technische universität
dortmund

Masterarbeit

FLASH: Ein on-premise Workflow Automation Service im Kontext Smart Home – Prototypische Implementierung basierend auf dem Eclipse Open IoT Stack

Konstantin Tkachuk
Februar 2017

Gutachter:
Prof. Dr.-Ing. Olaf Spinczyk
Dr. Sven Seiler

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
http://ls12-www.cs.tu-dortmund.de

In Kooperation mit:
Materna GmbH

Zusammenfassung

Das sich gerade stark weiterentwickelnde Gebiet des Internets der Dinge ist derzeit noch sehr fragmentiert. Zwei bis dato völlig unabhängige Bereiche von IoT sind Smart Home und webbasierte Task Automation Services. Dennoch weisen sie viele Gemeinsamkeiten auf und ein Verschmelzen der beiden Ansätze verspricht Vorteile in Aspekten, wie Reaktionszeiten, Durchsatz und Sicherheit.

Im Rahmen dieser Arbeit gilt es diese Annahmen in der Praxis zu prüfen. Zunächst sollen die Gemeinsamkeiten und Unterschiede der Ansätze im Detail betrachtet werden, wonach konkrete Anforderungen an einen Prototypen zu formulieren sind. Daraufhin wird ein kurzer Einblick in den Eclipse Open IoT Stack gegeben, auf dessen Basis der Prototyp entworfen und implementiert wird. Anschließend wird er einem umfangreichen Test unterzogen, wodurch die zugrundeliegenden Annahmen bestätigt oder widerlegt werden sollen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Im Laufe der Zeit ist deutlich geworden, dass eingebettete Systeme eine zunehmend wichtigere Rolle im alltäglichen Leben des Menschen spielen. Nahezu jedes elektrische Gerät enthält heutzutage eingebettete Rechnerhardware und lässt sich aus der Entfernung steuern. Es hat sich der Begriff Internet der Dinge (IoT) geprägt.

Ein aktuelles und sich gerade stark weiterentwickelndes Gebiet des Internets der Dinge ist Smart Home. Darunter versteht man den Einsatz von von Sensoren und Aktoren zur Automatisierung des Verhaltens der intelligenten Geräte im Hause des Endnutzers. Die dadurch gewonnenen Komfort, Sicherheit und Flexibilität bieten eine Unterstützung für den Nutzer, die schwierig zu überschätzen ist.

Das Internet hat sich über wenige Jahrzehnte von einer neuartigen Idee zu einem der Grundpfeiler der Welt entwickelt. Die Anzahl der existierenden Webservices ist nicht mehr überschaubar. Ein durchschnittlicher Mensch nutzt dutzende Dienste täglich, oftmals in sich ständig wiederholenden Szenarien. Diese Szenarien zu automatisieren versprechen webbasierte Task Automation Services (TAS). Unter TAS versteht man Dienste, die es dem Nutzer ermöglichen Regeln für die Interaktionen zwischen unterschiedlichen Webservices zu definieren. Daraufhin werden diese spezifizierten Interaktionen von dem TAS übernommen.

Webbasierte Task Automation Services sind bis dato völlig unabhängig von Smart Home Lösungen. Gleichzeitig realisieren sie dieselben Kernideen auf eine vergleichbare Weise, nur in einem anderen Umfeld. Dennoch wurden bisher keine Versuche unternommen, diese unterschiedlichen Ansätze in einem gemeinsamen Kontext zu verschmelzen. Dies ist der Punkt, an dem diese Arbeit ansetzen soll.

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist es die Welten von Smart Home und webbasierten Task Automation Services zusammen zu bringen. Es soll ein Demonstrator entwickelt werden, der die Funktionalitäten beider Ansätze kombiniert und in einem gemeinsamen Kontext anbietet. Konkreter sind die Ziele in Sektion 2.2.1 erläutert.

1.3 Aufbau der Arbeit

Nach der Einleitung wird der aktuelle Stand der Forschung bezüglich *Internet der Dinge*, *SmartHome* und *Task Automation Services* vorgestellt und die Ziele der Arbeit daraus abgeleitet. Daraufhin wird in Kapitel 3 das Framework Eclipse SmartHome (ESH) vorgestellt. In Kapitel 4 wird konkret betrachtet, wie die Funktionalitäten eines Task Automation Services in ESH integriert werden können.

Im folgenden Kapitel 5 wird die Implementierung des Entwurfs mit Hilfe von entsprechenden Diagrammen vorgestellt. Der Quell-Code ist auf der mit der Arbeit mitgelieferten CD einsehbar. Danach wird in Abschnitt 6 die Implementierung evaluiert. Schließlich folgt noch eine kurze Zusammenfassung der geleisteten Arbeit und ein Ausblick auf mögliche Weiterentwicklungen in Kapitel 7.

Kapitel 2

Stand der Forschung

In diesem Kapitel wird der aktuelle Stand der Forschung des Internets der Dinge erläutert. Es wird besonders auf die Themen Smart Home und Task Automation Services eingegangen, sowie darauf, welche Vorteile sich durch das Kombinieren dieser Ansätze erhoffen lassen.

2.1 Internet of Things

Die enorm steigende Anzahl von „intelligenten Gegenständen“ mit eingebetteten Computern, die den Menschen im alltäglichen Leben unterstützen sollen, hat zu der Prägung des Begriffs „Internet der Dinge“ (IoT) geführt. Jedes dieser Dinge hat seine eigene Funktionalität und im Verbund stellen sie eine große Menge an Daten zur Verfügung. Im Rahmen von zahlreichen Forschungsprojekten [26] werden Möglichkeiten untersucht, IoT mit unterschiedlichen Technologien zu kombinieren. Unter anderem werden Technologien, wie Cloud Computing[23], Machine-2-Machine Learning[28] und Semantic Web[29] kritisch betrachtet. Außerdem werden Kernprinzipien, wie Architektur und Standardisierung intensiv recherchiert. Sie werden in dedizierten Forschungsprojekten [31][30] immer wieder aufgegriffen.

Im Laufe der Zeit haben sich verschiedene Aspekte des IoT herausgebildet, unter anderem das Smart Home.

2.1.1 Smart Home

Ein mit IoT eng verwobenes Thema ist das Smart Home[33], welches die elektronische Steuerung von ausgewählten Geräten mit z.B. einer Rule Engine kombiniert um eine Automatisierung des Geräteverhaltens in einem Zusammenspiel zwischen Sensorik und Aktortik zu erreichen. Smart Home grenzt sich von IoT ab indem es auf Sensoren und Aktoren spezialisiert ist, die im Kontext eines Hauses relevant sind.

Bis dato wurden zahlreiche Smart Home Lösungen von verschiedenen Anbietern entwickelt. Man kann prinzipiell zwei Arten von Lösungen unterscheiden. *Proprietäre* Produkte (z.B. *RWE SmartHome*[14]) spezialisieren sich auf eine sehr begrenzte Anzahl von Geräten und bemühen sich maximale Unterstützung für diese Geräte zu bieten. Dies sorgt für eine Fragmentierung des Marktes. *Open Source* Lösungen hingegen verfolgen das Ziel möglichst offen für verschiedene Geräte und Protokolle zu bleiben.

Da Open Source Lösungen aktuell noch aktiv in der Entwicklung sind und hauptsächlich als Software existieren, erfordert ihr Einsatz von Nutzern ein gewisses Maß an technischen Vorkenntnissen. In der Regel muss der Nutzer passende Hardware bereitstellen und die Software selbst installieren. Bei Problemen kann Support bestenfalls auf Foren von der Community gefunden werden.

Diese Gründe führen dazu, dass Open Source Lösungen derzeit noch nicht massenhaft zum Einsatz kommen.

2.1.2 Task Automation Services

Die Automatisierung von Aufgaben ist eins der zentralen Bestreben unseres alltäglichen Lebens. Es macht das Leben einfacher und erlaubt uns kostbare Zeit zu sparen. Ob Notifikation auf dem Smartphone, wenn eine Email eingeht oder das Einschalten von Lampen, wenn ein Raum betreten wird, solche Automatisierung ist heutzutage überall zu finden. Lange Zeit musste jede derartige Automatisierung einzeln entworfen, konfiguriert und implementiert werden. Doch die steigende Anzahl von intelligenten Gegenständen und die Allgegenwärtigkeit des Internets belassen dies der Vergangenheit. Nun hat sich der Ansatz der Task Automation Services[25] gebildet.

Ein Task Automation Service (TAS) ist ein Dienst, der es Endnutzern ermöglicht das Verhalten von verschiedenen Services und Geräten in eigenen Szenarien jederzeit selbst zu automatisieren. Solche Szenarien basieren auf Event-Condition-Action (ECA) Regeln[21], welche es ermöglichen, auf Events unter festgelegten Bedingungen mit entsprechenden Aktionen zu reagieren. Meistens wird dies durch einen intuitiven visuellen Regel Editor ermöglicht.

Aktuell gibt es noch vergleichsweise wenige TAS. Ein Überblick über existierende Services bietet Abbildung 2.1.

Wie in der Abbildung zu sehen ist, gibt es unterschiedliche Ansätze. Einige TAS sind in der Cloud angesiedelt, was bedeutet, dass sie, sofern Internet verfügbar ist, jederzeit und von überall erreichbar sind. Die aktuell mächtigsten und bekanntesten TAS sind *IFTTT* [17] und *Zapier*[19]. Sie unterstützen hunderte unterschiedlicher Web Services, bieten aber keine Möglichkeit mit Geräten direkt zu interagieren. Ihr Fokus ist es zu ermöglichen eine Vielzahl von Szenarien auf eine einfache Art und Weise zu erstellen. Auf komplexere Regeln

		Web						Smartphone				Home	
		Ifttt	Zapier	CloudWork	Elastic.io	ItDuzzit	Wappwolf	On{x}	Tasker	Atooma	Automatelt	WigWag	Webee
Channels	Web channel support	✓	✓	✓	✓	✓	Few	Few	✓	✓	✓	✓	✓
	Device channel support	Few	×	×	×	×	×	✓	✓	✓	✓	✓	✓
	Smartphone resources as channels	✓	×	×	×	×	×	✓	✓	✓	✓	×	×
	Public channels support	✓	×	×	✓	✓	×	✓	×	×	×	×	×
	Pipe channel support	×	×	×	✓	×	Few	Few	×	×	×	×	×
	Group channel support	×	×	×	×	×	×	×	×	×	×	Few	×
	Device channel discovery	×	×	×	×	×	×	×	×	×	×	✓	Few
Rules	Multi-event rules	×	×	×	×	×	×	✓	✓	×	×	✓	×
	Multi-action rules	×	×	×	✓	×	×	✓	✓	×	✓	✓	✓
	Chain rules	×	×	×	✓	×	Few	Few	×	×	×	×	×
	Group rules	×	×	×	×	×	×	×	×	×	×	Few	×
	Collision handling	×	×	×	×	×	×	×	×	×	×	×	×
	Predefined common rules	×	×	✓	×	✓	✓	×	×	×	✓	✓	✓
	Rule execution profile	WD	WD	WD	WD	WD	WD	DD	DD	DD	DD	DD	DD
TAS	Visual rule editor	✓	✓	×	✓	✓	✓	×	✓	✓	✓	✓	✓
	Provides API	×	✓	×	✓	×	×	×	×	×	×	×	×
	Programming language	×	×	×	✓	×	×	✓	Few	×	✓	✓	✓

* ✓ = supported; × = not supported; Few = few support; WD = Web-driven execution profile; and DD = device-driven execution profile.

Abbildung 2.1: Überblick über existierende Task Automation Services [25]

und Szenarien sind ihre Rule Engines nicht ausgelegt. Um diese TAS zu nutzen, muss man jedoch bereit sein, sämtliche Zugriffsrechte, die für die zu automatisierenden Dienste (z.B. *Facebook*, *Twitter*, etc.) benötigt werden, dem Cloud Service anzuvertrauen.

Andere Task Automation Services arbeiten lokal auf Smartphones. Solche TAS konzentrieren sich auf die Automatisierung von den auf dem Gerät laufenden Services. Die Unterstützung von der Automatisierung von Web Services ist nur in dem Umfang gegeben, in dem diese Web Services direkten Kontakt mit dem Smartphone haben.

Schließlich gibt es TAS, die auf einer dedizierten Basis im Haus arbeiten. Solche Task Automation Services konzentrieren sich auf die direkte Steuerung von Geräten mithilfe der entsprechenden Protokolle. Im Grunde sind sie äquivalent zu Smart Home.

Im Rahmen dieser Arbeit wird Smart Home jedoch getrennt von TAS behandelt, da der Fokus der Automatisierung völlig unterschiedlich ist. Mehr dazu in Sektion 2.2.

IFTTT

Ein Beispiel für Task Automation Services in der Cloud ist IFTTT, welches eine große Anzahl verschiedener Services (*Facebook*, *Dropbox*, *Philips Hue*[11], etc.) integriert und eine rudimentäre Rule Engine anbietet, die es erlaubt auf eine einfache Art und Weise

serviceübergreifende „if this than that“ Anweisungen zu hinterlegen, die der klassischen „EDV“ ähneln. Es wird eine feste Trigger Komponente gewählt, die etwas auslöst, daraufhin wird die Aktion festgelegt, die ausgeführt werden soll. Solche Condition/Command Paare werden in IFTTT *Recipes* genannt.

Bis dato lassen sich komplexere Szenarien mit *IFTTT* nicht abbilden. Das bedeutet, dass es keine Möglichkeit gibt, beispielsweise, Und- und Oder-Bedingungen zu definieren, die es ermöglichen würden, mehrere Conditions in einem Recipe zu verknüpfen.

Ein weiterer Aspekt von IFTTT ist, dass es für die Anbindung von Services Zugriffsrechte auf sämtliche Accounts des Users bedarf. Aus einer Datenschutz-Perspektive[22] stellt das ein Risiko für den Endnutzer dar, da seine sämtlichen Account-Daten an einer Stelle gesammelt sind. Im Falle einer Sicherheitslücke bei IFTTT wären alle damit gekoppelten Services in Gefahr.

2.2 Zentrale Idee

Wie in Sektion 2.1.2 zu sehen ist, kann man drei prinzipiell unterschiedliche Arten von Task Automation Services unterscheiden: Cloud Service, Smartphone Applikation und Smart Home. Sie weisen zwar viele Gemeinsamkeiten auf, doch allein aus den Namen lässt sich ablesen, dass es sich dabei um grundlegend verschiedene Lösungen handelt.

Die Cloud Services unterstützen nur begrenzt die „direkte“ Steuerung von Geräten. Außerdem sind die zum Einsatz kommenden Rule Engines sehr rudimentär. Smart Homes hingegen haben in der Regel mächtige Rule Engines und bieten die Möglichkeit komplexe Szenarien zu definieren. Zusätzlich sind sie in der Lage, Geräte direkt anzusteuern. Im Rahmen dieser Arbeit werden die Smartphone Applikationen nicht näher betrachtet.

Die zentrale Idee dieser Arbeit ist es ein Smart Home mit einem Cloud Service TAS zu verschmelzen. Dadurch soll ein Ganzes entstehen, welches mehr ist, als die Summe seiner Teile: Ein (Work-)Flow Automation Service im Kontext Smart Home (FLASH).

2.2.1 Ziele

Ziel dieser Arbeit ist es die Welten von Smart Home und webbasierten Task Automation Services zusammen zu bringen. Durch die Integration mit Smart Home soll die direkte Steuerung von Geräten befähigt werden. Außerdem soll die Mächtigkeit der Smart Home Rule Engine genutzt werden, um komplexere Regeln und Szenarien zu ermöglichen. Schließlich sollen die Datenschutzprobleme gelöst werden, indem sämtliche Zugriffsdaten on-premise galagert und dadurch nie einem Risiko ausgesetzt werden.

Es soll ein Demonstrator entwickelt werden, der die Funktionalitäten von Smart Home und TAS kombiniert und in einer on-premise Anwendung anbietet. Hierzu soll das Eclipse

SmartHome Framework als Basis verwendet und um Funktionalitäten von webbasierten TAS angereichert werden. Die entstandene Anwendung soll auf einem Raspberry Pi[13] laufen.

Die entstandenen Funktionalitäten sollen anhand von einer Reihe von konkreten Services demonstriert werden. Unter anderem sollen ein Wetterdienst, ein Filesharing Service und ein Social Media Service angebunden werden. Die Zusammenarbeit dieser Services untereinander und mit Smart Home Geräten soll anhand von Beispiel-Regeln demonstriert werden. Außerdem soll es möglich sein neue Regeln zum System über eine Benutzeroberfläche hinzufügen zu können.

Zum Schluss soll der entstandene Demonstrator evaluiert werden. Es soll geprüft werden, inwiefern Task Automation Services als on-premise Lösung sinnvoll sind unter Betrachtung von Aspekten wie Reaktionszeiten und Netzwerklast. Hierzu soll ein Vergleich der erstellten Anwendung mit *IFTTT* stattfinden.

2.2.2 Konkrete Anforderungen

In dieser Sektion werden aus den oben genannten Zielen konkrete Anforderungen an den resultierenden Demonstrator abgeleitet und im Detail festgehalten.

Es wird erwartet, dass der Demonstrator folgende Eigenschaften erfüllt:

1. Der Demonstrator soll intern wie eine Smart Home Zentrale agieren und in der Lage sein, Geräte innerhalb des Hauses auch ohne Internetverbindung steuern zu können.
2. Bei vorhandener Internetverbindung soll er in der Lage sein ausgewählte webbasierte Dienste zu automatisieren.
3. Sämtliche (Zugriffs-)Daten sollen lokal auf dem Gerät gelagert werden.
4. Die Webdienste sollen nahtlos in die Rule Engine integriert werden, sodass komplexe Szenarien ermöglicht werden.
5. Es soll eine grafische Benutzeroberfläche angeboten werden, in der der Nutzer in der Lage ist eigene Szenarien zur Laufzeit zu definieren.
6. Er soll on-premise auf einem Raspberry Pi 3 Model B laufen.

Anzubindende Services

Im Rahmen der Arbeit sollen folgende populäre Internetdienste in den Demonstrator exemplarisch integriert werden:

1. **Twitter**[15] Der Demonstrator soll in der Lage sein für den Nutzer Tweets zu schreiben, sowie auf Tweets zu reagieren. Beispielsweise soll es möglich sein, Medien in Tweets automatisch in die Dropbox zu speichern.

2. **Dropbox**[4] Es soll möglich sein, automatisch Dateien in die Dropbox zu speichern, sowie auf das Ändern von existierenden Dokumenten zu reagieren.

3. **Wetterdienst**[10] Es sollen Wetterdaten aus dem Internet bezogen und auf konkrete Wetterbedingungen reagiert werden können.

4. **Email** Der Demonstrator soll in der Lage sein an beliebige Adressen Emails zu versenden.

Kapitel 3

Eclipse Smarthome

In diesem Kapitel wird *Eclipse SmartHome* (ESH) [5] vorgestellt. Es werden zunächst die grundlegenden Konzepte des Frameworks erläutert. Anschließend wird auf für die Arbeit relevante konkrete Aspekte näher eingegangen.

3.1 Überblick

Eclipse SmartHome positioniert sich als ein Framework, dass als Grundlage für die Entwicklung von konkreten SmartHome Endlösungen dienen soll. Tatsächlich kommt es in vielen bekannten Produkten zum Einsatz, wie z.B. openHAB[9] und Qivicon[12].

Architektur

Eclipse SmartHome hat eine serviceorientierte Architektur, basierend auf dem Java OSGi Framework[20]. OSGi spezifiziert eine dynamische Softwareplattform, die hardwareunabhängig ist und es ermöglicht, Anwendungen zu modularisieren und zu verwalten. Der Funktionsweise der Kommunikation ist in Abbildung 3.1 veranschaulicht.

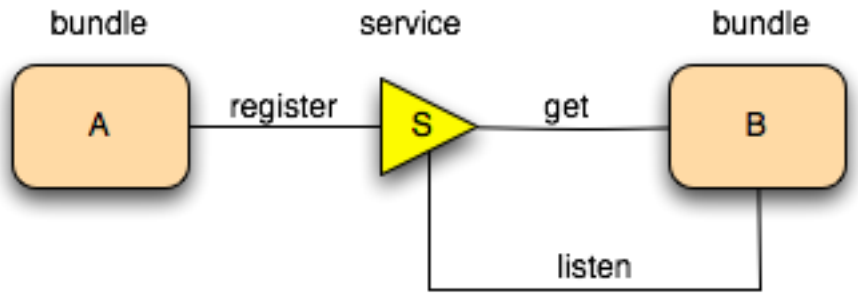

Abbildung 3.1: OSGi Komponentenmodell: Kommunikation zwischen Bundles [18]

Im Rahmen des OSGi Frameworks wird das gesamte Programm in verschiedene Softwarekomponenten (Bundles) aufgeteilt. Jede Komponente bietet und bezieht Services. Die

angebotenen Services werden von den entsprechenden Bundles zur Laufzeit im System registriert, wonach andere Bundles, die diese Services beziehen, darüber informiert und ihrerseits gestartet werden können. Ein Bundle wird erst gestartet, wenn alle von ihm benötigten Services im System registriert wurden.

Ein derartiger modularer Aufbau erlaubt es verschiedene Bundles nahezu unabhängig von einander zu entwickeln und später einzelne Bundles gegen aktuellere Versionen problemlos auszutauschen. Eclipse SmartHome besteht derzeit aus über 130 solcher untereinander kommunizierender Softwarekomponenten.

Features

Trotz der Positionierung als Grundlage für konkrete Smart Home Lösungen anderer Anbieter, verfügt das Framework über den kompletten Service Stack, der in einem SmartHome zum Einsatz kommt. Es existiert ein allgemeines Modellgerüst für die virtuelle Abbildung von realen Geräten. Für einige ausgewählte Geräte ist die Steuerung implementiert. Automatische Discovery von Geräten wird begrenzt unterstützt. Eine Rule Engine ist bereits integriert. Schließlich gibt es eine rudimentäre Benutzeroberfläche, die es erlaubt zur Laufzeit Geräte zum System hinzuzufügen und sie zu steuern. ESH hat eine ganze Reihe weiterer Features, wie z.B. Sprachunterstützung, die jedoch im Rahmen dieser Arbeit nicht von Interesse sind.

Zentrales Prinzip des gesamten Frameworks ist es möglichst generische Schnittstellen zu definieren und exemplarische Implementierungen anzubieten, die die beabsichtigte Funktionsweise veranschaulichen.

Im Folgenden werden die verschiedenen Aspekte von ESH, die für die Arbeit relevant sind, erläutert.

3.2 Modell

Die virtuelle Abbildung der realen Geräte im System veranschaulicht Abbildung 3.2.

Things

Things repräsentieren Entitäten, die zum System hinzugefügt werden können. In der Regel handelt es sich hierbei um verschiedene intelligente Geräte, wie beispielsweise eine *Hue* Lampe oder eine Wetterstation. Derartige Geräte bieten eine oder mehrere Funktionalitäten über Channels an.

Things, die als Brücke zu anderen intelligenten Geräten dienen, werden als Bridges bezeichnet (z.B. *Hue* Bridge);

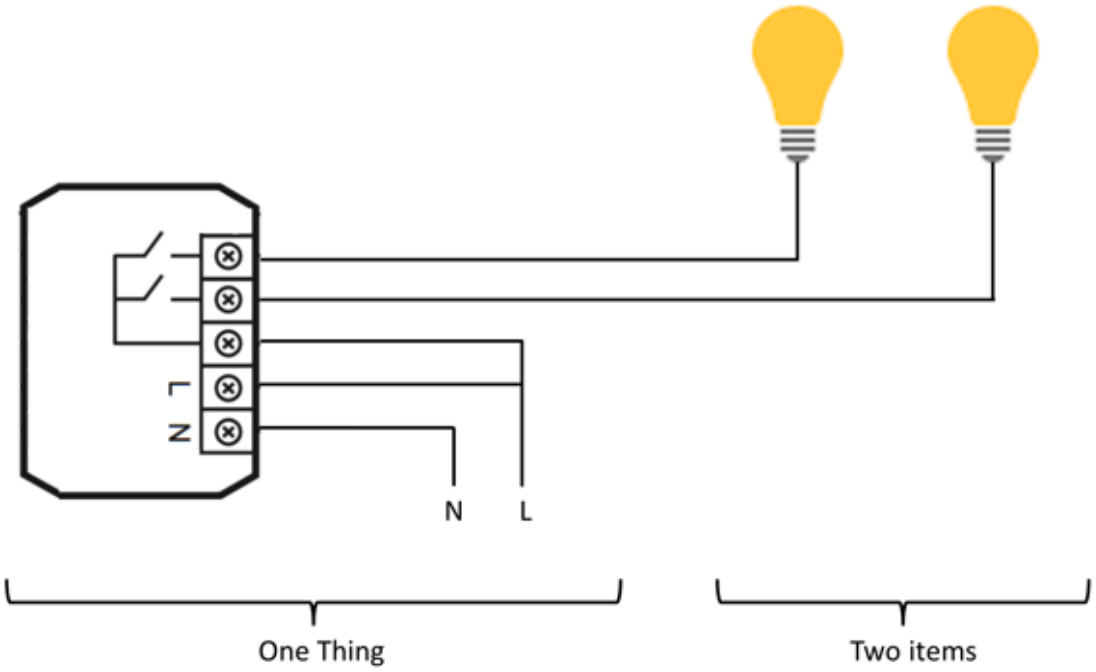

Abbildung 3.2: Architektur von *Things* und *Items* in ESH [5]

Channels

Ein Channel stellt eine bestimmte Funktionalität eines Things dar. Things können eine beliebige Anzahl von Channels anbieten. Beispielsweise kann eine Hue Lampe einen einzigen „Licht"-Channel anbieten, während eine Wetterstation die Channels „Temperatur", „Druck" und „Feuchtigkeit" unterstützt.

Items

Items repräsentieren konkrete, detaillierte Funktionalitäten eines Things. Beispielsweise kann ein Thing den Channel „Licht" unterstützen, wobei es 2 verschiedene Items besitzt, die beide jeweils eine physische Glühbirne verkörpern. Items haben einen Zustand, der im System gespeichert wird und können Commands empfangen.

Links

Links verbinden Items mit Things. Jeder Link assoziiert genau einen Channel des Things mit genau einem Item. Erst wenn ein Channel mit mindestens einem Item verbunden ist, gilt er als „aktiviert". Channels und Items können über eine beliebige Anzahl von Links untereinander verbunden werden.

3.2.1 Persistenz

In Eclipse SmartHome ist eine MapDB[8] bereits integriert. Dabei handelt es sich um eine leichtgewichtige, eingebettete Datenbank, in der zur Laufzeit alle Things, Items und Rules persistiert werden.

3.3 Automatisierung

In ESH herrschen eine Reihe von Design Prinzipien, die sich durch das gesamte Framework verfolgen lassen. Vor allem macht sich die Aufteilung in Modell und Controller bemerkbar: Für sämtliche Entitäten werden zunächst Typen deklariert, für welche über *ModuleTypeProvider* zugehörige Handler im System registriert werden. Dadurch wird die klassische Trennung von Datenmodell und Logik realisiert.

3.3.1 Rule Engine

Teil des Eclipse SmartHome Frameworks ist eine Regelmaschine. Sie hat eine generische Struktur und arbeitet mit ECA-Regeln, die durch eine Zusammenarbeit von Auslösern (Trigger), Bedingungen (Conditions) und Befehlen (Actions) die Automatisierung von Things ermöglicht. Trigger, Conditions und Actions werden in ESH auch als Module bezeichnet.

Trigger

Durch Trigger wird die Ausführung einer Regel angestoßen. Die Logik dahinter ist absolut frei implementierbar - der bekannte Aufbau, dass ein hineinkommendes Event den Trigger auslöst, ist nur eine Möglichkeit unter vielen. Dies wird ermöglicht, indem das Auslösen von Triggern von der Regelmaschine los gekoppelt und in einem dedizierten Trigger-Handler behandelt wird.

Der Handler erhält nach der Instanziierung einen Callback zur Rule Engine vom System. Über diesen Callback teilt er der Regelmaschine mit, wann immer die Ausführung einer Regel angestoßen werden soll. Wann dies geschieht, muss vom Handler selbst entschieden werden, die Rule Engine hat hierauf keinerlei Einfluss. Beispielsweise, falls der Trigger bei einem Event feuern sollte, müsste der Handler sich als *EventSubscriber* im System registrieren und auf das Event entsprechend reagieren.

Trigger-Handler werden für Instanzen von Triggern über *ModuleTypeProvider* bereitgestellt. Solche Provider sind dafür verantwortlich für bestimmte Modul-IDs zugehörige Handler zur Verfügung zu stellen. Jedes Mal, wenn eine Instanz eines Moduls im System erzeugt wird, wird unter den registrierten Providern nach einem passenden gesucht, der

für die ID des Moduls einen Handler generieren kann. Falls kein entsprechender Handler instanziiert werden kann, bleibt das Modul tot im System liegen und sämtliche damit gekoppelte Funktionalitäten fallen aus. Beispielsweise würde eine Rule, die einen Trigger enthält, für den kein korrespondierender Handler erzeugt werden kann, im System als „nicht initialisiert“ untätig bleiben.

Kontext

Während der Ausführung einer Rule ist ein Kontext gegeben, der stets von den Triggern über die Conditions an die Actions weitergereicht wird, was es ermöglicht komplexe Logik einzubauen. Alle Komponenten der Regel erhalten ihn bei der Ausführung übergeben und können ihn mit Informationen füllen. Beispielsweise kann das auslösende Event im Kontext gespeichert werden, sodass die darauffolgende Condition aufgrund dessen eine Entscheidung treffen kann, ob die Actions tatsächlich ausgeführt werden sollen. Eine Action könnte anhand des Kontextes entscheiden, auf welchen Wert ein Item geschaltet wird.

Conditions

Conditions sind analog zu Triggern aufgebaut. Sie agieren als Filter, indem sie festlegen, ob eine Regel, die durch einen Trigger angestoßen wurde, die zugehörigen Actions in Wirklichkeit ausführt. Bedingungen haben ebenfalls Handler, die den Kontext erhalten und entsprechend ihrer Logik eine Entscheidung treffen. Die Handler werden über zugehörige Provider instanziiert.

Actions

Actions spezifizieren, was genau getan werden soll, wenn eine Rule ausgeführt wird. Dabei kann es sowohl um die klassische Steuerung von intelligenten Geräten gehen, als auch um beliebige andere Automatisierung. Zu beachten ist, dass der Handler selbst dafür verantwortlich ist, das entsprechende Item ausfindig zu machen, an das er ein Command senden möchte. Hierfür registriert ESH im System Services wie *ThingRegistry* und *ItemRegistry*. Die Rule Engine bietet an dieser Stelle keinerlei Unterstützung - sie übernimmt ausschließlich das Weiterreichen des Kontextes zwischen Modulen.

Commands

Die Commands sind das verbindende Glied zwischen den Aktionen und den ThingHandlern. Sie werden von Aktionen erstellt und durch das System an den entsprechenden ThingHandler übermittelt. Hierfür muss die Aktion beim Erstellen spezifizieren, an welches Item das Command gesendet werden soll.

Es gibt eine ganze Reihe unterschiedlicher Commands in ESH. Unter anderen gibt es OFF/ON-Commands, String-Commands und Number-Commands. Falls gewünscht, lassen sich weitere Command-Typen im System registrieren.

3.3.2 Deklarative Typen

Eclipse SmartHome unterstützt die Möglichkeit viele Entitäten deklarativ im JSON-Format[35] in externen Dateien zu definieren. Es kann sich dabei sowohl um die Typen von Triggern, Conditions und Actions handeln, als auch beispielsweise um konkrete Regeln, die beim Systemstart ausgelesen und importiert werden. Sämtliche JSON-Dateien, die auf diese Weise Entitäten deklarieren, müssen innerhalb eines Bundles im Ordner *ESH-INF* gelagert werden, wobei weitere Unterordner (z.B. *automation*, *moduletypes*) feinere Unterteilung ermöglichen.

Dabei werden die Namen und Typen der Modell-Attribute festgelegt. Jede Entität muss eine eindeutige ID besitzen, denn für diese ID werden sogenannte *ModuleTypeProvider* im System registriert, die dafür verantwortlich sind für eine konkrete Instanz einer Entität einen entsprechenden Handler bereit zu stellen.

Alternativ ist es auch möglich Typen von Modulen klassisch per Java-Code zu definieren.

3.3.3 Events

ESH arbeitet über den klassischen OSGi Event Bus. Dennoch ist es auch möglich eigene Event Typen zu definieren. Hierfür muss zunächst der Typ des Events (POJO mit Attributen) festgelegt werden. Daraufhin muss für dieses Event ein EventProvider im System registriert werden, der dafür verantwortlich ist, ein generisches OSGi Event in ein entsprechendes Event vom spezifizierten Typ umzuwandeln. Die Zuordnung findet klassischerweise über eindeutige IDs statt. Ein OSGi Event besteht aus *type*, *source*, *topic* und *payload*.

3.4 Bindings

Ein Binding ist eine Erweiterung (in der Regel ein eigenständiges Bundle) des Eclipse SmartHome Frameworks, dass für die Integration einer externen Funktionalität in Form von Things verantwortlich ist. In der Regel handelt es sich dabei um verschiedene intelligente Geräte, doch es können auch beispielsweise Webservices angebunden werden.

Ein Binding besteht aus einer Reihe von Komponenten. Die grundlegenden Elemente sind im Folgenden aufgeführt:

1. Eine Datei *binding.xml* im Unterordner *ESH-INF/binding*, die die ID des Bindings eindeutig festlegt.
2. Eine Datei *thing-types.xml* im Unterordner *ESH-INF/thing*, in der die durch dieses Binding unterstützten Things und Channels spezifiziert werden.
3. Eine *HandlerFactory*, die für die in der *thing-types.xml*-Datei spezifizierten Things zugehörige Handler instanziiert.
4. Eine Reihe von Thing-Handlern, über die die Things tatsächlich gesteuert werden können.

Thing Handler

Thing Handler sind dafür verantwortlich, die Logik für die Steuerung von Geräten zu implementieren. Sie sind zuständig für die automatische Aktualisierung von Zuständen im System (z.B. regelmäßige Aktualisierung der Temperatur, die von einem Thermostat geliefert werden). Außerdem erhalten sie Commands vom System (z.B. als Teil einer Action einer Rule).

Kapitel 4

Entwurf

Bevor mit der Implementierung angefangen werden kann, müssen erst Entscheidungen getroffen werden, wie die einzelnen Anforderungen umgesetzt werden sollen. Es folgt eine schrittweise Analyse der einzelnen Anforderungen.

4.1 Überblick

Im Rahmen der Arbeit müssen die Anforderungen umgesetzt werden, die in Sektion 2.2.2 zusammengefasst wurden. Durch die Nutzung von Eclipse SmartHome ist die Anforderung, dass das System in der Lage ist Geräte innerhalb des Hauses ohne Internetverbindung zu steuern, bereits gegeben. Es ist jedoch darauf zu achten, dass durch die Integration der neuen Funktionalitäten die existierenden nicht beeinträchtigt werden. Auf welche Weise die übrigen Anforderungen umgesetzt werden sollen, wird in den nachfolgenden Sektionen diskutiert.

4.2 Webservices

Man kann im Allgemeinen zwischen *öffentlichen* und *privaten* Webservices differenzieren. Der wesentliche Unterschied ist, dass öffentliche Dienste keine Nutzerverwaltung einschließen - sämtliche Schnittstellen und bereitgestellten Informationen sind frei verfügbar. Ein online Wetterdienst ist ein Beispiel für einen öffentlichen Webservice.

Unter privaten Webservices handelt es sich in diesem Kontext um Dienste, deren Funktionalität von eingeloggten User abhängt. Ein Beispiel hierfür ist *Dropbox*. Der Zugriff auf die in der Dropbox gelagerten Dateien ist erst möglich, nachdem sich der User im System authentisiert hat.

Es gibt eine Vielzahl von verschiedenen Sicherheitsprotokollen, die an dieser Stelle zum Einsatz kommen. Vor allem hat sich derzeit der OAuth Standard etabliert, welcher auch

in den Webservices *Twitter* und *Dropbox* genutzt wird.

OAuth Standard

OAuth[24] ist ein offener Standard für Autorisierung, der es ermöglicht Nutzern Applikationen von Drittanbietern den Zugriff auf Webdienste in ihrem Namen zu gestatten, ohne dabei das eigene Passwort freizugeben. Dies geschieht in der Regel in mehreren Schritten.

1. Die Applikation registriert sich bei dem Webservice und erhält einen Consumer-Key und Consumer-Secret.
2. Ein Nutzer möchte der Applikation gestatten auf seinen Account zuzugreifen.
3. Die Applikation teilt dem Nutzer eine URL mit, über die er dies erlauben kann. Diese URL wird vom Webservice basierend auf dem Consumer-Key bereitgestellt.
4. Der Nutzer autorisiert sich unter der URL beim Webservice klassischerweise mit seinem Namen und Passwort. Daraufhin erhält er die Möglichkeit, der Applikation die geforderten Zugriffsrechte einzugestehen.
5. Nachdem der Nutzer die Erlaubnis erteilt hat, erhält er eine PIN, die er manuell in der Applikation eingeben muss.
6. Nachdem er dies getan hat erhält die Applikation ein generiertes OAuth-Token, mit dem sie Zugriff auf den Account des Users hat. Der Autorisierungsprozess ist damit abgeschlossen.

Im Rahmen dieser Arbeit wird es nötig sein, sowohl mit öffentlichen, als auch mit privaten Webservices Kontakt aufzunehmen. Es wird also notwendig sein, sich gegenüber den Diensten zu authentisieren um die erforderlichen Rechte zu erhalten, die es ermöglichen den Workflow des Nutzers zu automatisieren. Die Autorisierung wird über die Benutzeroberfläche stattfinden.

4.3 Integration in Eclipse SmartHome

Im Rahmen der Arbeit sollen die bereits existierenden Funktionalitäten von ESH sofern möglich wiederverwendet werden. Hierzu sollen die einzelnen Webservices durch Bindings in das System integriert werden. Dabei soll jede Instanz eines Webdienstes als ein Thing repräsentiert werden. Das bedeutet, dass beispielsweise im Falle eines Wetterdienstes für jeden Ort ein eigene Instanz des Thing-Typen erstellt wird, die diesen Ort darstellt. Alternativ könnte es sich bei den Instanzen um verschiedene Twitter-Accounts handeln.

Die benötigten Zugriffsdaten (z.B. OAuth-Token) werden in den Konfigurationen der Things abgelegt. Ein solcher Aufbau garantiert, dass jedes Thing über sämtliche Informationen verfügt, die benötigt werden, um es zu steuern. Die konkreten Funktionalitäten werden über ein oder mehrere Channels dargestellt.

4.3.1 Verbindung zum Webdienst

Nachdem ein Thing mit den benötigten Zugriffsdaten angelegt wurde, muss es vom System verwaltet werden können. Hierzu gehört vor allem die virtuelle Abbildung des realen Zustandes zu aktualisieren, sowie die tatsächliche Steuerung zu ermöglichen.

Polling und Webhooks

Die Aktualisierung des Zustandes kann auf zwei Arten geschehen: Polling und WebHooks. Beim Polling ist die Applikation selbst dafür verantwortlich die aktuellen Werte abzurufen. Beispielsweise könnte die Applikation in einem festgelegten Intervall prüfen, welche Dateien in der Dropbox vorhanden sind. Änderungen würden in entsprechenden Events publiziert werden.

Falls der Webservice es unterstützt, können statt Polling auch WebHooks angewendet werden. Hierbei handelt es sich um HTTP Callbacks - die Applikation registriert also eine Adresse beim Dienst und relevante Informationen (Zustandsänderungen, etc.) werden von Dienst an diese Adresse kommuniziert. Dieser Aufbau hat den Vorteil, dass die Netzwerklast deutlich reduziert und gleichzeitig die Reaktionszeit merklich erhöht wird. Leider unterstützen derzeit nur wenige Webservices WebHooks.

Verbindung

Es soll für jedes (Webservice-)Thing, welches im System registriert wird, eine Verbindung aufgebaut werden, die auf eine der oben beschriebenen Arten die Zustände aktuell hält. Diese Verbindung soll automatisch geöffnet werden, wenn ein Gerät dem System hinzugefügt wird und wieder geschlossen werden, wenn es entfernt wird. Sie ist dafür verantwortlich bei Zustandsänderungen entsprechende Events im System zu publizieren. Die Details sollen hierbei im Payload im JSON Format bereitgestellt werden.

JSON wurde an dieser Stelle ausgewählt, da es die Möglichkeit bietet, komplexe Datenstrukturen über das Payload Attribut zu vermitteln. Dies erlaubt es zu vermeiden, dass jedes Binding eigene Events definieren und registrieren muss. Außerdem müssen auch die verschiedenen Handler nicht für jeden neuen Event-Typ angepasst werden - es reicht, wenn sie im Payload nach den für sie relevanten Attributen suchen. Eine Tabelle mit den pu-

blizierten Events, sowie den Eingaben und Ausgaben von Regel-Modulen ist in Abbildung 5.1 aufgeführt.

Eigene Events und Module von Regeln

ESH bietet die Möglichkeit eigene Event-Typen zu definieren. Dies ist stets eine Option, die jedem Binding offen ist. Im Rahmen der Arbeit reicht es jedoch, einen allgemeinen neuen Event-Typ zu definieren und daraufhin die konkreten *topics* und *payloads* zu variieren. Beispielsweise würde bei Hochladen einer neuen Datei in die Dropbox ein Event mit dem Topic „flash/dropbox/added“ und einem Payload, dass die Metadaten (Name, Pfad, etc.) der Datei im JSON Format enthält, im System publiziert.

Analog soll mit Triggern, Conditions und Actions von Regeln umgegangen werden. Es sollen zunächst *GenericEventTrigger* und *EventCondition* zum Einsatz kommen.

Der *GenericEventTrigger* ist ein von ESH bereits implementierter Auslöser, der die Ausführung einer Regel anstoßt, wenn ein Event eintrifft. Das erwartete Event kann vom Nutzer konfiguriert werden. Der Trigger stellt das auslösende Event außerdem im Kontext (siehe Sektion 3.3.1) bereit.

Die *EventCondition* ermöglicht es das erhaltene Event nach seinen Attributen zu filtern. In der Konfiguration der Bedingung werden reguläre Ausdrücke für die vier Attribute des Events hinterlegt. Wenn die Bedingung evaluiert wird, wird geprüft, ob die tatsächlichen Werte des Events mit dem Regex übereinstimmen.

Zusammen decken diese beiden Module einen großen Teil der möglichen Szenarien ab. Bindings, die Funktionalitäten anbieten oder einer Logik bedürfen, die von diesen Werkzeugen nicht abgedeckt werden kann, haben stets die Option eigene Module im System zu registrieren.

4.3.2 Persistenz

Wie in Sektion 3.2.1 erläutert, ist eine MapDB in ESH bereits integriert. Da im Rahmen dieser Arbeit nur geringe Mengen an Daten persistiert werden müssen, ist es nicht notwendig, auf eine mächtigere Datenbank umzusteigen.

Dadurch, dass die Webservices selbst als Things modelliert werden und alle notwendigen Konfigurationsdaten mit sich tragen, kann dieser Aspekt von ESH in seiner aktuellen Form wiederverwendet werden. Bei einem Neustart des Systems werden sämtliche relevante Things aus der Datenbank ausgelesen und basierend auf den in ihrer Konfiguration gespeicherten Zugriffsdaten (z.B. OAuth Token) entsprechende Verbindungen aufgebaut, die für die Aktualität der Zustände sorgen.

4.3.3 Benutzeroberfläche

Die Benutzeroberfläche soll den Nutzer in die Lage versetzen Regeln zur Laufzeit zu bearbeiten. Das bedeutet, es soll ihm möglich sein neue Regeln anzulegen, sowie existierende Regeln zu betrachten, zu editieren und zu löschen. Da Eclipse SmartHome die Möglichkeit bietet Regeln im JSON-Format zu deklarieren (siehe Sektion 3.3.2), bietet sich an dieser Stelle an, es in der Benutzeroberfläche analog umzusetzen.

Des weiteren soll dem User die Möglichkeit gegeben sein, die Zugriffsrechte auf die unterstützten Webservices (z.B. auf seinen Twitter Account) der Anwendung zu übermitteln, bzw. auf diese Weise neue Instanzen des entsprechenden Thing-Typen zum System hinzuzufügen.

4.4 OSGi auf dem Raspberry Pi

Eclipse SmartHome besteht aus einer Reihe von OSGi Bundles und die im Rahmen der Arbeit erstellten zusätzlichen Funktionalitäten nehmen ebenfalls diese Form an. Es lässt sich annehmen, dass sofern ein OSGi Container auf dem Pi laufen wird, auch das gesamte Framework gestartet werden kann.

Konkret betrachtet wird auf dem Raspberry Pi ein Betriebssystem benötigt, das Java unterstützt. Daraufhin muss ein OSGi Container wie beispielsweise Concierge[3] oder Apache Felix GoGo[27] gestartet werden. Danach muss er entsprechend konfiguriert werden, sodass die Jar-Dateien der relevanten ESH Bundles auf die benötigten allgemeinen System-Jars von Java Zugriff haben. Schließlich müssen alle Abhängigkeiten der im Container installierten Bundles erfüllt sein, damit sie gestartet werden können. Beispielsweise bedarf eine Web-UI eines entsprechenden HTTP Servers.

4.5 Fazit

Jeder Webdienst wird in einem eigenen Binding in das System integriert. Dabei werden Thing-Typen registriert, deren Instanzen konkrete Ausprägungen des Webservices repräsentieren. Für jede Thing-Instanz wird eine Verbindung aufgebaut, die für die Aktualität der Zustandsdaten und das Publizieren entsprechender Events verantwortlich ist. Jedes Binding entscheidet selbst, ob ihm die existierenden Module ausreichen oder es neue Entitäten im System registriert.

Kapitel 5

Implementierung

Die Implementierung wurde entlang des Entwurfs durchgeführt. Es werden UML Klassen- und Sequenz-Diagramme verwendet, um die Zusammenhänge zu visualisieren.

5.1 Überblick

Im Rahmen der Arbeit sind eine Reihe von Bundles entstanden, die die verschiedenen Funktionalitäten umsetzen. Es lassen sich drei Arten von Bundles unterscheiden:

1. Die Bundles *tka.binding.core* und *tka.automation.extension* bieten eine Reihe generischer Schnittstellen, die von den Bindings genutzt werden.

2. *tka.binding.twitter*, *tka.binding.dropbox*, *tka.binding.weather* und *tka.binding.gmail* sind Bindings, die jeweils einen konkreten Webservice in das System integrieren.

3. Das *tka.flashui* Bundle stellt eine Benutzeroberfläche bereit, über die neue Webservice-Things zum System hinzugefügt und durch Regeln automatisiert werden können.

5.2 Schnittstellen-Bundles

5.2.1 tka.binding.core

Das Bundle *tka.binding.core* enthält die zentrale Schnittstelle für den Prozess der Authentifizierung. Sie ist generisch aufgebaut und entspricht den Anforderungen des OAuth Standards (siehe Sektion 4.2). In Abbildung 5.1 ist die Struktur visualisiert.

ConnectionService

Die ConnectionService Schnittstelle bietet drei wesentliche Methoden, die bei der Authentifizierung relevant sind. Per *requestAuthorization* teilt der Nutzer mit, dass er wünscht, die Anwendung mit einem seiner Webservice-Accounts (beispielsweise Twitter) zu verknüpfen.

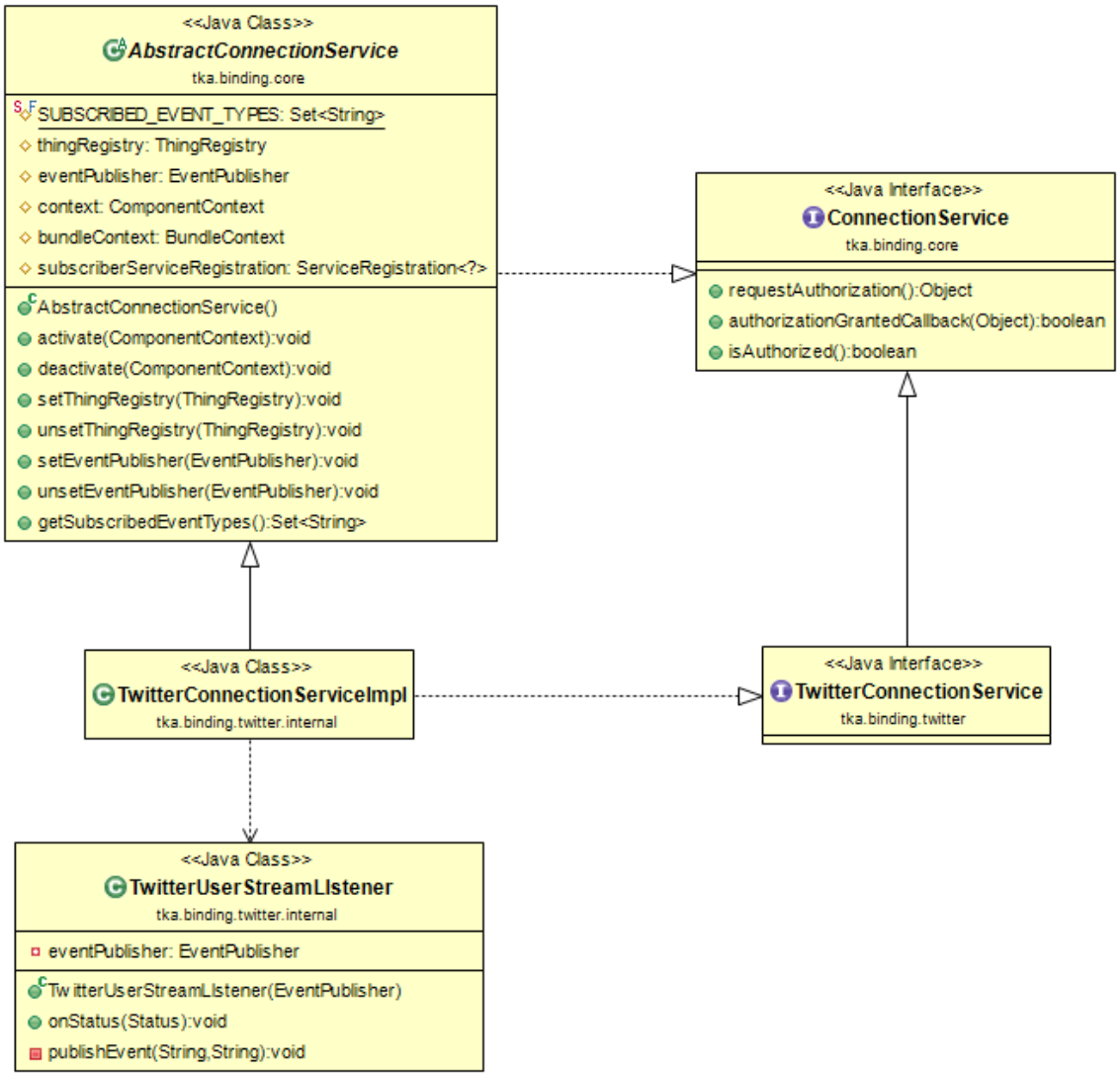

Abbildung 5.1: Die Struktur einer Verbindung mit Besipielimplementierung für Twitter

Daraufhin bekommt er in der Regel eine URL geliefert, über die er dies umsetzen kann. Die PIN, die er auf diese Weise erhält, teilt er über den *authorizationGrantedCallback* der Anwendung mit. Schließlich lässt sich über *isAuthorized* stets abfragen, ob die Anwendung bereits über die benötigten Zugriffsrechte verfügt. Im Rahmen dieser Arbeit wurde entschieden, sich auf die tatsächliche Unterstützung nur eines Accounts einer Art zu beschränken, obwohl die generische Struktur geeignet ist, um beliebige Mengen von Accounts zu verwalten.

AbstractConnectionService

Der AbstractConnectionService bietet eine Reihe von unterstützenden Funktionalitäten, die in der Regel von den Bindings benötigt werden. Hierzu gehört unter anderem die Beziehung des *ThingRegistry*- und *EventPublisher*-Services aus dem System, sowie die eige-

ne Registrierung als *EventSubscriber*. Die ThingRegistry wird benötigt um entsprechende Things im System zu registrieren, nachdem die Autorisierung erfolgreich abgeschlossen ist. Über den EventPublisher publizieren die konkreten Listener (z.B. TwitterUserStream-Listener) die Zustandsänderungsevents (mehr dazu in Sektion 5.3). Als EventSubscriber registriert sich ein ConnectionService um auf Änderungen von Things im System entsprechend reagieren zu können. Beispielsweise, falls ein Thing gelöscht wird, muss auch der zugehörige Listener entfernt werden.

Jedes Binding enthält eigene Implementierungen der oben genannten Schnittstellen, die die Besonderheiten des konkreten Webdienstes berücksichtigen. Diese Implementierungen werden außerdem als Services in OSGi registriert, damit über die Benutzeroberfläche die Authentifizierung umgesetzt werden kann.

5.2.2 tka.automation.extension

Das *tka.automation.extension* Bundle registriert im System einige gemeinsamen Services, die von anderen Bundles genutzt werden. Unter anderem wird hier ein neuer Event-Typ registriert: Das *FlashEvent*. Die implementierten Bindings publizieren alle Events dieses Typen, mit unterschiedlichen Topics und Payloads. Dies erlaubt es zu vermeiden, dass jedes Binding seine eigenen Event-Typen deklarieren muss.

5.3 Bindings

Die Bindings sind wie in Sektion 3.4 erläutert aufgebaut. Es wird stets ein neuer Thing-Typ über die *thing-types.xml* im System registriert. Die Anzahl der unterstützten Channels hängt dabei vom jeweiligen Webservice ab.

Jedes Binding integriert einen eigenen Webservice und fügt, sofern notwendig, spezielle Trigger und Actions dem System hinzu. Diese Module erlauben es die Interaktion mit diesen Services über die ESH Rule Engine sinnvoll zu automatisieren.

Alle Bindings haben gemeinsam, dass sie bei den respektiven Webdiensten zunächst registriert werden müssen. Dabei erhalten sie ein Token, mit dem sie sich gegenüber dem Dienst bei Aufrufen identifizieren. Wie komplex dieser Ablauf ist, hängt vom jeweiligen Service ab.

5.3.1 tka.binding.twitter

Das Twitter Binding fügt dem System Things des Typs „twitter“ hinzu. Es wird derzeit der Channel „status“ unterstützt. Dies bedeutet, dass es möglich ist über die Rule Engine den aktuellen Status des Nutzers über Regeln zu manipulieren. Die entsprechende Logik

ist in dem *TwitterHandler* und der *TwitterAction* hinterlegt.

Der *TwitterHandler* ist ein ThingHandler, der konkrete (String-)Commands erhält und dafür verantwortlich ist den Twitter-Status des Nutzers entsprechend zu aktualisieren. Hierzu liest er sich die Zugriffsdaten aus der Thing-Konfiguration aus und teilt Twitter mit, was geschehen soll.

Die *TwitterAction* ist dafür verantwortlich die konkreten Commands zu erzeugen und an den entsprechenden TwitterHandler zu vermitteln. Hierfür erwartet sie eine Reihe von Konfigurationsparametern, die benötigt werden, um eindeutige Befehle zu erstellen. In Tabelle 5.1 ist aufgeführt, welche Eingabeparameter das Modul erwartet.

Umgekehrt ist das Binding in der Lage auf viele Zustandsänderungen des Accounts zu reagieren. Die konkrete Logik hierfür ist im *TwitterUserStreamListener* implementiert. So werden unterschiedliche Events publiziert, wenn

1. sich ein Status, dem der Nutzer auf Twitter folgt, ändert. In diesem Fall wird der neue Status im Payload des Events gespeichert.
2. ein Status sich ändert und Medien (z.B. Bilder) enthält. In diesem Fall wird für jedes Medium ein eigenes Event publiziert, dass die Medien-URL enthält.
3. der Nutzer eine Nachricht erhält. Der Text der Nachricht wird im Payload hinterlegt.
4. viele weiter Ereignisse eintreffen.

Falls ein bestimmtes Szenario nicht abgedeckt ist, lässt sich der *TwitterUserStreamListener* entsprechend erweitern.

Bei der Interaktion mit Twitter kommt die Twitter4J-Bibliothek[16] zum Einsatz. Dabei handelt es sich um eine leichtgewichtige Java-Bibliothek, ohne zusätzliche Abhängigkeiten, die die Twitter API 1.1 komplett unterstützt. Sie assistiert zusätzlich bei dem OAuth Prozess.

Überblick über die relevanten Trigger, Conditions und Actions

In der Tabelle sind die Eingaben und Ausgaben von wichtigen Auslösern, Bedingungen und Aktionen aufgeführt. Zu beachten ist, dass die Eingaben sowohl über den Kontext übergeben, als auch direkt in der Konfiguration der konkreten Instanz fest definiert werden können. Dies erlaubt es sowohl statische als auch generische Regeln zu definieren. Falls

Modul Name	Eingaben	Ausgaben
GenericEventTrigger	-	<triggerId>.event
EventCondition	event	-
TwitterAction	event [itemName, message]	-
DropboxAction	event [itemName, directory, mediaUrl]	-
EmailAction	event [to, subject, message]	-

Tabelle 5.1: Eingaben und Ausgaben der relevanten Trigger, Conditions und Actions

Parameter fehlen sollten, wird die konkret betroffene Aktion nicht ausgeführt, der gesamte Ausführungsprozess wird nicht unterbrochen.

5.3.2 tka.binding.dropbox

Das Bundle *tka.binding.dropbox* ist analog zu *tka.binding.twitter* aufgebaut. Es wird der neue Thing-Typ „dropbox“ mit dem Channel „folder“ im System registriert. Über diesen Channel gibt es die Möglichkeit neue Dateien in die Dropbox hochzuladen.

Der *DropboxChangesTracker* übernimmt die Rolle des *TwitterUserStreamListeners*. Er überprüft alle 5 Sekunden den Inhalt der Dropbox und publiziert Events für sämtliche Änderungen.

Über die *DropboxAction* lassen sich neue Dateien automatisch in die Dropbox hochladen. Beispielsweise kann eine Regel erstellt werden, die mit Hilfe eines GenericEventTriggers auf vom Twitter-Binding publizierte (Medien-)Events reagiert und daraufhin über die Medien-URL die Datei an einem spezifizierten Ort speichert.

Bei der Interaktion mit dem Webservice kommt die vom Hersteller angebotene Dropbox SDK zum Einsatz. Dabei handelt es sich um eine Java-Bibliothek, die den Entwickler bei der OAuth-Authentifizierung und der gesamten Interaktion mit Dropbox (Hochladen und Löschen von Dateien, etc.) unterstützt.

Diese externen Bibliotheken (Dropbox SDK, Twitter4J) wurden mithilfe des bnd-Tools[1] zu OSGi-Bundles umgewandelt. Dabei stellte sich heraus, dass die Dropbox SDK Abhängigkeiten auf eine Reihe von Android Bibliotheken besitzt. Nach weiterer Recherche ließen sich diese Abhängigkeiten mithilfe des bnd-Tools auf optional umstellen, wonach die SDK problemlos in der OSGi Umgebung eingesetzt werden konnte.

5.3.3 tka.binding.weather

Das *tka.binding.weather* Binding weist sowohl Gemeinsamkeiten, als auch Unterschiede zu den bisher vorgestellten Bundles auf. Es handelt sich bei dem Wetterdienst um einen Webservice, der einer regulären intelligenten Wetterstation stark ähnelt. Dadurch ließen sich mehr Funktionalitäten von ESH wiederverwenden, als bei den anderen Diensten.

Gleichzeitig unterscheidet es sich von den bisher vorgestellten dadurch, dass es sich bei dem Webservice um einen öffentlichen Dienst handelt. Das bedeutet, dass ein komplexer Autorisierungsprozess (beispielsweise über OAuth) an dieser Stelle nicht benötigt wird. Dies führt dazu, dass keine eigene Implementierung der in Sektion 5.2.1 vorgestellten Schnittstellen notwendig ist.

Das Binding registriert den neuen Thing-Typen „weather“ mit den Channels *temperature*, *humidity* und *rain*.

Da die Zustandsaktualisierung in diesem Fall vergleichsweise simpel verläuft, wurde sie nicht in eine dedizierte Klasse ausgelagert. Stattdessen wurde der ThingHandler um die entsprechende Funktionalität erweitert. Da sich das Wetter von Region zu Region mit sehr unterschiedlicher Geschwindigkeit ändern kann, lässt sich die Aktualisierungsperiode über die Benutzeroberfläche beliebig konfigurieren.

5.3.4 tka.binding.gmail

Das Bundle *tka.binding.gmail* erlaubt es dem Nutzer Emails zu versenden. Es ist analog zu den bisherigen Bindings aufgebaut. Der Thing-Typ „gmail“ wird im System registriert und über einen entsprechenden ThingHandler gesteuert.

Das Binding registriert den Action Typen *EmailAction*, über den der Nutzer angeben kann, an wen eine Email mit den gewählten Informationen gesendet werden soll. Die Email wird von einer anwendungsinternen Google Mail[7] Adresse *noreply.flash.ma@gmail.com* über einen öffentlichen SMTP Server versandt.

5.4 Benutzeroberfläche

Die Benutzeroberfläche ist komplett im Bundle *tka.flashui* realisiert. Das Bundle ist von den Bindings los gekoppelt - es nutzt einige der von ihnen bereitgestellten Services um eine rudimentäre, aber dennoch voll funktionsfähige GUI bereitzustellen. Es ist jederzeit möglich diese Benutzeroberfläche auszutauschen oder weitere Schnittstellen parallel im System zu registrieren.

5.4.1 tka.flashui

Aus technischer Sicht wurde die GUI größtenteils über REST-Schnittstellen, Javascript und AJAX realisiert.

Backend

Seitens des Backends wird vom OSGi Container ein HTTP-Service bezogen, sowie die von den Bundles bereitgestellten konkreten ConnectionServices. Daraufhin wird in Form von Servlets eine Reihe von Schnittstellen definiert, über die verschiedene Informationen aus dem Frontend per AJAX abgefragt werden können. Auf diese Weise werden Informationen über die aktuell im System verwalteten Rules und Things bereitgestellt. Für jedes Binding gibt es ein eigenes Servlet, das die Interaktion damit (hauptsächlich für die Authentifizierung) ermöglicht.

Frontend

Bei dem Frontend handelt es sich um eine Single-Page-Webanwendung. Es gibt eine HTML-Datei, die ein leeres Grundgerüst für die Seite bietet und eine Reihe von Javascript Dateien, die beim Öffnen der Seite geladen werden. Die verschiedenen logischen Aspekte der Anwendung werden in separaten JS-Dateien realisiert. Für jedes Binding gibt es ebenfalls eine einzelne JS-Datei, in der die Logik der Zusammenarbeit mit dem konkreten Binding realisiert ist.

Das Erweitern der GUI um neue Bindings lässt sich damit auf eine einfache Art und Weise realisieren, ohne existierenden Code ändern zu müssen. Seitens des Backends muss für ein neues Binding eine neue REST-Schnittstelle in Form eines Servlets definiert werden. Das Frontend bedarf einer lediglich einer weiteren JS-Datei.

Funktionalität und visuelle Darstellung

Die Benutzeroberfläche bietet die Möglichkeit die implementierten Funktionalitäten zur Laufzeit zu konfigurieren. Sie ist in Abbildung 5.2 dargestellt.

Es ist leicht zu sehen, dass die Oberfläche grob in drei Abschnitte unterteilt werden kann. Im Abschnitt *Rules* gibt eine Tabelle mit den aktuell existierenden Regeln zu sehen, sowie der Möglichkeit, diese Regeln zu manipulieren. Neue Regeln können im JSON-Format konfiguriert und über *Import Rules* an das Backend übermittelt werden. Sie werden daraufhin mithilfe von GSON[34] zu Rules umgewandelt und dem System hinzugefügt. Auf diese Weise importierte Regeln überschreiben existierende Regeln mit der gleichen Id.

Im Abschnitt *Things* sind die aktuell im System vorhandenen Things mit den von ihnen unterstützten Channels zu sehen.

Im letzten Teil der Benutzeroberfläche ist eine Liste mit den unterstützten Webservices bereitgestellt. Für jeden Webdienst gibt es hier die Möglichkeit die Zugriffsrechte an die Anwendung zu vermitteln.

Beim Stylen der Oberfläche kam Bootstrap[2] zum Einsatz.

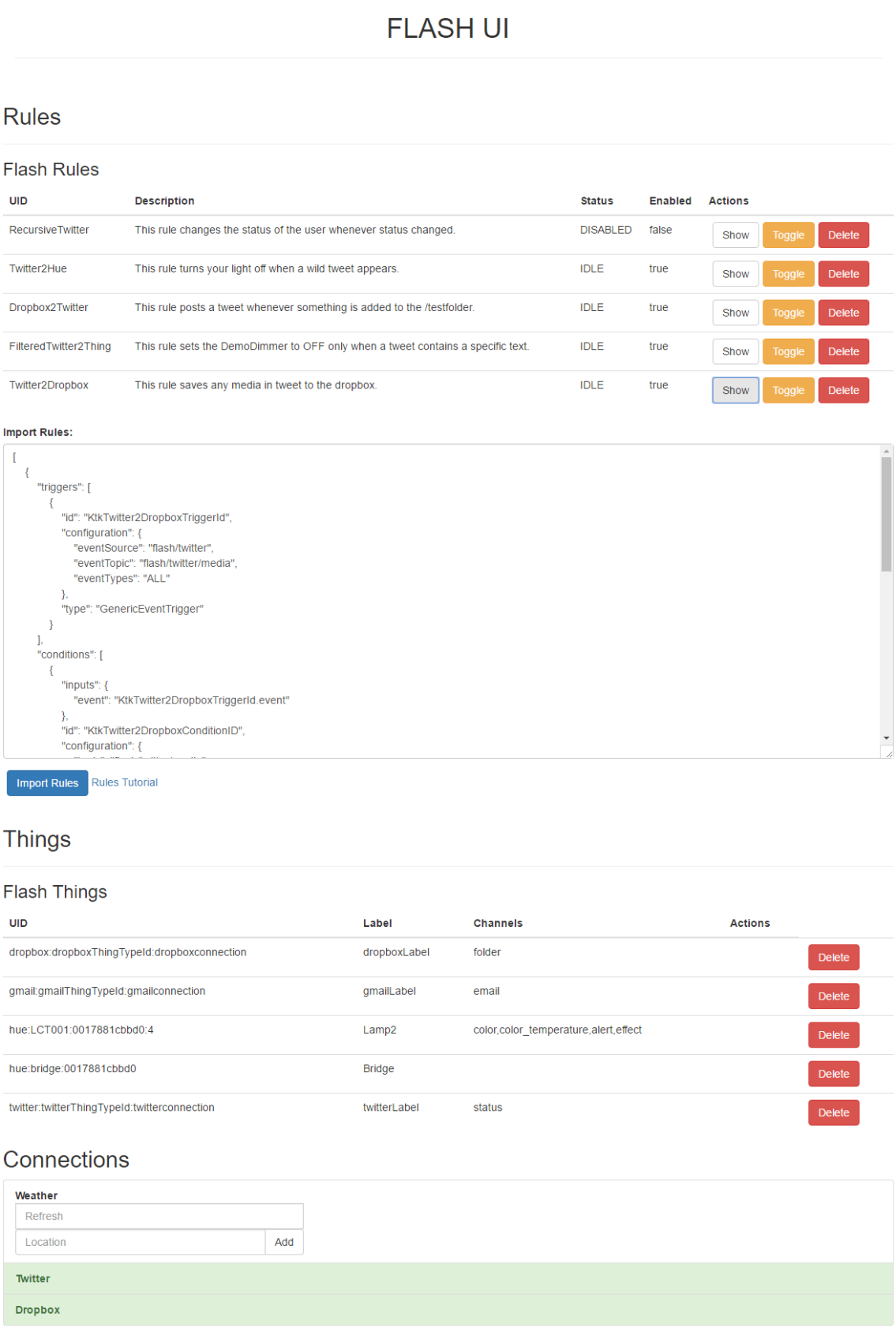

Abbildung 5.2: Die Benutzeroberlärche

5.4.2 Beispiel

Um die Zusammenarbeit der verschiedenen Komponenten zu visualisieren, wurde folgendes Diagramm erstellt:

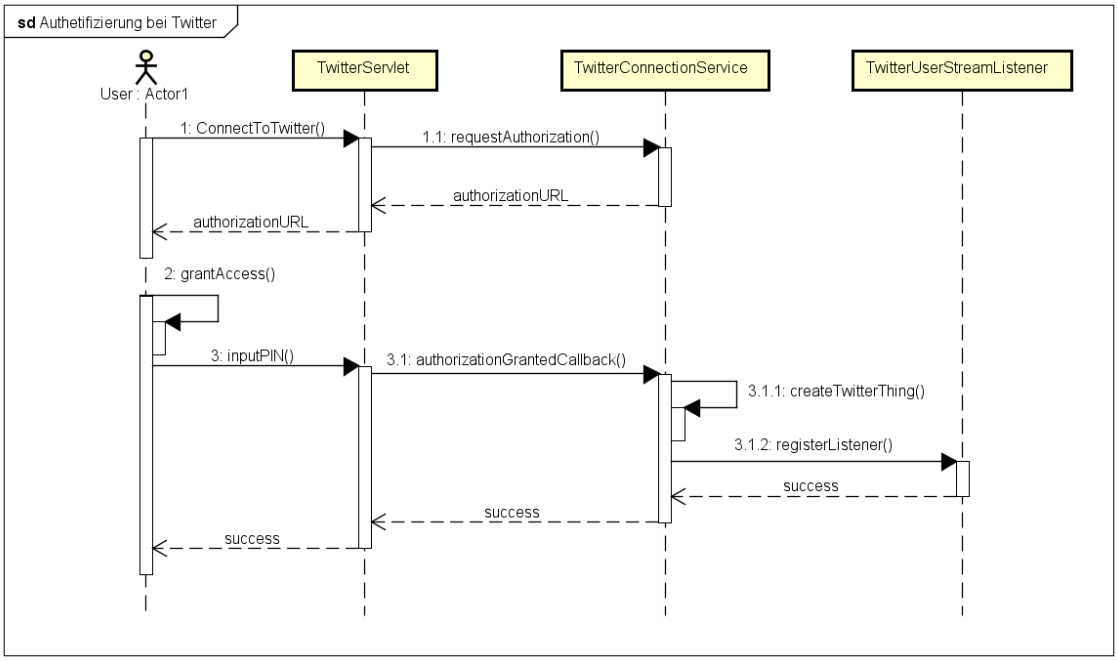

Abbildung 5.3: Authentifizierung bei Twitter als Sequenzdiagramm

In Abbildung 5.3 ist in einem Sequenzdiagramm schematisch dargestellt, wie die Authentifizierung bei Twitter abläuft. Der Nutzer äußert seinen Wunsch seinen Twitter Account dem Prototypen anzuvertrauen. Daraufhin vermittelt der TwitterServlet ihm eine URL, über die er dies wie in Sektion 4.2 erläutert umsetzen kann. Nachdem die korrekte PIN dem System eingegeben wurde, wird dem System ein neues Thing vom Typ „twitter“ hinzugefügt und ein entsprechender Listener im System registriert.

5.5 Deployment auf dem Raspberry Pi

Das Deployment fand auf einem Raspberry Pi 3 Model B statt. Auf dem Gerät lief das Betriebssystem *Raspbian* mit installiertem Java 8.

Als Grundlage für die Distribution wurde das *Eclipse SmartHome Packaging Sample*[6] verwendet. Es erlaubt es mithilfe von Maven[32] eine leichtgewichtigen OSGi Container mitsamt einigen der zusätzlich benötigten Services zu bauen. Diese Distribution wurde entsprechend angepasst:

1. Weitere notwendige Service-Bundles wurden hinzugefügt. Vor allem bei dem Einbinden von den externen Bibliotheken (Dropbox SDK und Twitter4J) mussten an dieser Stelle zusätzliche Konfigurationen vorgenommen werden.

2. Die eigenen Bundles wurden mithilfe von Maven zu OSGi-Jar-Dateien umgewandelt und zum Deployment hinzugefügt.

Schließlich wurde die Distribution auf dem Raspberry Pi gestartet und und einem umfangreichen Test unterzogen. Die Ergebnisse dieses Tests werden in Kapitel 6 vorgestellt.

Anmerkung: Die Distribution kann über das *start.sh* Script gestartet werden. Die Benutzeroberfläche ist unter *http://localhost:8080/flash/index.html* erreichbar.

5.6 Fazit

Die Implementierung wurde entlang des in Kapitel 4 definierten Entwurfs umgesetzt. Sie ist generisch aufgebaut und lässt sich leicht um neue Funktionalitäten erweitern.

Kapitel 6

Evaluation

In diesem Kapitel wird die entstandene Implementierung gegen die Anforderungen verglichen und das Ergebnis evaluiert.

6.1 Erfüllte Ziele

Alle formalen Anforderungen, wie sie in Sektion 2.2.2 festgehalten sind, wurden erfüllt. Der Demonstrator läuft on-premise auf dem Raspberry Pi und verwaltet sämtliche Daten lokal. Es ist möglich komplexe Szenarien zu definieren, wobei intelligente Geräte mit Webdiensten frei zusammengefügt werden können. Dies ist möglich, da die integrierten Webservices ebenfalls als *Things* im System abgebildet sind. Schließlich ist eine Benutzeroberfläche vorhanden, die es dem Nutzer erlaubt zur Laufzeit neue Szenarien zu erstellen, zu editieren und zu löschen.

6.2 Vergleich mit Smart Home

Bei dem entwickelten Demonstrator handelt es sich um eine klassische Smart Home Lösung, die auf *Eclipse SmartHome* basiert und um weitere Funktionalitäten angereichert wurde. Dadurch verfügt er über alle üblichen Funktionalitäten, die in einem Smart Home enthalten sind - er ist in der Lage ausgewählte intelligente Geräte direkt anzusteuern und in Szenarien zu automatisieren.

6.3 Vergleich mit webbasierten Task Automation Services

Einen Einblick, wie sich der entstandene Demonstrator in die aktuell existierenden Task Automation Services eingliedern lässt, verleiht Abbildung 6.1. Wie zu sehen ist, gibt es Verbesserungen in vielen Aspekten gegenüber anderen Lösungen. Gegenüber den webbasierten TAS grenzt sich FLASH vor allem durch seine Fähigkeit ab intelligente Geräte

			Web						Smartphone				Home	
		FLASH	Ifttt	Zapier	CloudWork	Elastic.io	ItDuzzit	Wappwolf	On{x}	Tasker	Atooma	AutomateIt	WigWag	Webee
Channels	Web channel support	✓	✓	✓	✓	✓	✓	Few	Few	✓	✓	✓	✓	✓
	Device channel support	✓	Few	×	×	×	×	×	✓	✓	✓	✓	✓	✓
	Smartphone resources as channels	×	✓	×	×	×	×	×	✓	✓	✓	✓	×	×
	Public channels support	✓	✓	×	×	✓	✓	×	✓	×	×	×	×	×
	Pipe channel support	✓	×	×	×	✓	×	Few	Few	×	×	×	×	×
	Group channel support	NA	×	×	×	×	×	×	×	×	×	×	Few	×
Rules	Device channel discovery	✓	×	×	×	×	×	×	×	×	×	×	✓	Few
	Multi-event rules	✓	×	×	×	×	×	×	✓	✓	×	×	✓	×
	Multi-action rules	✓	×	×	×	✓	×	×	✓	✓	×	✓	✓	✓
	Chain rules	✓	×	×	×	✓	×	Few	Few	×	×	×	×	×
	Group rules	NA	×	×	×	×	×	×	×	×	×	×	Few	×
	Collision handling	×	×	×	×	×	×	×	×	×	×	×	×	×
	Predefined common rules	✓	×	×	✓	×	✓	✓	×	×	×	✓	✓	✓
	Rule execution profile	DD	WD	WD	WD	WD	WD	WD	DD	DD	DD	DD	DD	DD
TAS	Visual rule editor	×	✓	✓	×	✓	✓	✓	×	✓	✓	✓	✓	✓
	Provides API	✓	×	✓	×	✓	×	×	×	×	×	×	×	×
	Programming language	✓	×	×	×	✓	×	×	✓	Few	×	✓	✓	✓

* ✓ = supported; × = not supported; Few = few support; WD = Web-driven execution profile; and DD = device-driven execution profile; NA = not applicable

Abbildung 6.1: Vergleich des Demonstrators mit existierenden Task Automation Services

direkt anzusteuern, sowie der deutlich mächtigeren Rule Engine ab. Es ist möglich komplexe ECA-Szenarien zu definieren, mit beliebig vielen Triggern, Bedingungen und Befehlen. Teile des Szenarios werden sequentiell abgearbeitet und können als Eingabe für spätere Elemente dienen.

Der entstandene Demonstrator wurde einem umfassenden Test unterzogen, im Laufe dessen Durchsatz und Reaktionszeiten gemessen und mit IFTTT verglichen wurden.

6.3.1 Kopieren von Dateien in der Dropbox

Um Durchsatz zu messen, wurde ein Szenario definiert, dass sämtliche Dateien, die in einen bestimmten Ordner in Dropbox hinzugefügt wurden, in einen anderen Ordner in der Dropbox zu kopieren. Zu Beachten ist, dass IFTTT sich auf die Abarbeitung von bis zu 15 Dateien pro Abfrage begrenzt und Dateien, die größer, als 30 MB sind, nicht beachtet. Zusätzlich wird vom Service gewarnt, dass die Ausführung aller Szenarien sich um bis zu 1 Stunde verzögern kann. Diese Tatsache führt zur Streuung, die in den nachfolgenden Grafiken festzustellen ist.

Der Demonstrator lief auf einem Raspberry Pi 3 Model B, der an einen Router mit VDSL (50 MBit) Verbindung angeschlossen war.

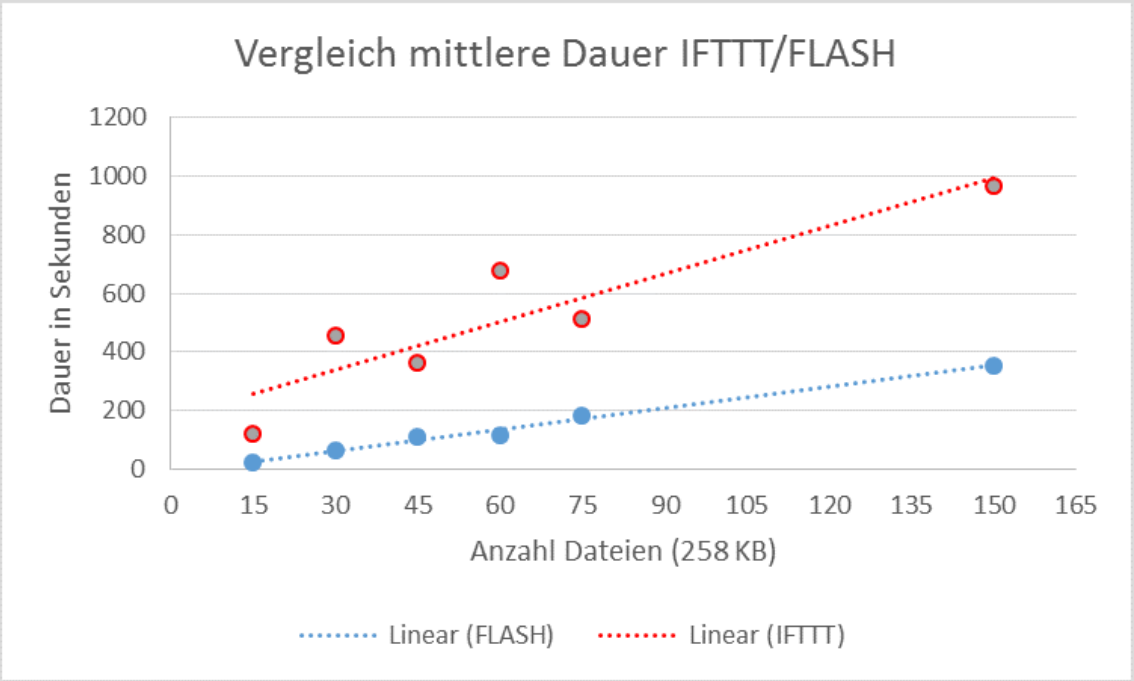

Abbildung 6.2: Vergleich des Mittelwerts zwischen IFTTT und FLASH für unterschiedliche Dateimengen

Es wurde geprüft, wie gut die beiden Anwendungen mit einer großer Anzahl kleiner Dateien, sowie mit geringer Anzahl von großen Dateien umgehen können.

Zahlreiche kleine Dateien

Es wurden Mengen von 258 KB großen Dateien gleichzeitig in einen Dropbox Ordner hochgeladen. Es wurden jeweils Vielfache von 15 verwendet, da dies die maximale Anzahl von Dateien ist, die IFTTT in einer Abfrage bearbeitet.

Es entstanden die Grafiken 6.2 und 6.3.

Für die Erstellung von Grafik 6.2 wurde für jede hochgeladene Datei einzeln gemessen, wie viele Sekunden es dauert, bis eine Kopie im Zielordner vorhanden ist. Es wurde anschließend ein Mittelwert gebildet und die Messung mehrmals wiederholt. Schließlich wurde ein gemeinsamer Mittelwert über die gesammelten Werte gebildet und in der Grafik für die verschiedenen Mengen von Dateien dargestellt.

In der Grafik 6.3 wurde analog gehandelt, mit dem Unterschied, dass hier dargestellt ist, was die längste Dauer zwischen hochladen einer Datei und der Bereitstellung der Kopie ist. Da alle Dateien nahezu gleichzeitig hochgeladen werden, kann man diesen Wert auch als Gesamtdauer, bis alle Dateien verschoben wurden, betrachten.

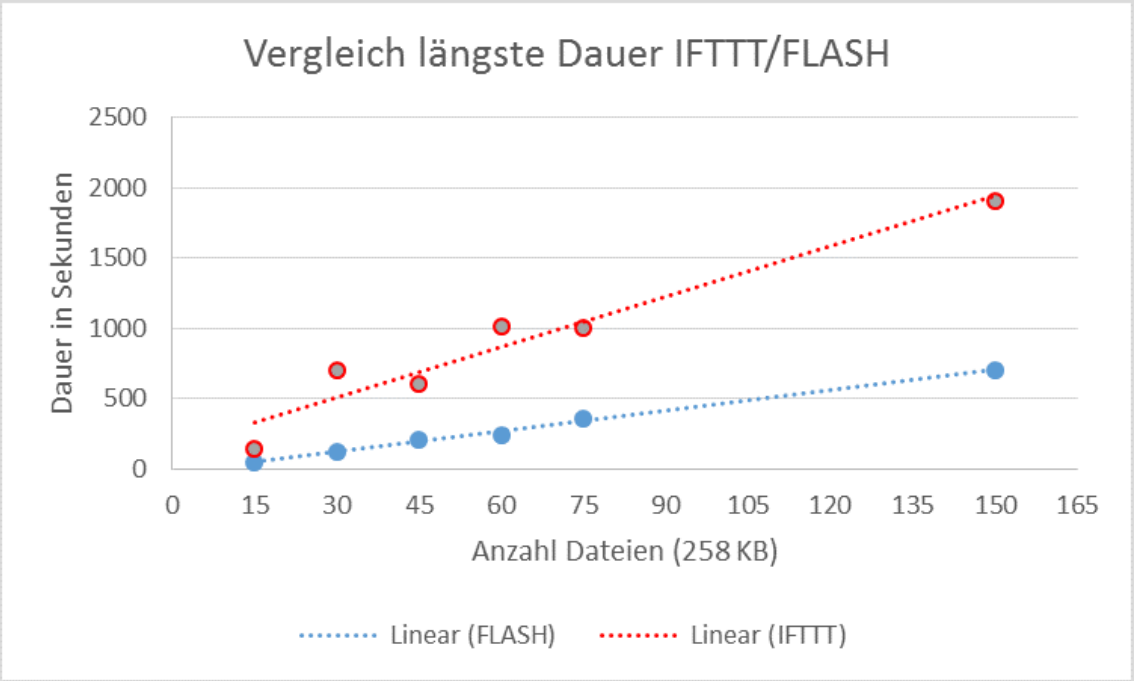

Abbildung 6.3: Vergleich der längsten Dauer (equivalent zu Gesamtdauer, da ca. 1-2 Sekunden vernachlässigt werden können) zwischen IFTTT und FLASH für unterschiedliche Dateimengen

Große Dateien

IFTTT ist in der Lage mit Dateien umzugehen, die bis zu 30 MB groß sind. Eine Wiederholung der oben genannten Messung mit 25 MB großen Dateien hat ergeben, dass sich die Ergebnisse analog verhalten.

Der Demonstrator ist in der Lage mit größeren Dateien umzugehen. Im Test gab es auch bei Verschiebung von 300 MB großen Dateien keinerlei Probleme.

6.3.2 Steuern von Lampen bei Tweets

Es wurde sowohl auf IFTTT als auch auf dem Demonstrator ein Szenario erstellt, dass bei jedem Tweet eine Philips Hue Lampe ausschaltet. Anschließend wurde gemessen, wie lange es dauert, bis die Lampe tatsächlich ausgeschaltet wird, nachdem der Tweet veröffentlicht wurde. Dabei steuert ITTTT die Lampe über die von Hue bereitgestellte Webschnittstelle an, während der Demonstrator das Gerät über WLAN bedient.

Es hat sich ergeben, dass die Reaktionszeit des Demonstrators stets unter einer Sekunde lag, während die Schaltung durch IFTTT eine sehr variable Dauer aufwies. Die schnellste Reaktionszeit lag bei ungefähr 10 Sekunden.

6.3.3 Auswertung

Vorteile

Aus den Messungen hat sich gezeigt, dass der Demonstrator IFTTT in vieler Hinsicht deutlich voraus ist. Dadurch, dass er sich im Hause des Nutzers befindet, ist er in der Lage mit Geräten direkt zu kommunizieren, was Grund für merklich bessere Reaktionszeiten in Szenarien, die intelligente Geräte im Haus betreffen, ist.

Auch in Szenarien, die nur Web-basiert sind und viel Download/Upload-Volumen mit sich ziehen, hat sich der Raspberry Pi als überlegen erwiesen. Dies lässt sich dadurch erklären, dass es sich bei dem Demonstrator im Gegensatz zu IFTTT um ein dediziertes Gerät handelt, dass nur einen konkreten User bedient. Dies führt auch dazu, dass er keine arbiträren Eingrenzungen besitzt, wie z.B. die Begrenzung der Dateigröße.

Schließlich ist der Raspberry Pi aus einer Datenschutzperspektive dem Cloud-Service zu bevorzugen, da sämtliche Zugriffsdaten nur lokal auf dem Gerät gesichert werden.

Nachteile

Negative Aspekte des Demonstrators sind Konsequenz der Tatsache, dass es sich dabei um eine on-premise Lösung handelt. Sämtliche Szenarien, die Webdienste automatisieren, funktionieren nur solange eine Internetverbindung vorhanden ist. Sollte sie ausfallen, bleiben nur die charakteristischen Smart Home Funktionalitäten erhalten. In diesem Aspekt ist IFTTT dem Demonstrator voraus, da die Ausfallsicherheit in einem Rechenzentrum gegenüber der eines herkömmlichen Heimanschlusses weit überlegen ist.

Die Tatsache, dass sämtliche Zugriffsdaten auf Accounts des Users ausschließlich lokal auf dem Gerät vorhanden sind, zieht mit sich, dass im Falle eines Ausfalls (z.B. Hardware-Fehler) sämtliche Daten verloren gehen können. Dies könnte dazu führen, dass sämtliche Geräte, Dienste und Szenarien nach einer Reparatur komplett neu aufgesetzt werden müssten.

6.4 Eclipse SmartHome und Quellcode

6.4.1 Positive Aspekte

Im Laufe der Arbeit is klar geworden, dass das Eclipse SmartHome Framework sich gut als Grundgerüst für unterschiedlichste Lösungen handelt. Die sehr generische Struktur des Frameworks hat es ermöglicht, die im Rahmen der Arbeit implementierten Funktionalitäten nahtlos in das Gesamtkonstrukt zu integrieren.

Dies, sowie OSGi im Allgemeinen, ermöglicht es weitere Funktionalitäten (z.B. weitere Webdienste) der Anwendung hinzuzufügen, ohne bereits existierenden Code ändern zu müssen. Dank der Option, Szenarien im JSON-Format zu definieren, ist es sogar begrenzt möglich, neu hinzugefügte Webdienste in Regeln zu automatisieren, ohne die Benutzeroberfläche anzupassen.

6.4.2 Negative Aspekte

Eclipse SmartHome

Negativ aufgefallen ist die mangelnde Dokumentation von ESH, die den Einstieg in das Framework deutlich erschwert. Verstehen der einzelnen Funktionalitäten bedarf der Betrachtung des Quellcodes, wobei relevante Stellen unter den mehr als hundert Bundles erst noch identifiziert werden müssen.

Eclipse SmartHome befindet sich derzeit noch auf Version 0.9.x und wird aktiv weiterentwickelt. Dies hat unvermeidbar dazu geführt, dass an einigen Stellen der Code nicht fehlerfrei ist. Im Laufe der Implementierung mussten an mehreren Stellen Bugs zunächst ausfindig gemacht und danach behoben werden.

Quellcode

Vor allem der Mangel eines visuellen Regeleditors begrenzt derzeit die Nutzungsmöglichkeiten des Demonstrators. Zwar ist es möglich, Szenarien zur Laufzeit zu editieren, so bedarf dies dennoch Kenntnissen von JSON, sowie den existierenden Event-Typen, sowie der Eingaben und Ausgaben von Triggern, Bedingungen und Aktionen.

6.5 Fazit

Die Implementierung erfüllt alle explizit definierten Anforderungen an die Funktionsweise und weist deutliche Verbesserungen in wesentlichen Aspekten gegenüber der Konkurrenz auf. Sie lässt sich leicht um neue Elemente und Funktionalitäten erweitern.

Kapitel 7

Zusammenfassung

7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde die Kompatibilität von Smart Home mit webbasierten Task Automation Services überprüft. Es stellte sich heraus, dass diese beiden Ansätze viele Gemeinsamkeiten haben und durch ihre Integration miteinander sich viele Vorteile gewinnen lassen.

Zu erwarten war, dass durch die Erweiterung einer Smart Home Anwendung um die Funktionalitäten von TAS die inhärenten Sicherheitsrisiken eines webbasierten Task Automation Services vermeiden lassen. Außerdem ließ sich erhoffen, dass ein dediziertes Gerät Verbesserungen in Punkten Reaktionszeit und Durchsatz aufweisen würde.

Diese Annahmen wurden in der Praxis getestet. Hierzu wurde das Open Source Framework Eclipse SmartHome als Grundlage verwendet und auf dessen Basis ein vollständiger Prototyp entworfen und implementiert. Im Prototypen wurden die Webservices Twitter, Dropbox, ein Wetter- und ein Emaildienst integriert. Der Code wurde möglichst generisch gehalten, sodass Erweiterungen ohne großen Aufwand eingebaut werden können.

Im Laufe der Evaluationsphase haben sich die zugrundeliegenden Annahmen vollständig bestätigt. Es hat sich gezeigt, dass der Prototyp einer herkömmlichen Smart Home Lösung überlegen ist, da er vergleichbare Reaktionszeiten aufweist und gleichzeitig um zusätzliche Funktionalitäten verfügt. Gleichzeitig ist er auch den webbasierten Task Automation Services in Punkten Reaktionszeit, Durchsatz und Sicherheit voraus.

7.2 Ausblick

Es hat sich gezeigt, dass die notwendige Hardware durch eine typische Smart Home Basis bereits gegeben ist. Auch was den Software Aspekt betrifft, sind viele notwendige Elemente bereits vorhanden. Es ist daher etwas verwunderlich, dass bis dato weder die proprietären noch die open source Smart Home Lösungen die Automatisierung von Webdiensten anbieten.

Diese Tatsache lässt sich zum Teil durch die sehr starke Fragmentierung des Marktes erklären. Typische kommerzielle Smart Home Lösungen unterstützen nur einen kleinen Bruchteil aller existierenden intelligenten Geräte und priorisieren diesen Aspekt, wenn es um die Weiterentwicklung geht. Dennoch lässt sich vermuten, dass in Zukunft, nachdem sich die IoT Landschaft stärker standardisiert hat, die Integration von Webservices in Smart Home etablieren wird.

In diesem Zusammenhang kann Ziel zukünftiger Arbeiten werden, weiter in die Richtung der Integration von SmartHome mit Task Automation Services zu forschen. Beispielsweise wäre ein einfacher, aber gleichzeitig mächtiger visueller Regeleditor für den Erfolg eines solchen Unternehmens von großer Wichtigkeit.

Ein anderer möglicher Ausbaupunkt für einen solchen Workflow Automation Service wäre die automatische Generierung von Szenarien. Sofern der Nutzer ein derartiges System in seinem Haus hat und der Anwendung die nötigen Zugriffsrechte auf seine Accounts gewährt hat, könnte die Anwendung das Verhalten des Nutzers mitverfolgen und basierend auf einer erhobenen Statistik automatisch eigene Szenarien vorschlagen.

Anhang A

Weitere Informationen

Der Quellcode der Implementierung befindet sich auf der mit der Arbeit abgegebenen CD.

Abbildungsverzeichnis

Literaturverzeichnis

[1] *bnd*. Available online at `http://bnd.bndtools.org/`; visited on February 19th, 2017.

[2] *Bootstrap*. Available online at `http://getbootstrap.com/`; visited on February 19th, 2017.

[3] *Concierge*. Available online at `http://www.eclipse.org/concierge/`; visited on February 19th, 2017.

[4] *Dropbox*. Available online at `https://www.dropbox.com/`; visited on February 6th, 2017.

[5] *Eclipse SmartHome Homepage*. Available online at `https://www.eclipse.org/smarthome/`; visited on February 6th, 2017.

[6] *Eclipse SmartHome Packaging Sample*. Available online at `https://github.com/eclipse/smarthome-packaging-sample`; visited on February 19th, 2017.

[7] *Google Mail*. Available online at `https://mail.google.com/`; visited on February 19th, 2017.

[8] *MapDB*. Available online at `http://www.mapdb.org/`; visited on February 19th, 2017.

[9] *openHAB*. Available online at `http://www.openhab.org/`; visited on February 19th, 2017.

[10] *OpenWeatherMap*. Available online at `https://openweathermap.org/api`; visited on February 6th, 2017.

[11] *Philips Hue*. Available online at `www.meethue.com/`; visited on February 19th, 2017.

[12] *Qivicon*. Available online at `https://www.qivicon.com/`; visited on February 19th, 2017.

[13] *Raspberry Pi*. Available online at `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`; visited on February 19th, 2017.

[14] *RWE SmartHome.* Available online at `http://www.rwe-smarthome.de/`; visited on February 6th, 2017.

[15] *Twitter.* Available online at `https://twitter.com/`; visited on February 6th, 2017.

[16] *Twitter4J.* Available online at `http://twitter4j.org/`; visited on February 19th, 2017.

[17] *IFTTT.* Website, 2017. Available online at `http://ifttt.com`; visited on February 6th, 2017.

[18] *The OSGi Architecture.* Website, 2017. Available online at `https://www.osgi.org/developer/architecture/`; visited on February 6th, 2017.

[19] *Zapier.* Website, 2017. Available online at `http://zapier.com/`; visited on February 6th, 2017.

[20] Bakker, Paul und Bert Ertman: *Building Modular Cloud Apps with OSGi.* Ö'Reilly Media, Inc.", Sebastopol, 2013.

[21] Beer, Wolfgang, Volker Christian, Alois Ferscha und Lars Mehrmann: *Modeling Context-Aware Behavior by Interpreted ECA Rules*, Seiten 1064–1073. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[22] Bhadauria, Rohit und Sugata Sanyal: *Survey on Security Issues in Cloud Computing and Associated Mitigation Techniques.* CoRR, abs/1204.0764, 2012.

[23] Botta, A., W. de Donato, V. Persico und A. Pescapé: *On the Integration of Cloud Computing and Internet of Things.* In: *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, Seiten 23–30, Aug 2014.

[24] Boyd, Ryan: *Getting Started with OAuth 2.0.* Ö'Reilly Media, Inc.", 2012.

[25] Coronado, M. und C. A. Iglesias: *Task Automation Services: Automation for the Masses.* IEEE Internet Computing, 20(1):52–58, Jan 2016.

[26] European Research Cluster on the Internet of Things: *IERC Projects Portfolio.* https://www.smart-action.eu/publications/detail/114/90c9734fda7c5c2c68e631bf29a9a9d2/.

[27] Gedeon, Walid Joseph: *OSGi and Apache Felix 3.0 Beginners Guide.* Packt Publishing, 2010.

[28] Gyrard, A., S. K. Datta, C. Bonnet und K. Boudaoud: *Cross-Domain Internet of Things Application Development: M3 Framework and Evaluation.* In: *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, Seiten 9–16, Aug 2015.

[29] Gyrard, A., M. Serrano und G. A. Atemezing: *Semantic web methodologies, best practices and ontology engineering applied to Internet of Things.* In: *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, Seiten 412–417, Dec 2015.

[30] Jacobs, Tobias, Markus Joos, Carsten Magerkurth et al.: *Adaptive, faulttolerant orchestration of distributed IoT service interactions.* Technischer Bericht D2.5, The Internet of Things - Architecture, 2012.

[31] Menoret, Stephane et al.: *iCore - Final architecture reference model.* Technischer Bericht D2.5, iCore Project, 2014.

[32] O'Brien, Tim, Manfred Moser et al.: *Maven: The Complete Reference.* Sonatype. Available online at `https://books.sonatype.com/mvnref-book/reference/public-book.html`; visited on February 19th, 2017.

[33] Ran, Chen, Baoan Li und Jianjun Yu: *CEIS 2011 Research and Application on the Smart Home Based on Component Technologies and Internet of Things.* Procedia Engineering, 15:2087 – 2092, 2011.

[34] Singh, Inderjeet, Joel Leitch und Jesse Wilson: *Gson User Guide*, 2008. Available online at `https://sites.google.com/site/gson/gson-user-guide`; visited on October 25th, 2013.

[35] Sriparasa, Sai Srinivas: *JavaScript and JSON Essentials.* Packt Publishing", 2013.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 20. Februar 2017

Konstantin Tkachuk