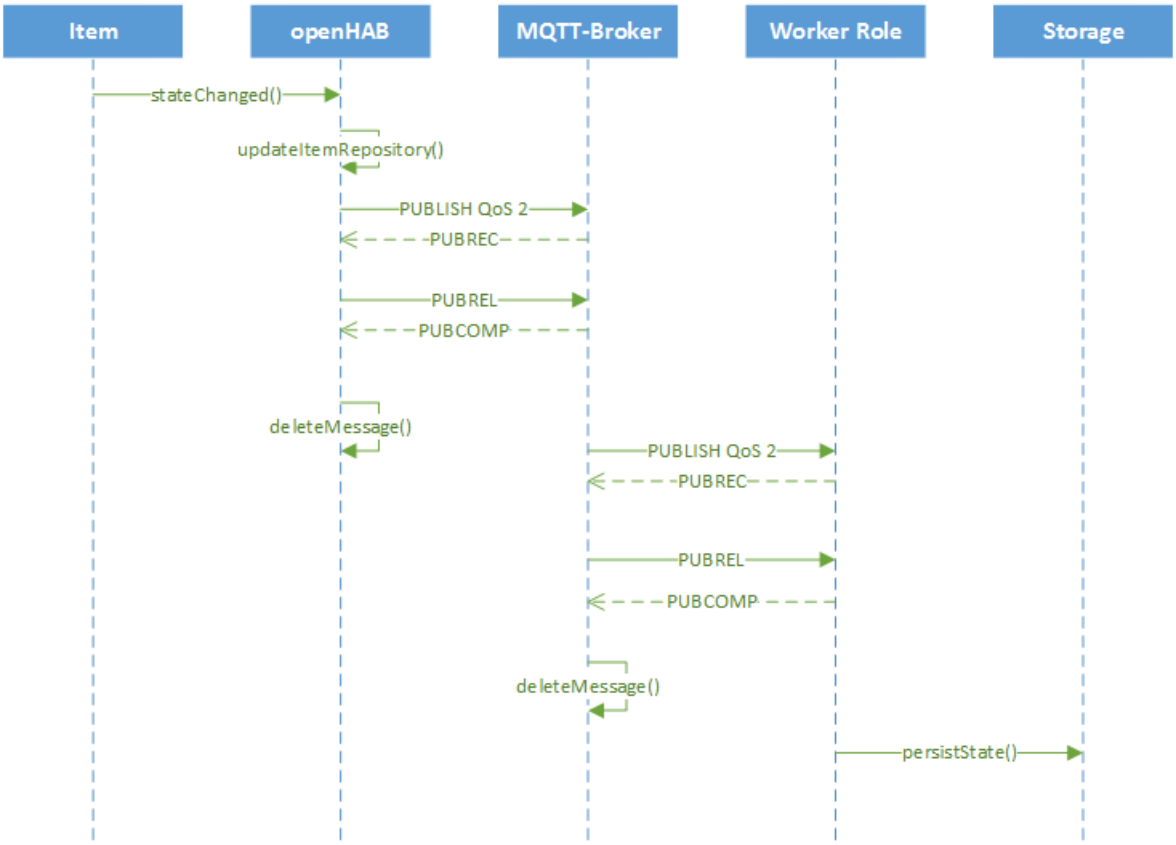


Abbildung 18.: Sequenzdiagramm MQTT

4.2.5. openHAB MQTT Action

Normalerweise wird aufgrund unserer Konfiguration ein openHAB Item direkt via MQTT an die Cloud gesendet, sobald sich der Status des Items ändert (beschrieben in Abschnitt 4.2.3 auf Seite . Da unsere Überwachungskamera aber nicht durch ein openHAB Item repräsentiert wird, kann openHAB auch nichts davon an die Cloud schicken. Trotzdem möchten wir gerne einen Snapshot der Kamera an die Cloud senden, sobald der Alarm ausgelöst wird (siehe Abbildung 6 - Sequenzdiagramm Alarm auf Seite 27 im Lösungskonzept).
Deshalb benutzen wir eine sogenannte openHAB Action zum Senden des Snapshots an den MQTT Broker. Die Action kann in der DSL zum Beschreiben der Rules, ähnlich wie ein statischer Methodenaufruf, verwendet werden. Natürlich gibt es diese spezielle Action noch nicht im Standardumfang von openHAB, sondern wir mussten diese in Form eines

Action-Plugins selbst programmieren.

Programmierung der MQTT Action

Zum Entwickeln von openHAB Plugins muss eine entsprechende Eclipse Umgebung eingerichtet werden. Dies sollte sich noch als der schwierigste Teil davon herausstellen. Eine Anleitung dazu findet man unter `https://github.com/openhab/openhab/wiki/IDE-Setup`. Bei uns hat nur die Variante mit «yoxos» funktioniert. Und auch bei dieser Anleitung mussten wir einige Tricks anwenden. Bei Schritt 8) musste die JDK Version in den Projekteinstellungen zunächst auf 1.6 zurückgestuft werden und vor Schritt 9) wieder auf Version 1.7. Beim Kompilieren werden alle openHAB Artifakte erstellt: OpenHAB selbst, alle Bindings, Actions und sonstige Plugins, sowie auch die Eclipse RCP's des openHAB Designers für alle Zielsysteme.

Sobald Eclipse vollständig eingericht war, kann ein Action Skeleton anhand eines Maven Archetypes generiert werden. Eine Anleitung dazu befindet sich im openHAB GitHub Wiki: `https://github.com/openhab/openhab/wiki/How-To-Implement-An-Action`. Die openHAB Runtime wird die Action-Aufrufe in den Rules später zur Laufzeit an die statischen Java-Methoden im Action-Plugin binden. Damit das später alles sauber funktioniert und auch die Dependency Injection von gewissen Service-Objekten gemacht werden kann, müssen die Dateien OSGI-INF/action.xml und META-INF/MANIFEST.MF angepasst werden:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
4       activate="activate" deactivate="deactivate" immediate="true"
5       name="org.openhab.action.mqtt.action">
6     <implementation class="org.openhab.action
7        .mqtt.internal.MqttActionService" />
8     <service>
9        <provide interface="org.openhab.core
10          .scriptengine.action.ActionService" />
11       <provide interface="org.osgi.service
12          .cm.ManagedService" />
13    </service>
14    <property name="service.pid" type="String"
15       value="org.openhab.mqtt-action" />
```

```
16    <reference bind="setMqttService" cardinality="1..1"
17          interface="org.openhab.io
18           .transport.mqtt.MqttService" name="MqttService"
19          policy="static" unbind="unsetMqttService"/>
20 </scr:component>
```

Listing 11: OSGI-INF/action.xml

```
1  Manifest-Version: 1.0
2  Private-Package: org.openhab.action.mqtt.internal
3  Ignore-Package: org.openhab.action.mqtt.internal
4  Bundle-License: http://www.eclipse.org/legal/epl-v10.html
5  Bundle-Name: openHAB Mqtt Action
6  Bundle-SymbolicName: org.openhab.action.mqtt
7  Bundle-Vendor: openHAB.org
8  Bundle-Version: 1.7.0.qualifier
9  Bundle-Activator: org.openhab.action.mqtt.internal.MqttActivator
10 Bundle-ManifestVersion: 2
11 Bundle-Description: This is the Mqtt action of the open Home Aut
12  omation Bus (openHAB)
13 Import-Package: org.apache.commons.lang,
14  org.openhab.core.scriptengine.action,
15  org.openhab.io.transport.mqtt,
16  org.osgi.framework,
17  org.osgi.service.cm,
18  org.slf4j
19 Bundle-DocURL: http://www.openhab.org
20 Bundle-RequiredExecutionEnvironment: JavaSE-1.6
21 Service-Component: OSGI-INF/action.xml
22 Bundle-ClassPath: .
23 Bundle-ActivationPolicy: lazy
```

Listing 12: META-INF/MANIFEST.MF

Dank diesen Konfigurationen wird der gleiche MQTT-Service Injected, der auch im MQTT-Persistence Plugin eingesetzt wird, und die statischen Java Methoden aus der Klasse org.openhab.action.mqtt.internal.Mqtt werden für die DSL sichtbar. Zudem stellt die openHAB Runtime der MQTT Action die zugehörigen Konfigurationen zur Verfügung, die in der globalen Konfigurationsdatei hinterlegt wurden (openhab.cfg).

Installation MQTT Action

Nachdem wir die MQTT Action programmiert hatten, mussten wir sie mit Hilfe von Eclipse als .jar Datei exportieren. Danach haben wir in der Konfigurationsdatei von openHAB (openhab.cfg) den Eintrag für die MQTT Action gemacht. Diese verweist auf den selben Broker, den wir auch für das Persistence Plugin benötigen.

```
1 mqtt-action:broker=mosquitto
```

Listing 13: openhab.cfg

Anschliessend haben wir die .jar Datei ins Add-on Verzeichnis unserer openHAB Installation kopiert. Die openHAB Runtime erkennt das neue OSGI-Modul sofort und aktiviert es.

Verwendung der MQTT Action

Die MQTT Action implementiert zwei Methoden: sendMQTTString(topic, message) und sendMQTTFile(topic, fileUrl). Die erste Methode sendet einen String an das Topic des vorkonfigurierten Brokers. Die zweite Methode ist diejenige, weswegen wir das Plugin überhaupt entwickelt haben. Sie nimmt eine URL entgegen, lädt den Inhalt von dort herunter und sendet ihn an das gewünschte Topic des Brokers. Beide Methoden können von jetzt an in openHAB Rules oder Scripts verwendet werden.

4.3. Android App

Im Lösungskonzept im Abschnitt 3.7.3, ab Seite 31, haben wir bereits erklärt, wie die Architektur der App aussieht. In diesem Abschnitt gehen wir auf die eingesetzten Frameworks, wichtige Klassen und das User Interface Design ein.

4.3.1. Frameworks und Libraries

Für Android gibt es mittlerweile eine grosse Vielzahl an äusserst hilfreichen und robusten Frameworks und Libraries, die den Entwickleralltag enorm erleichtern. Manche der Frameworks haben einen starken Einfluss darauf, wie der Code strukturiert wird. Für das Verständnis des Codes ist es deshalb wichtig, sich mit ReactiveX, Retrofit und Butterknife vertraut zu machen. Folgende Liste beinhaltet alle Gradle Build Dependencies:

- `com.squareup.retrofit:retrofit:1.9.0`

- `io.reactivex:rxandroid:0.24.0`
- `com.android.support:appcompat-v7:22.0.+`
- `com.android.support:cardview-v7:21.0.+`
- `com.android.support:recyclerview-v7:21.0.+`
- `com.squareup.okhttp:okhttp:2.4.0-RC1`
- `com.rengwuxian.materialedittext:library:2.0.3`
- `com.jakewharton:butterknife:6.1.0`
- `jp.wasabeef:recyclerview-animators:1.2.0@aar`
- `com.nispok:snackbar:2.10.+`
- `com.github.clans:fab:1.5.0`
- `com.google.android.gms:play-services:6.5.87`

ReactiveX

Die Reactive Extension wurde usprünglich für .NET entwickelt und ist mittlerweile fester Bestandteil davon. Heute gibt es die Library auch für diverse andere Plattformen, darunter Java bzw. auch eine spezielle Version für Android. ReactiveX verwendet das Paradigma der reaktiven Programmierung und unterstützt den Programmierer beim Ausdrücken von komplizierten Datenflüssen. In der konventionellen, imperativen Programmierung herrscht das Pull-Prinzip, sodass Änderungen an Daten aktiv geholt werden müssen. ReactiveX hingegen «pusht» neue Daten in die Konsumenten hinein. Dabei berücksichtigt ReactiveX auch viele Concurrency-Aspekte automatisch.

Das folgende (leicht vereinfachte) Beispiel stammt aus unserer App und demonstriert sehr gut die Vorteile von ReactiveX. Vom openHAB REST-API soll ein Item geladen werden, danach soll der boolsche Wert invertiert und wieder an das API zurückgesendet werden. Bevor der neue Status im User Interface angezeigt wird, wird der Status aus Konsistenzgründen noch einmal geladen:

```
1  api.getItem("Alarm_active")
2  .flatMap(item -> {
3      return Observable.just(item.state.equals("ON") ? "OFF" : "ON")
4  })
5  .flatMap(state -> api.sendCommand("Alarm_active", state))
6  .flatMap(aVoid -> api.getItem("Alarm_active"))
7  .subscribeOn(Schedulers.io())
8  .observeOn(AndroidSchedulers.mainThread())
9  .subscribe(item-> {
10    updateUi(item.state);
11 });
```

Listing 14: ReactiveX Beispiel

Jede dieser aneinander gereihten Methoden gibt ein Objekt von Typ `Observable<T>` zurück und modifiziert das vorherige Observable. Der Code in den Lambdas wird aber noch nicht ausgeführt sondern lediglich als Callable im Observable hinterlegt, für den Zeitpunkt wo Daten «hindurchfliessen». Erst beim Aufruf von `subscribe()` auf Zeile 9 wird die Maschine in Gang gesetzt. Die beiden Methoden auf Zeile 7 und 8 sind dazu gedacht, dass die API Calls in einem Hintergrundthread und das letzte Resultat wieder im Android Mainthread ausgeführt werden.
Ohne ReactiveX müsste man für diese Aufgabe drei Callback-Handler registrieren, die dann jeweils den nächsten API Call anstossen. Deshalb haben wir ReactiveX so oft wie möglich verwendet, mit Ausnahme von einigen Spezialfällen.

Retrofit

Diese Library erleichtert das Konsumieren von REST Webservices. Man erstellt zunächst ein Java Interface und schreibt die Methodensignaturen für die Requests. Der Rückgabetyp ist entweder das erwartete POJO oder `void`. Methoden mit dem Rückgabetyp `void` werden automatisch asynchron ausgeführt, verlangen aber einen Callback Handler als zusätzlichen Parameter. Methoden mit POJO Rückgabetypen werden synchron ausgeführt. Danach müssen alle Methoden noch mit den Annotations von Retrofit versehen werden. Unter anderem werden die gewünschte HTTP Methode (`GET, PUT, POST, DELETE, ...`), ein relativer URL Pfad und die Parameter angegeben. Besonders komfortabel ist, dass das Interface nicht selbst implementiert werden muss. Retrofit generiert die entsprechende Klasse anhand der Annotations. Das Erzeugen einer Instanz ist denkbar einfach:

```
1 RestAdapter adapter = new RestAdapter.Builder()
2     .setEndpoint(endpoint)
3     .setConverter(new GsonConverter(gson))
4     .build();
5
6 OpenHabApi api = adapter.create(OpenHabApi.class);
```

Listing 15: Retrofit - Service-Klasse generieren

In Listing 15, auf Zeile 3, wird ein `GsonConverter` übergeben. Dieser sorgt dafür, dass die JSON Response korrekt deserialisiert und auf den gewünschten POJO Typ gemappt wird. Dem `gson`-Objekt selbst können noch sogenannte TypeAdapter hinzugefügt werden, um mehr Kontrolle beim Mapping zu haben.

Noch besser als das automatische Generieren der Service-Klassen ist an Retrofit der Support für ReactiveX. Wenn man asynchrone REST Calls machen möchte, würde man normalerweise einen Callback Handler übergeben, wie bereits beschrieben. Es gibt allerdings eine elegantere Variante. Sofern ReactiveX als Dependency im Projekt aufgelöst werden kann, ist es möglich, den Rückgabetyp der Methoden im Service-Interface mit `Observable<T>` zu deklarieren. Anschliessend weiss Retrofit, dass die Methode asynchron und mit Hilfe von ReactiveX ausgeführt werden soll. Beim Aufruf der Methode erhält man also nicht direkt das POJO, sondern ein Observable, welches man subscriben kann. Sobald sich ein Subscriber anmeldet, wird der Request ausgeführt und das Resultat an den Subscriber geschickt. Folgendes Service-Interface deklariert die Methoden, welche bereits aus dem ReactiveX Code-Beispiel in Listing 14 bekannt sind:

```
1  public interface OpenHabApi {
2      @GET("/rest/items/{item}?type=json")
3      Observable<Item> getItem(@Path("item") String item);
4
5      @Headers("Content-type: text/plain")
6      @POST("/rest/items/{item}")
7      Observable<Void> sendCommand(@Path("item") String item,
8          @Body TypedInput command
9      );
10 }
```

Listing 16: OpenHabApi Service-Interface

Dank dem Zusammenspiel von ReactiveX und Retrofit wird dem Programmierer ein gewaltiger Teil an Fleissarbeit abgenommen. Es kann schneller entwickelt werden, die möglichen Fehlerquellen werden reduziert, und die Lesbarkeit bzw. Nachvollziehbarkeit verbessert sich.

Butterknife

Butterknife ist eine nützliche kleine Library zur Dependency Injection von View Komponenten in Activities. Jede Activity besitzt in der Regel eine Root-View, die in einem Layout Resource File mit XML definiert wird. In diesen Layout Files befinden sich typischerweise weitere User Interface Elemente, wie Buttons, Input-Felder oder Bilder. Die Activity ist dazu gedacht, jene Elemente mit Code zu beleben, also Events zu behandeln, Texte zu aktualisieren und so weiter. Die Referenzen auf diese Elemente muss sich die Activity selbst beschaffen, indem sie die `findViewById(int id)` Methode aufruft und das Element anschliessend einer lokalen Variable oder Instanzvariable zuweist. Die Methode `findViewById(int id)` hat den Rückgabetyp `View`, deshalb müssen alle Element explizit gecastet werden.

Butterknife macht diese Tipparbeit überflüssig, indem die Elemente als Instanzvariable deklariert und mit einer Annotation versehen werden:

```
1  public class SampleActivity extends Activity {
2
3      @InjectView(R.id.button)
4      Button mButton;
5
6      @Override
7      protected void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.activity_sample);
10         ButterKnife.inject(this);
11         // instead of: mButton = (Button) findViewById(R.id.Button);
12     }
13 }
```

Listing 17: Butterknife Beispiel 1

Ein Vorteil ergibt sich vor allem bei Activities mit vielen referenzierten User Interface Elementen. Butterknife erleichtert zudem das Behandeln von Events, indem Methoden ebenfalls mit einer Annotation versehen werden, anstatt dass die Activity ein Callback Interface implementieren muss bzw. anonyme Klassen verwendet werden:

```
1 @OnClick(R.id.button)
2 public void onClickButton(View view) {
3     // do something useful
4 }
```

Listing 18: Butterknife Beispiel 2

4.3.2. Wichtige Klassen

Zum besseren Verständnis werden hier einige ausgewählte Klassen und Packages genauer erklärt.

ch.hsr.baiot.openhab.sdk.model.*
Dieses Package ist Teil des SDK und beinhaltet die POJO Model-Klassen sowie einige Hilfsklassen. Die JSON Responses des REST-API werden auf diese Model-Klassen gemappt. Manche Model-Klassen implementieren das von uns definierte Interface `MemberEquals` mit der Methode `hasEqualMembers(other)`. Grund dafür ist, dass die Methode `equals(other)` bereits überschrieben wird und die Gleichheit zweier Objekte nur anhand des jeweiligen «Primärschlüssels» festgestellt wird. In einigen Fällen soll aber auch überprüft werden können, ob alle Instanzvariablen gleich sind, beispielsweise um zu bemerken, ob ein Objekt sich nach dem erneuten Laden verändert hat. Listing 19 zeigt die Model-Klasse `Item`:

```
1 package ch.hsr.baiot.openhab.sdk.model;
2
3 [...] // Imports
4
5 public class Item implements MemberEquals<Item>{
```

```
6     public String type  = "";
7     public String name = "";
8     public String state = "";
9     public String link = "";
10
11    @Override
12    public boolean equals(Object o) {
13       if (this == o) return true;
14       if (o == null || getClass() != o.getClass()) return false;
15       Item item = (Item) o;
16       if (!name.equals(item.name)) return false;
17       return true;
18    }
19
20    @Override
21    public int hashCode() { return name.hashCode();}
22
23    @Override
24    public boolean hasEqualMembers(Item other) {
25       if(this == other) return true;
26       if(other == null) return false;
27
28       if(this.type != null ? !this.type.equals(other.type) :
29          other.type != null) return false;
30       if(this.state != null ? !this.state.equals(other.state) :
31          other.state != null) return false;
32       if(this.link != null ? !this.link.equals(other.link) :
33          other.link != null) return false;
34       return true;
35    }
36 }
```

Listing 19: Item.java - Beispiel einer Model-Klasse

ch.hsr.baiot.openhab.sdk.model.WidgetListModel

OpenHAB Sitemaps definieren eine baumartige Stuktur aus Pages und Widgets. Jede Page besteht aus einer Collection von Widgets. Widgets können weitere Pages referenzieren. In der App wird immer jeweils eine Page angezeigt, die durch HTTP Long-Polling oder manuelles Laden oft aktualisiert wird. Die Klasse `WidgetListModel` hat eine Setter-Methode für eine Liste von Widgets. Danach stellt die Klasse fest, ob Widgets innerhalb

der Liste hinzugefügt, entfernt, verschoben oder sonst inhaltlich verändert wurden. Die Änderungen werden danach jeweils in Form eines `ListModificationEvent` auf einem ReactiveX Observable emittiert. User Interface Elemente (z.B. `RecyclerView` bzw. `RecyclerView.Adapter`) können dann auf dieses Observable subscriben und werden aktiv über Modifikationen an der Liste benachrichtigt, ohne dass das User Interface Element etwas über den Reload wissen muss. Für das User Interface unterscheidet sich dieses Design nur geringfügig vom klassischen Observer-Pattern.

```
1  package ch.hsr.baiot.openhab.sdk.model;
2
3  [...] // Imports
4
5  public class WidgetListModel {
6
7     [...] // Instance variables, getters
8
9     public void setWidgets(List<Widget> modified) {
10
11       List<Widget> original = new ArrayList<>(widgets);
12       List<Widget> state = new ArrayList<>(widgets);
13
14       List<Widget> changed = ListUtils.changed(original, modified);
15       notifyChanged(changed, original);
16       state = ListUtils.update(state, changed);
17
18       [...] // Detect removed, added, moves
19
20       widgets = modified;
21    }
22
23    public void notifyChanged(List<Widget> changed,
24                        List<Widget> original) {
25       for(Widget widget : changed) {
26          int index = original.indexOf(widget);
27          subject.onNext( new ListModificationEvent<Widget>(
28                widget,
29                ListModificationEvent.CHANGED,
30                index,
31                index
```

```
32              ));
33          }
34      }
35
36      [...] // notifyRemoved, notifyAdded, getMoves
37
38      public Observable<ListModificationEvent<Widget>>
39      onModification() {
40          return subject;
41      }
42  }
```

Listing 20: Auszug aus WidgetListModel.java

ch.hsr.baiot.openhab.app.adapter.WidgetListAdapter

Diese Klasse erbt vom Typ `RecyclerView.Adapter` und verbindet eine Collection von Widgets mit einer `RecyclerView`. Die RecyclerView fragt den Adapter nach der Anzahl von Elementen, lässt die Elemente erzeugen und sagt dem Adapter, dass er ein Element an das jeweilige Pendent in der Liste binden soll. Wenn sich die Liste im Adapter ändert, kann der Adapter der RecyclerView mitteilen, inwiefern sich die Liste verändert hat. Die RecylcerView fragt den Adapter danach wieder entsprechend ab. Dieses Pattern wird in vielen User Interface Frameworks zum Anzeigen von Listen verwendet, so auch in Android. Die Klasse `WidgetListAdapter` ist ein Subscriber des `WidgetListModel`, wie bereits im Abschnitt zuvor beschrieben. Jedoch kennt der Adapter das ListModel nicht direkt, sondern betrachtet dieses nur als ReactiveX Observable. Abbildung 19 zeigt die Abhängigkeiten zwischen den beteiligten Klassen:

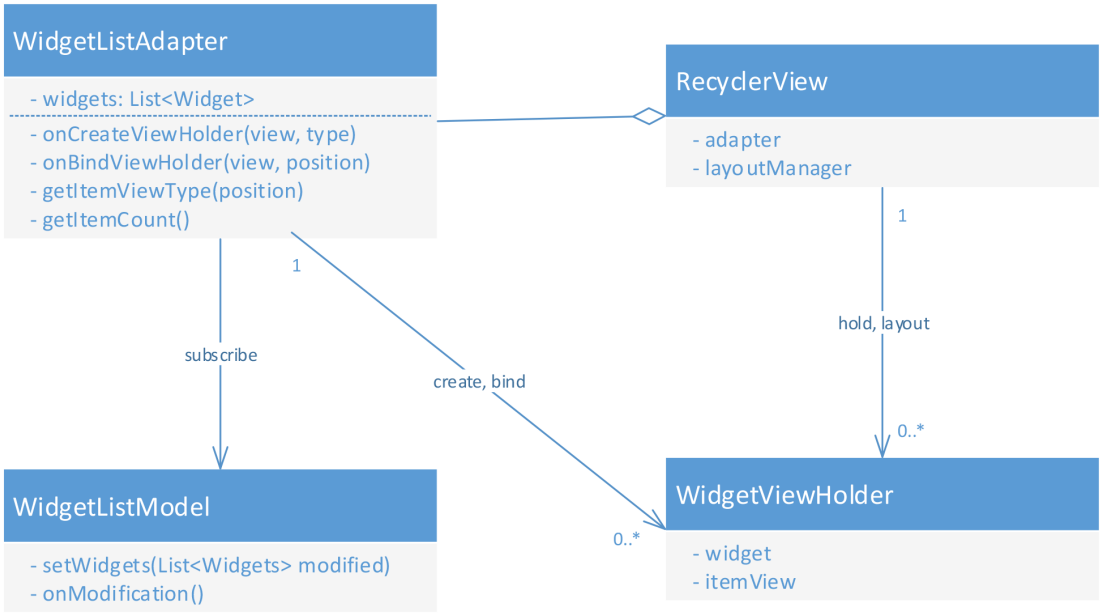


Abbildung 19.: Klassendiagramm WidgetListAdapter mit Collaborators

ch.hsr.baiot.openhab.app.widget.*

Die User Interface Elemente, die in der RecyclerView als Listeneinträge dargestellt werden, haben pro «Typ» jeweils ein eigenes XML Layout Resource File. Das Layout wird danach vom Android OS instanziiert und einem sogenannten `ViewHolder` zugewiesen. ViewHolder sind jene Objekte, die zwischen RecyclerView und Adapter ausgetauscht werden (siehe Abbildung 19). ViewHolder Objekte referenzieren die Layoutinstanz und werden im Adapter erzeugt. Das Einsetzen der Daten geschieht auf Anfrage der RecyclerView im Adapter und nennt sich «Binding». Im Package `ch.hsr.baiot.openhab.app.widget` befinden sich Subklassen von ViewHolder (`Widget*.java`). Die Subklassen werden benötigt, damit unterschiedliche Arten von Listeneinträgen in der RecyclerView dargestellt werden können. Beispielsweise solche mit Icon und Text, Detailtext oder solche mit einem Schalter und so weiter.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <RelativeLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     android:layout_width="match_parent"
5     android:layout_height="72dp">
```

```
6
7     <LinearLayout
8         android:orientation="vertical"
9         android:layout_marginTop="16dp"
10        android:layout_marginLeft="72dp"
11        android:layout_width="match_parent"
12        android:layout_height="wrap_content">
13
14        <TextView
15            android:id="@+id/text_view"
16            android:fontFamily="sans-serif-medium"
17            android:textColor="#000000"
18            android:textSize="16sp"
19            android:layout_width="match_parent"
20            android:layout_height="wrap_content" />
21
22        <TextView
23            android:id="@+id/text_detail"
24            android:fontFamily="sans-serif"
25            android:textColor="#000000"
26            android:textSize="14sp"
27            android:layout_width="match_parent"
28            android:layout_height="wrap_content" />
29
30    </LinearLayout>
31
32    <ImageView
33        android:id="@+id/icon"
34        android:src="@drawable/icon_dummy"
35        android:layout_marginLeft="16dp"
36        android:layout_marginTop="16dp"
37        android:layout_width="40dp"
38        android:layout_height="40dp" />
39 </RelativeLayout>
```

Listing 21: Layout mit Icon, Text und Detailtext

ch.hsr.baiot.openhab.app.activity.PageActivity

Die wichtigste Activity in unserer App ist die `PageActivity`. Sie stellt jeweils eine Page einer Sitemap dar. Für jede Hierarchiestufe in der Sitemap kommt eine neue Pa-

geActivity auf den Android Activity Stack. Die PageActivity besitzt unter anderem eine `RecyclerView`, einen `WidgetListAdapter` und ein `WidgetListModel`. Mit Hilfe der Klassen aus dem SDK wird die Page via REST-API geladen und die Widgets der Page an das WidgetListModel übergeben. Das WidgetListModel erkennt die Veränderung an der Liste von Widgets (zu Beginn war diese leer) und benachrichtigt die Subscriber über die Veränderungen. Der WidgetListAdapter ist ein solcher Subscriber, wodurch die Widgets letztlich in die RecyclerView gelangen. Die PageActivity koordiniert also das Zusammenspiel dieser Komponenten. Das mag etwas komplex erscheinen, jedoch ist somit eine hohe Kohäsion und eine geringe Kopplung erreicht, und der Code in den einzelnen Klassen ist leicht nachzuvollziehen. In der offiziellen openHAB App für Android sieht man was passiert, wenn versucht wird, die gleiche Aufgabe mit nur einer Klasse zu lösen. Das Ergebnis ist eine gigantische Klasse mit Methoden, die zweihundert Zeilen gerne überschreiten.

Wie schon erwähnt, lädt die PageActivity initial die gewünschte Page. Anschliessend soll der openHAB Server die App informieren, sobald sich etwas auf dieser Page verändert hat. Dazu verwendet die PageActivity den `LongPollingSocketClient` aus unserem SDK. Durch einen Bug von openHAB ist die Response des Long-Polling Requests oftmals leer, sodass die Page konventionell aktualisiert werden muss. Wenn der Long-Polling Request ein Timeout überschreitet, ohne eine Response erhalten zu haben, passiert dasselbe.

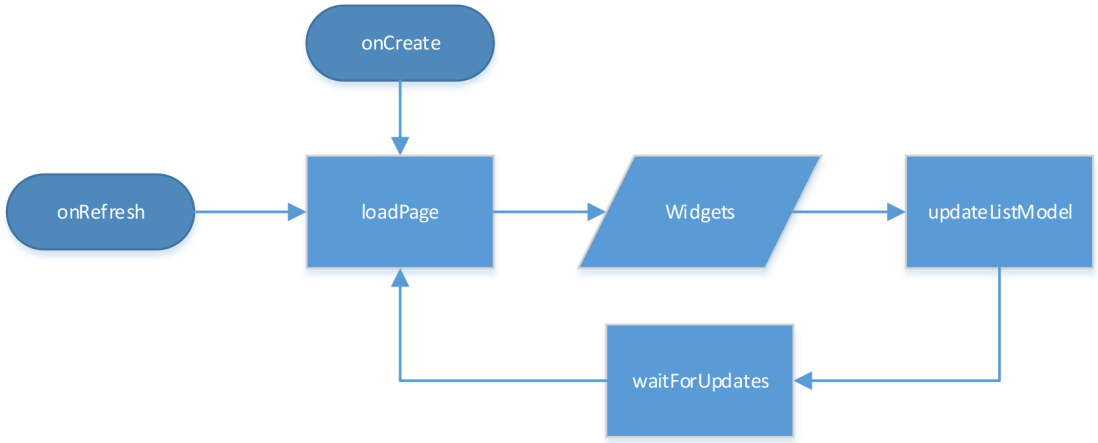


Abbildung 20.: PageActivity Ablauf (vereinfacht dargestellt)

In Abbildung 20 wird der Ablauf beim Laden anhand eines Flussdiagramms vereinfacht

aufgezeigt. In dieser Darstellung ist nicht ersichtlich, wie die PageActivity den Ablauf bei Fehlern oder beim Pausieren (im Sinne des Android Activity Lifecycle) unterbricht.

4.3.3. Material Design

Das App wurde nach den Prinzipien des Material Designs gestaltet. Google bietet dazu Guidelines an: `https://www.google.com/design/spec/material-design/introduction.html`. Insbesondere haben wir versucht, die strengen Rastervorgaben mit Hilfe von Margins und Paddings einzuhalten, damit für das Auge ein harmonischer Eindruck entsteht. Alle verwendeten Farben stammen aus den offiziellen Material Design Farbpaletten.

- Primary Color: Indigo 500 (`#3F51B5`)
- Primary Color Dark: Indigo 700 (`#303F9F`)
- Accent Color: Cyan 500 (`#00BCD4`)

Für die Darstellung unserer verschiedenen Komponenten wurden Card-Views eingesetzt. Cards werden verwendet, um einzelne Bereiche abzugrenzen. Sie dienen als Eintrittspunkt für weitere Informationen. In unserem Fall gibt es folgende Bereiche: Fenster, Lichter und Überwachung. Jeder dieser Bereiche enthält detaillierte Informationen und spezifische Aktionen, die ausgeführt werden können. Zum Beispiel beinhaltet der Bereich Überwachung die Information, ob der Bewegungsmelder ausschlägt, oder das Bild der Überwachungskamera kann betrachtet werden. Weiter kann der Alarm ein- bzw. ausgeschaltet werden.

Innerhalb dieser Cards wird eine List-View eingesetzt. Sie dient dazu, Elemente in Listenform übersichtlich darzustellen. In unserem Fall werden die einzelnen Elemente durch Trennstriche abgegrenzt.

Die einzelnen Screens sind im Anhang B, in der Benutzeranleitung der App zu finden.

4.4. Push-Notifications

Der Mobile-Client wird durch Push-Notifications benachrichtigt, sobald der Alarm ausgelöst wird. Umgesetzt wird dies mit Hilfe des Google Cloud Messaging Dienstes.

4.4.1. Cloud

Um den GCM-Dienst nutzen zu können, muss zuerst in der Google Developer Console (`https://console.developers.google.com`) ein Projekt erstellt werden mit einer Cloud-Messaging-API. Für dieses API kann ein Schlüssel erzeugt werden, der für das Versenden der Notification über das neu erstellte API benötigt wird.

Ausgelöst wird der ganze Vorgang durch die MQTT-Nachricht «alarm_activated». Die Worker Role nimmt diese Nachricht entgegen und erzeugt eine Notification-Message:

```
1 {
2   "registration_ids" : ["APA91bHun4MxP5egoKMwt2KZFBaFUH-1RYqx..."],
3   "data" : {
4    "message" : "Alarm wurde ausgeloest!"
5   }
6 }
```

Listing 22: Notification.cs - Notification Message

Die Registration-Id bezeichnet den Client, der die Message erhalten soll. Dazu müssen sich die Clients zuvor bei GCM registrieren und erhalten dann die Id. Diese Message muss also für jeden Client einzeln über einen HTTP-Post gesendet werden. GCM weiss anhand des Id-Strings, an welche Android-Clients er die Message weiterleiten soll.

Folgende Grafik stellt den Registrations-Vorgang dar:

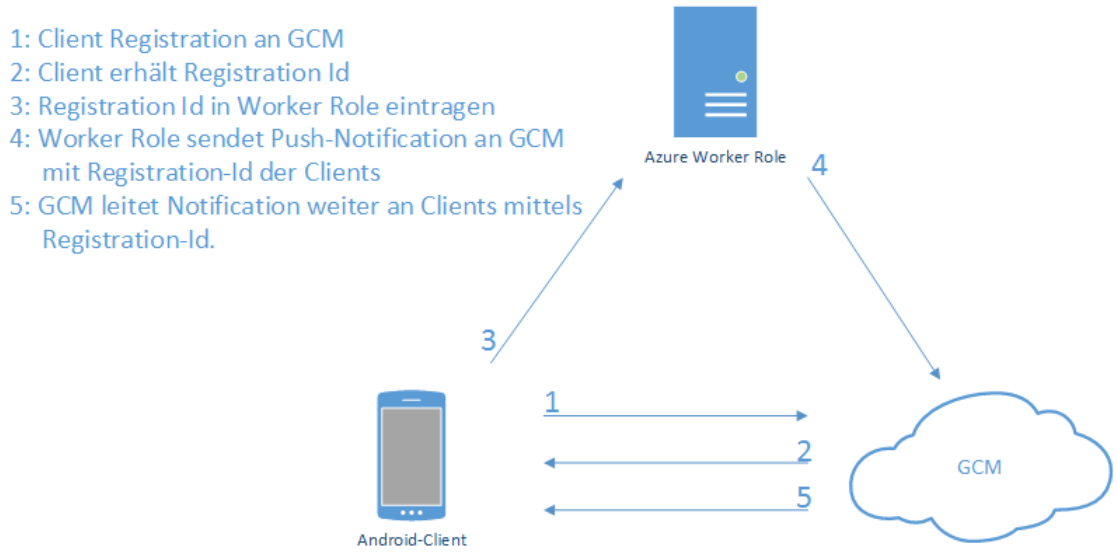


Abbildung 21.: Vorgang Push-Notification

Die nachfolgende Grafik (Abbildung 22) zeigt auf, wie anhand der MQTT-Message die Notification ausgelöst wird. OpenHAB bemerkt eine Statusänderung eines Items und weiss anhand der Persistency Konfiguration, dass das Item via MQTT an den Broker geschickt werden soll (Näheres dazu im Abschnitt 4.2.3 auf Seite 51). Die Worker Role empfängt die Message und entscheidet aufgrund deren Inhalt, ob eine Notification gesendet werden muss. Sollte dies der Fall sein, wird eine Push-Notification beim GCM-Dienst ausgelöst.

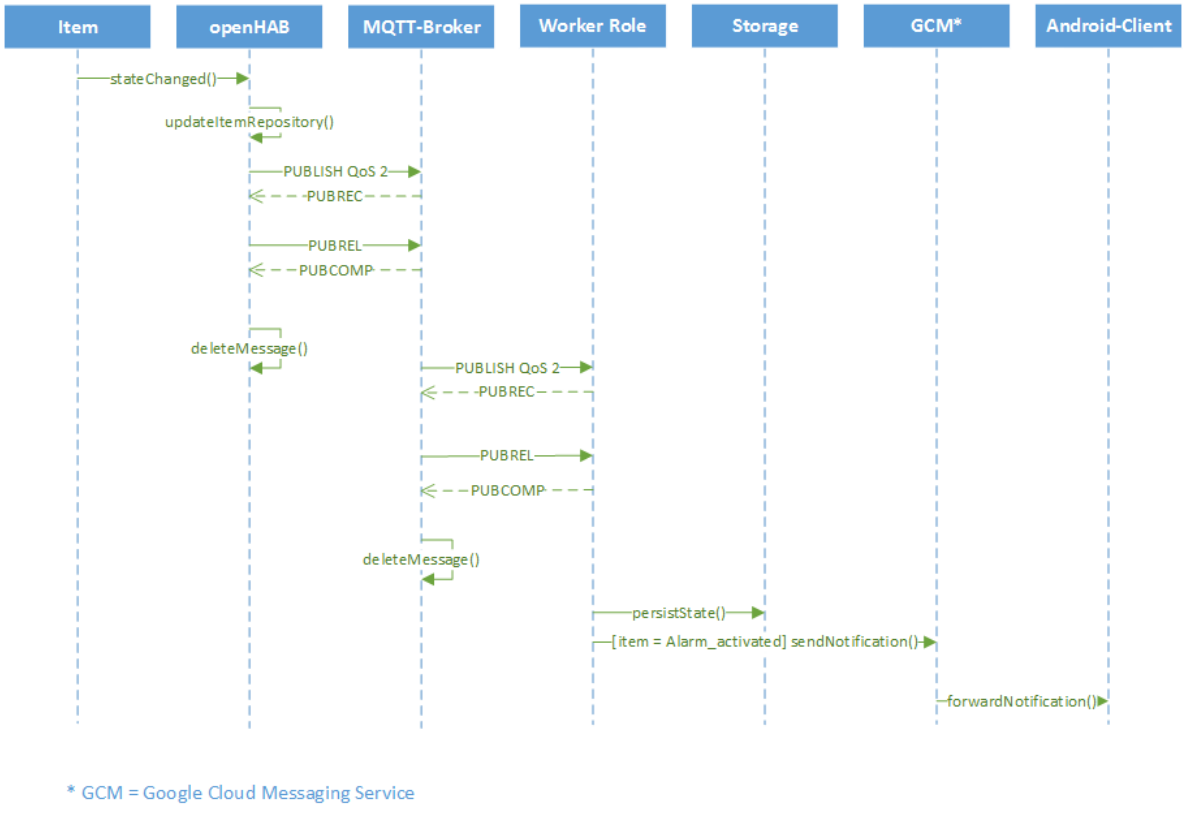


Abbildung 22.: Sequenzdiagramm MQTT-Notification

4.4.2. Android-Client

Wie bereits erklärt, benötigt die Worker Role eine Registration-Id, um das Android Phone referenzieren zu können. Die Registration-Id ist an ein GCM Projekt gekoppelt und wird in der Android App mit Hilfe des Google PlayServices SDK beim GCM-Dienst angefordert. Die Registration-Id muss dann in eigener Verantwortung an die Worker Role übertragen und geheim gehalten werden. Um eine Nachricht zu erhalten muss ein Android BroadcastReceiver im App-Projekt vorhanden sein, der die Nachricht entgegennimmt und zur Verarbeitung an einen IntentService weitergibt. Dort wird das Notification UI-Element (Dialog am oberen Bildschirmrand) erzeugt und angezeigt.

4.5. Sicherheit

Aus technischer Sicht können folgende Massnahmen getroffen werden, um die im Lösungskonzept beschriebenen Schwachstellen zu sichern:

- Verschlüsselung der MQTT-Verbindung.
- Anmeldung am MQTT-Broker durch Benutzername & Passwort.
- Verschlüsselung des WLANs.

4.5.1. Verschlüsselung MQTT-Verbindung

Die Verbindung zum MQTT-Broker wird durch SSL/TLS verschlüsselt. Dazu wird ein Self-signed Zertifikat eingesetzt. Wie dieses Zertifikat erzeugt wird, ist aus dem Anhang B zu entnehmen.
Die jeweilige Implementierung der TLS-Verbindung wird im Kapitel 4.2 auf Seite 48 erläutert.

4.5.2. Konfiguration MQTT-Broker

Am Broker sollen sich nur authentifizierte und autorisierte Clients anmelden können. Dazu bietet der MQTT-Standard die Möglichkeit, ein Password-File zu erzeugen. In dieses File wird der Benutzername und das zugehörige Passwort eingetragen, mit dem sich die Clients anmelden können. Nur wenn diese Angaben übereinstimmen, kann eine Verbindung aufgebaut werden.
Mosquitto bringt ein Tool mit zur Erzeugung und Verschlüsselung dieses Files. Über die Konsole wird das Skript aufgerufen, und die Benutzerangaben können als Parameter übergeben werden:

```
1 > mosquitto_passwd -c passwordFile
2 > mosquitto_passwd -b passwordFile username password
```

Listing 23: mosquitto_passwd.exe - generate password-file

Wie sich die Clients gegenüber dem Broker authentifizieren, ist im Abschnitt 4.2.3, auf Seite 51 (openHAB Client), und im Abschnitt 4.2.4, auf Seite 53 (Worker Role Client), ersichtlich.

4.5.3. Verschlüsselung WLAN

Über das WLAN kann man ohne Authentifizierung auf das openHAB zugreifen. Um unautorisierten Zugriff zu verhindern, wird das WLAN mit dem Sicherheitsstandard WPA2 verschlüsselt. Die Kommunikation wird also über den symmetrischen AES (Advanced Encryption Standard) Algorithmus verschlüsselt. Da es sich um eine symmetrische Verschlüsselung handelt, muss jeder Client ein PSK (Pre Shared Key) besitzen, der jeweils beim Verbindungsaufbau eingegeben werden muss.

4.6. Problematik Systemaufbau

In diesem Systemaufbau gibt es aufgrund der Infrastruktur der HSR einige Probleme, die es in einem richtigen Szenario nicht geben würde.
Da die Komponenten nicht in die Infrastruktur der Schule integriert werden durften, musste auf ein eigenes, lokales Netzwerk zurückgegriffen werden. Dabei handelt es sich um ein unabhängiges Netzwerk, auf das von externer Seite nicht zugegriffen werden kann. Dies stellt einige Herausforderungen an die Erreichbarkeit unseres Systems. Einerseits die Konnektivität mit der Cloud und andererseits das Erreichen von openHAB durch unseren Android Client.
Durch das Anbinden des Raspberry Pi an das lokale und das HSR-Netzwerk wird die Erreichbarkeit der Cloud gewährleistet. Es sind also zwei Network-Interfaces konfiguriert. Eines für das lokale und eines für das HSR Netzwerk. Damit die MQTT-Nachrichten an das richtige Interface gesendet wird, muss die Default Route auf das HSR-Interface (eth0) gesetzt werden: `ip route add default dev eth0`.

Da sich das Raspberry Pi jedoch nicht in der DMZ befindet, besteht keine Möglichkeit, die REST-Schnittstelle von openHAB von extern zu erreichen. Aus diesem Grund muss sich der Android Client zwingend im lokalen Netz befinden.
Aufgrund dieser Tatsache ist der Client nicht für Notifications erreichbar. Für Demonstrations-Zwecke muss zwischen dem HSR und dem lokalen Netzwerk gewechselt werden, um die Notifications zu erhalten.

5. Zusammenfassung und Ergebnisse

Die Aufgabenstellung forderte eine Lösung zur lokalen Vernetzung von Sensoren und Aktoren. Bestehende Systeme auf dem Markt sollten miteinander verglichen und als Lösungsansatz in Betracht gezogen werden. OpenHAB besitzt schon die Fähigkeit zum Vernetzen und intelligenten Steuern von Sensoren und Aktoren. Dadurch konnten wir uns auf den Aufbau eines Demo-Systems mit dem Fokus auf das Szenario Einbruchschutz konzentrieren.

OpenHAB als Basistechnologie einzusetzen erschien uns die geeignetste Wahl, denn es bietet genügend Flexibilät um das Szenario nach unseren Vorstellungen zu realisieren. Dank der Flexibilität konnten wir die Android App und die Cloud-Integration perfekt an das Szenario Einbruchschutz anpassen. Zusammengefasst entstanden während der Arbeit folgende Ergebnisse:

- Marktanalyse und Evaluation einer Plattform
- Versuchsaufbau mit echter Hardware
- OpenHAB Add-On für MQTT in Rules
- Cloud-Anwendung zum Sichern von Items
- Android App als openHAB Client

Mit openHAB liess sich die Hardware sehr gut in den Versuchsaufbau integrieren. Auch das Zusammenspiel der Hardware funktionierte meist einwandfrei. Man muss kein Software Ingenieur sein, um einen derartigen Versuchsaufbau nachzubilden. Etwas technisches Verständnis und Geduld wird aber vorausgesetzt.

Wir konnten mit dieser Arbeit zeigen, wie openHAB eingesetzt wird und wie ein eigenes Szenario darauf aufgebaut werden kann. Es wurde gezeigt, dass sich openHAB über das MQTT Protokoll mit der Azure Cloud verwenden lässt.

6. Persönliche Reflektion

Marco Leutenegger

Nach einer erfolgreichen Semesterarbeit durften wir die Bachelorarbeit wieder bei Herrn Prof. Hansjörg Huser machen, was mich persönlich sehr freute. Die Arbeit mit Herrn Huser war sehr interessant und die Zusammenarbeit stets unkompliziert, was ich sehr schätzte.

In der Semesterarbeit stand die Cloud bzw. unser Cloudservice im Fokus. Der Teil des Smart-Homes wurde dabei simuliert. In der Bachelorarbeit konnten wir uns nun diesem Bereich widmen. Dies stellte zu Beginn der Arbeit einige Herausforderungen dar. Einerseits weil ich mich bis jetzt noch nie mit der Entwicklung zusammen mit Hardware auseinandergesetzt habe. Andererseits war die Aufgabenstellung bezüglich des Szenarios sehr offen. Das heisst, wir mussten uns ein realistisches Konzept erarbeiten, wie Sensoren und Aktoren in einem Haus eingesetzt werden können. Des Weiteren mussten wir evaluieren, was für Technologien und Frameworks zur Heimautomation verwendet werden sollen. Dabei mussten wir auch beachten, wie kompatibel die Hardware mit den Frameworks ist.
Wir konnten uns dann auf das Szenario «Einbruchschutz» festlegen. Das entsprach auch meiner Vorstellung eines sinnvollen Einsatzes von Things, im Rahmen des Internet der Dinge. Wir durften uns darauf hin Hardware im Wert von ca. 500 Franken beschaffen, auf Kosten des INS. Das ist nicht selbstverständlich und spricht für das Vertrauen, das uns entgegengebracht wurde.

Das Highlight für mich war der Zeitpunkt, als das System aufgebaut war und die Item-Updates über MQTT im Storage der Azure Cloud erfolgreich persistiert wurden. Das Spezielle an dieser Lösung ist die Kombination der Cloud mit der openHAB Plattform.

Nach diesem Teil der Arbeit stand uns eine letzte grosse Hürde bevor, nämlich das Schreiben einer eigenen Android App. Da ich noch nie mit Mobileapplikationen gearbeitet habe, war das ganz neu für mich. Mit der Unterstützung von Dominik Freier, der bereits Erfahrungen in diesem Bereich hatte, fiel die Einarbeit in diese Thematik um einiges leichter und die Freude am fertiggestellten App war gross.

Ich kann nun auf eine gelungene und interessante Bachelorarbeit zurückblicken, die mir viel Spass bereitet hat.

Dominik Freier

Zu Beginn der Bachelorarbeit stand uns ein sehr breites Themengebiet offen. Wir hatten das Privileg, selbst mitbestimmen zu dürfen, wohin sich die Arbeit entwickeln soll und wo wir die Schwerpunkte setzen möchten. In der ersten Zeit entwickelten wir zusammen mit unserem Betreuer Herrn Prof. Hansjörg Huser das Smart-Home Szenario Einbruchschutz und definierten die genauen Anforderungen.

Nach einer Analyse des Marktes erkannten wir, dass es keinen Sinn machen würde selbst eine Lösung zum Integrieren und Kombinieren von Hardware zu entwickeln. OpenHAB passte am besten zu unseren Anforderungen, denn es hielt für alle wichtigen Kriterien einen Lösungsweg bereit. Andere Smart-Home Lösungen auf dem Markt sind zwar ausgereifter, sind aber zu geschlossen und geben das Szenario in den meisten Fällen vor. Bei der Auswahl von Hardware war mir wichtig, dass wir funktionierende Sensoren und Aktoren verwenden, die schon erprobt sind und in Privathaushalten eingesetzt werden. In unserer Arbeit konnten wir demonstrieren, wie sich Herstellergrenzen überwinden lassen. Ich bin der Meinung, dass unser Arbeitsergebnis eine gute Balance zwischen vorgegebenem Szenario und Flexibilität getroffen hat. Einen Mehrwert gegenüber einer reinen openHAB Installation haben wir durch die Cloud-Anbindung und die Android App geschaffen. Obwohl ich mit unserer Arbeit sehr zufrieden bin, möchte ich unser Demo-System nicht als Ersatz für eine professionelle Alarmanlage bezeichnen. Für den produktiven und verlässlichen Einsatz müssten weitere Randbedingungen und Situationen getestet und implementiert werden.

Für den Erfolg dieser Arbeit war die gute Zusammenarbeit zwischen Marco Leutenegger und mir ein entscheidender Faktor. Wir hatten von Anfang an dieselbe Vision und konnten unsere persönlichen und technischen Fähigkeiten ideal einsetzen. Durch paralleles Arbeiten und gute Planung hatten wir keine Stillstände. Trotzdem synchronisierten wir uns täglich über Fortschritte und überprüften die geleisteten Arbeiten gegenseitig.

Ich fand es etwas schade, dass wir unsere Lösung aufgrund der HSR Richtlinien nicht zu den gleichen Bedingungen wie in einem Privathaushalt aufbauen konnten. Wir konnten nicht ausprobieren, wie man per Remote auf den Versuchsaufbau zugreifen kann, obwohl dies für ein realistisches Szenario wichtig gewesen wäre.

Die Bachelorarbeit war eine sehr lehrreiche und spannende Erfahrung mit einem Themengebiet, das ich mir jederzeit erneut aussuchen würde.

Literaturverzeichnis

[1] openHAB, “openHAB documentation/wiki.” [Online]. Available: https://github.com/openhab/openhab/wiki

[2] “Mosquitto documentation.” [Online]. Available: http://mosquitto.org/documentation/

[3] “M2Mqtt documentation.” [Online]. Available: https://m2mqtt.wordpress.com/m2mqtt_doc/

[4] “Material design.” [Online]. Available: https://www.google.com/design/spec/material-design/introduction.html

[5] “Android developer.” [Online]. Available: http://developer.android.com/index.html

[6] “ReactiveX documentation.” [Online]. Available: http://reactivex.io/

[7] “Retrofit documentation.” [Online]. Available: http://square.github.io/retrofit/

[8] “ButterKnife documentation.” [Online]. Available: http://jakewharton.github.io/butterknife/

Glossar

AES	Advanced Encryption Standard
API	Application Programming Interface
BLOB	Binary Large Objects
CA	Certificate Authority
CN	Common Name
DMZ	Demilitarized Zone
DSL	Domain Specific Language
GCM	Google Cloud Messaging
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
JDK	Java Development Kit
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LAN	Local Area Network
MQTT	Message Queue Telemetry Transport
NFC	Near Field Communication
OSGi	Open Services Gateway Initiative
POJO	Plain Old Java Object
PSK	Pre Shared Key
QoS	Quality of Service
REST	Representational State Transfer
SDK	Software Development Kit
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UI	User Interface
URL	Uniform Resource Locator

Smart Home

Aufgabenstellung für Marco Leutenegger und Dominik Freier

Ziel
Aufbau einer Smart-Home Beispielapplikation (z.B. intelligente Lichtsteuerung und Storensteuerung, Türklingel, Einbruchüberwachung), welche wesentliche Aspekte einer Internet-of-Things-Anwendung demonstriert, wie Steuern von Devices, Lesen von Sensoren, Event-Verarbeitung, Überwachung und intelligente Abläufe steuern, Streaming von Sensordaten und Online-Analyse der Daten usw.
Das System soll auf einer tragfähigen und erweiterbaren Architektur aufgebaut werden und Microsoft Azure als Cloud Plattform benutzen.

Aufgabe
Folgende theoretischen und praktischen Themenbereiche sollen abgehandelt werden:

- Studium von Architektur-Referenzmodellen für IoT – Anwendungen:
  Beispiele:
  http://www.iot-a.eu/public
  http://iot6.eu/
  http://wso2.com/whitepapers/a-reference-architecture-for-the-internet-of-things/?utm_expid=96583979-25.XrJu5q-0RcShwB-AUhHx_g.0&utm_referrer=https%3A%2F%2Fwww.google.ch%2F
  Welche Erkenntnisse können für die vorliegende Arbeit verwendet werden?
- Analyse von bestehenden Frameworks für Smart-Home Anwendungen
  http://www.openhab.org/
  https://eclipse.org/smarthome/
  http://www.lab-of-things.com/
  Kann ein Framework für diese Arbeit eingesetzt werden? Lassen sich Konzepte/Ideen auf eine eigene Implementierung adaptieren?
- Definition eines Show-Case „Smart Home“ (oder aus einem anderen Anwendungsgebiet)
- Evaluation HW (Intel Galileo, Spark, Tinkerforge, Arduino, LightBlue Bean, Rasperry-PI, Netduino etc). Spezifikation HW-Bausteine (Bestellliste) sowie für das Demo-Objekt (Lego?)
- Konzeption und Implementation Smart-Home-Applikation mit .NET/Azure
  (Items, Scripts, Action, Rules analog openHAB für die Definition der Sensoren/Aktoren und den damit verbundenen Bindings und des Verhaltens).
- UI auf Mobile/Tablet: Generischer Aufbau aus der Item-Beschreibung (no-engineering) sowie Prozessbilder mit dynamischen Elementen.
- evtl. Event Processing und Analytics : Sensordaten streamen und auswerten

Resultate:

- Konzeptpapiere
- Lauffähiges System
- Dokumentation gemäss Richtlinien des Studiengangs Informatik

Projektteam

Marco Leutenegger, mleutene@hsr.ch
Dominik Freier, dfreier@hsr.ch

Betreuung HSR

Hansjörg Huser, hhuser@hsr.ch, Tel: 055 222 49 12 (HSR Raum 6.010)

Experte

Stefan Zettel, szettel@ascentive.ch, Ascentive Zürich

Gegenleser
Prof. Dr-Ing. Andreas Rinke

Projektabwicklung

Termine:

- Beginn der Arbeit: M**o., 16. Feb. 2015**
- Zwischenpräsentation für Gegenleser und evtl Experte: ca. Ende April.
- Abgabetermin Abstract für DA-Broschüre: **5 .Juni 2015**
- Abgabetermin Kurzfassung/Poster/Mgmt-Summary zum Review: : **5 .Juni 2015**
- Abgabetermin (inkl. Poster): **12. Juni 2015, 12.00 Uhr**
- Präsentation der Bachelorarbeiten, **12. Juni,16 bis 20 Uhr**
- Mündliche BA-Prüfung **Juli/Aug. 2015** : genaues Datum folgt
- Zwischenbesprechung/Review mit Auftraggeber nach Projektplan

Arbeitsaufwand

Für die erfolgreich abgeschlossene Arbeit werden 12 ECTS angerechnet. Dies entspricht einer Arbeitsleistung von mind. 360 Stunden pro Student. (14 Wochen zu ca. 21h, 2 Wochen zu 45h)

Hinweise für die Gliederung und Abwicklung des Projektes:

Gliedern Sie Ihre Arbeit in 4 bis 5 Teilschritte. Schliessen Sie jeden Teilschritt mit einem Meilenstein ab. Definieren Sie für jeden Meilenstein, welche Resultate dann vorliegen müssen!

Folgende Teilschritte bzw. Meilensteine sollten Sie in Ihrer Planung vorsehen:
Schritt 1: Projektauftrag inkl. Projektplan (mit Meilensteinen),

- Meilenstein 1: Review des Projektauftrages abgeschlossen. Projektauftrag von Auftraggeber und Dozent genehmigt
  Letzter Meilenstein: Systemtest abgeschlossen
  Termin: ca. eine Woche vor Abgabe
- Entwickeln Sie Ihre SW in einem iterativen, inkrementellen Prozess: Planen Sie möglichst früh einen ersten lauffähigen Prototypen mit den wichtigsten und kritischsten Kernfunktionen. In die folgenden Phasen können Sie dieses Kernsystem schrittweise ausbauen und testen.
- Falls Sie in Ihrer Arbeit neue oder Ihnen unbekannte Technologien einsetzen, sollten Sie parallel zum Erarbeiten des Projektauftrages mit dem Technologiestudium beginnen.
- Verwalten Sie ihre Software und Dokumente auf einem geeigneten Repository. Stellen Sie sicher, dass der/die Betreuer jederzeit Zugriff auf das Repository haben und dass das Projekt anhand des Repositories jederzeit wiederhergestellt werden kann.
- Achten Sie auf die Einhaltung guter Programmier- und Designprinzipien
- Halten Sie sich im Übrigen an die Vorgaben aus dem Modul SE-Projekt.

Projektadministration

- Führen Sie ein individuelles Projekttagebuch aus dem ersichtlich wird, welche Arbeiten Sie durchgeführt haben (inkl. Zeitaufwand). Diese Angaben sollten u.a. eine individuelle Beurteilung ermöglichen.
- Dokumentieren Sie Ihre Arbeiten laufend. Legen Sie Ihre Projektdokumentation mit der aktuellen Planung und den Beschreibungen der Arbeitsresultate elektronisch in einem Projektordner ab. Dieser Projektordner sollte jederzeit einsehbar sein (z.B svn-Server oder File-Share).

Inhalt der Dokumentation

Bei der Abgabe muss jede Arbeit folgende Inhalte haben:

- Dokumente gemäss Vorgabe: https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html (Dokument Ablaeufe_und_Regelungen_Studien-_und_Bachelorarbeiten_141027_.pdf)
- Aufgabenstellung
- Technischer Bericht
- Projektdokumentation
- Die Abgabe ist so zu gliedern, dass die obigen Inhalte klar erkenntlich und auffindbar sind.
- Zitate sind zu kennzeichnen, die Quelle ist anzugeben.
- Verwendete Dokumente und Literatur sind in einem Literaturverzeichnis aufzuführen.
- Projekttagebuch, Dokumentation des Projektverlaufes, Planung etc.
- Weitere Informationen: https://www.hsr.ch/Ablaeufe-und-Regelungen-Studie.7479.0.html

Fortschrittsbesprechung:

Regelmässig findet zu einem fixen Zeitpunkt eine Fortschrittsbesprechung statt.
Teilnehmer: Dozent und Studenten, bei Bedarf auch Vertreter der Auftraggeber
Termin: jeweils Mi. 10:10 bis 11h, Raum 6.010 (Abweichungen werden rechtzeitig kommuniziert)
Traktanden

- Was wurde erreicht, was ist geplant, welche Probleme stehen an
- Review von Code/Dokumentation (Abgabe jeweils einen Tag vor dem Meeting)

Falls notwendig, können weitere Besprechungen / Diskussionen einberufen werden.
Sie erstellen zu jeder Besprechung ein Kurzprotokoll, welches Sie spätestens 2 Tage nach der Sitzung per e-mail an den Betreuer senden.

Rapperswil, 17. Feb. 2014
Hansjörg Huser

B. Installationsanleitungen

openHAB Framework

Voraussetzungen:

- Java Installation
- openHAB Server
- Bindings

Installation openHAB Server

Die Installation wird gemäss openHAB-Wiki durchgeführt (`https://github.com/openhab/openhab/wiki/Linux---OS-X`). Dazu wird `apg-get` verwendet. Der Vorteil liegt darin, dass die entsprechenden Dateien automatisch an den richtigen Ort kopiert werden.

1. echo "deb http://dl.bintray.com/openhab/apt-repo stable main sudo tee /etc/apt/-sources.list.d/openhab.list
2. sudo apt-get update
3. sudo apt-get install openhab-runtime

Um den Server zu starten bzw. zu stoppen, können folgende Kommandos benutzt werden:

```
sudo /etc/init.d/openhab start
sudo /etc/init.d/openhab stop
sudo /etc/init.d/openhab restart
```

Konfiguration

Die ganze Konfiguration des Servers wird im File `openhab.cfg` vorgenommen. Das betrifft einerseits die Konfiguration des Servers, andererseits die Erfassung der Zentralen von HomeMatic und Philips Hue. Es müssen folgende Parameter gesetzt werden:

- mqtt-action:broker=mosquitto
- mqtt-persistence:broker=mosquitto
- mqtt-persistence:topic=openhab/items
- mqtt-persistence:message={"name":"%1$s", ßtate":"%3$s", "date":"%4$s"}
- mqtt:mosquitto.url = ssl://mqttbrokerba.cloudapp.net:8883
- mqtt:mosquitto.user=mosquitto
- mqtt:mosquitto.pwd=password
- mqtt:mosquitto.qos=2
- mqtt:mosquitto.lwt=openhab/health:openhaboffline:2:true
- hue:ip=192.168.1.110
- hue:secret=openHABRuntime
- homematic:host=192.168.1.137
- homematic:callback.host=192.168.1.139

Sitemap

Im File `sitemaps/demo.sitemap` wird die Sitemap definiert. Dort werden die verschiedenen GroupItems deklariert, die unsere Übersicht darstellt.

Rules

Die Regeln werden im File `rules/demo.rules` werden die Regeln definiert. Das beinhaltet die Logik, was passieren soll wenn ein Status update eintrifft. Dient zur Automatisierung.

Addons

Addons sind die verschiedenen Module, die bei Gebrauch heruntergeladen werden können. Alle vorhandenen Addons können durch `sudo apt-cache search openhab` aufgelistet werden.
Nach dem Schema `sudo apt-get install openhab-addon-${addon-type}-${addon-name}` können diese heruntergeladen werden. Wir benötigen folgende Addons:

- homematic
- http
- hue

Web-GUI

OpenHAB bietet von sich aus bereits ein GUI in Form einer Webapplikation zur Verfügung. Dazu kann über den Browser auf das System zugegriffen werden: `http://localhost:8080/openhab.app?sitemap=demo`.
In der Abbildung 23 ist das GUI abgebildet.

Abbildung 23.: Web-GUI openHAB

Zertifikat Erzeugen

Die Zertifikate wurden mit OpenSSL (`http://slproweb.com/products/Win32OpenSSL.html`) erzeugt.

CA Zertifikat erstellen

```
openssl req -new -x509 -days 3650 -keyout m2mqtt_ca.key -out m2mqtt_ca.crt
```
Dieser Command erzeugt ein CA (Certificate Authority) Zertifikat mit einem privatem Schlüssel. Das X.509-Zertifikat ist für 3650 Tage gültig.

Server Zertifikat erstellen

```
openssl genrsa -des3 -out m2mqtt_srv.key 1024
```
Erzeugt einen 1024 Bit grossen 3DES, privaten Schlüssel.

Signierung des Server Zertifikates

```
openssl req -out m2mqtt_srv.csr -key m2mqtt_srv.key -new
```
Dieses Kommando erzeugt ein Zertifikat-Request vom Server zur Signierung an die CA. Durch dieses Kommando wird lediglich der Request erzeugt. Das bedeutet, dass die Signierung noch nicht durchgeführt wurde.

```
openssl x509 -req -in m2mqtt_srv.csr -CA m2mqtt_ca.crt -CAkey m2mqtt_ca.key
-CAcreateserial -out m2mqtt_srv.crt -days 3650
```

Mit diesem Kommando wird der Server-Request signiert. Als Produkt entsteht das finale Zertifikat für den Broker.

Für die Client-Library (M2Mqtt) muss das Zertifikat noch ins DER-Format konvertiert werden:
```
openssl x509 -outform der -in m2mqtt_ca.crt -out m2mqtt_ca.der
```

Android App - Hawk Eye

Diese Android App bietet einen Zugang zum openHAB System. Dadurch können die Status der verschiedenen Sensoren und Aktoren abgefragt werden. Ebenfalls kann über ein Schalter die einzelnen Glühbirnen ein- oder ausgeschaltet werden. Nachfolgend werden alle Screens gezeigt und erklärt.

Die Abbildung 24 zeigt den Home-Screen. Durch Tippen auf «Einrichten» gelangt man zu den Einstellungen. Durch Tippen auf den Pfeil gelangt man zur Übersicht. Dies ist jedoch nur möglich, wenn die Verbindungseinstellungen korrekt sind und eine Verbindung zum openHAB hergestellt werden kann.

Abbildung 24.: Start Screen

Die Abbildung 25 zeigt die Einstellungen. Dazu muss die openHAB-URL (IP-Adresse des Raspberry Pis mit Port 8080) und der Name der Sitemap eingetragen werden. Durch Tippen auf «Verbindung Testen» können die Eingaben geprüft und die Konnektivität getestet werden.

Des weiteren kann die Notification ein- und ausgeschaltet werden. Durch Tippen auf den Haken werden die Einstellungen gespeichert und die View geschlossen.

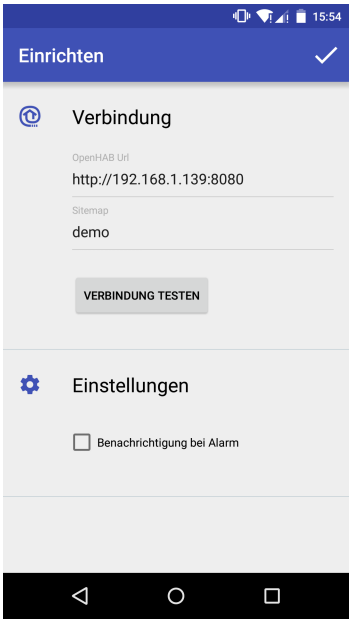


Abbildung 25.: Config Screen

Wenn die Verbindungseigenschaften korrekt sind, wird das Sitemap dargestellt (Abbildung 26). Da sind die drei Bereiche zu sehen. Durch Tippen auf das Element gelangt man in die Detailansicht des entsprechenden Bereiches. Je nach Status wird dabei ein anderes Icon gezeigt. Falls zum Beispiel die Lichter ausgeschaltet sind, ändert die Farbe des Icons auf grau (inaktiv).

In der Actionbar befindet sich rechts ein Zahnrad. Durch antippen dieses Symbols gelangt man zu den Einstellungen.

Um die Seite zu aktualisieren und die Status neu zu laden, kann mit dem Finger von oben nach unten gezogen werden (swipe).

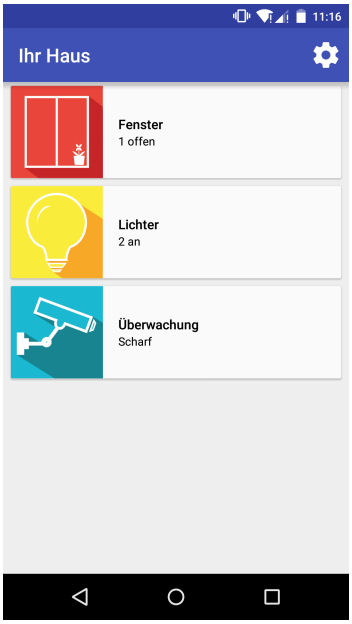


Abbildung 26.: Alarm Beispiel

Diese Abbildung zeigt die Detailansicht des Bereiches «Fenster». Auch hier ändert sich jeweils das Icon, falls sich der Status des Items ändert (siehe Abbildung 28). Durch Tippen auf den Back-Button von Android, gelangt man zurück zur Übersicht.

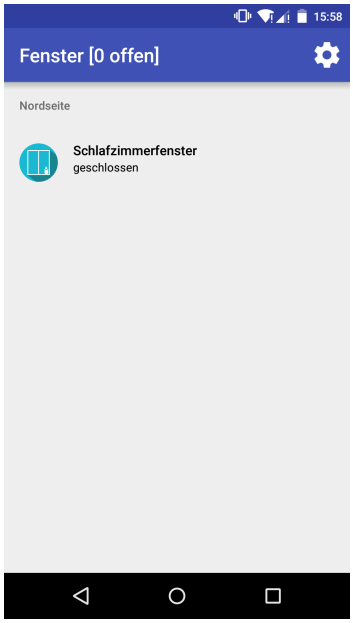


Abbildung 27.: Window Item Screen

Das Icon ändert auf rot, wenn mindestens ein Fenster geöffnet wurde.

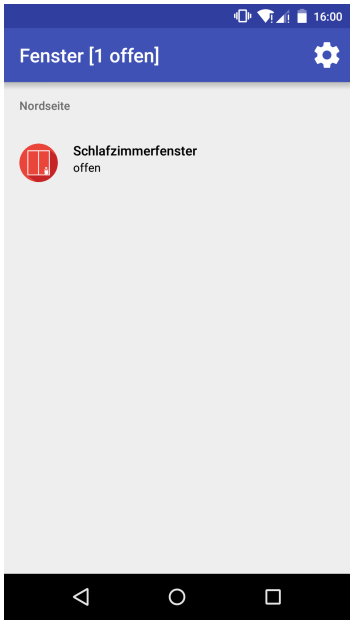


Abbildung 28.: Alarm Item Beispiel

In der Abbildung 29 wird die Detailansicht des Bereiches «Licht» gezeigt. Die einzelnen Lampen können über den Schalter ein- und ausgeschaltet werden. Je nach dem ändert sich auch hier das Icon.

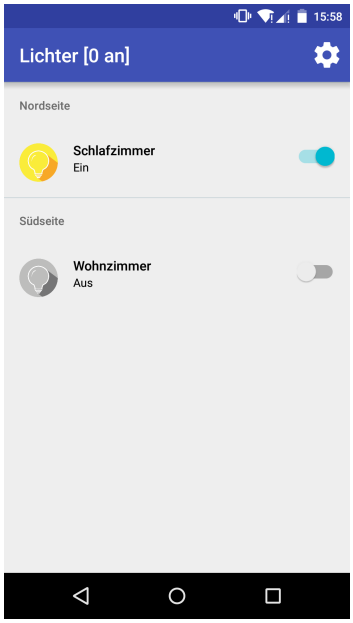


Abbildung 29.: Light Item Screen

Der Bereich Überwachung ist das Kernstück der App. Hier kann die Alarmbereitschaft ein- und ausgeschaltet werden. Wenn der Alarm ausgeschaltet ist, wird man weder benachrichtigt, noch ändern sich die Icons, wenn sich ein Status ändert.
Weiter wird hier die Information gegeben, ob eine Bewegung detektiert wurde.

Im unteren Bereich ist die Kamera zu sehen. Durch Scrollen sieht man das ganze Bild.

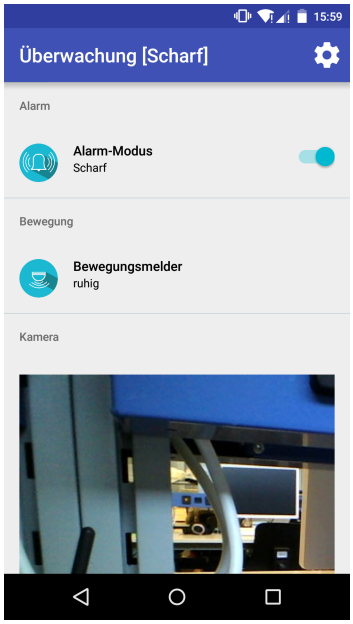


Abbildung 30.: Monitoring Screen