JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Eine generische Web-of-Things Plattform für Smart Vending unter besonderer Betrachtung von REST-basierter Architektur und Business Rules

MASTERARBEIT

zur Erlangung des akademischen Grades

MASTER OF SCIENCE (MSC)

im Masterstudium

WIRTSCHAFTSINFORMATIK

Eingereicht von:
Stefan Penner, BSc

Angefertigt am:
Institut für Wirtschaftsinformatik – Data & Knowledge Engineering

Beurteiler:
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Mitbetreuung:
Mag. Dr. Bernd Neumayr

Mitwirkung:
Andreas Neuhauser, BSc

Linz, Juli 2015

Vorwort

Diese Masterarbeit gibt einen Einblick in die Entwicklung einer Web-of-Things-Plattform für komplexe Dinge. Während diese Arbeit insbesondere die REST-basierte Architektur sowie Business Rules diskutiert, fokussiert eine begleitende Arbeit, verfasst durch Andreas Neuhauser, unterschiedliche Multi-Tenancy-Ansätze sowie Templates. Die grundlegenden Abschnitte dieser Arbeit (Kapitel 2 - 5) sowie die konkrete Implementierung wurden von beiden Autoren gemeinsam entwickelt und sind in beiden Arbeiten deckungsgleich, sodass beide Arbeiten voneinander unabhängig gelesen werden können.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 17. Juli 2015

Stefan Penner, BSc

Kurzfassung

Die Fortschritte bei der Miniaturisierung von physischen Geräten und Sensoren in den letzten Jahrzehnten führten zu einem Aufschwung des Pervasive Computing. Das *Web-of-Things*, bei dem eine Vielzahl an alltäglichen Gegenständen als intelligente Objekte mit ihrer Umwelt über das Internet interagieren, erlangte durch die Nutzung von weit verbreiteten und interoperablen Standards wie HTTP, XML oder JSON zunehmende Bedeutung. Im Vergleich zum *Internet-of-Things* liegt der Fokus nicht in der Schaffung von Kommunikationsprotokollen, sondern in der Orchestrierung und Nutzung der Daten für Anwendungen. Die Autoren wurden durch die zunehmende Popularität des Verkaufs von Waren und Dienstleistungen mittels Automaten (gen. Vending) dazu veranlasst ein Smart Vending Szenario zu entwickeln, auf Basis dessen analysiert werden soll, ob und wie eine Integration von Automaten in das Web-of-Things erfolgen kann.

Das Ziel dieser Arbeit ist zu untersuchen wie komplexe Dinge, am Beispiel eines Getränkeautomaten, ins Web-of-Things integriert bzw. in diesem abgebildet werden können. Zunächst wird von den Autoren, unter Mitwirkung von Marktteilnehmern, ein Smart Vending Szenario erarbeitet und dessen Umsetzbarkeit in bestehenden Web-of-Things-Plattformen untersucht. Die durchgeführte Evaluierung zeigte, dass die verfügbaren Web-of-Things-Plattformen intentional nicht für komplexe Anwendungsszenarien entwickelt wurden. Vor allem aufgrund der fehlenden Unterstützung zur Abbildung und Integration komplexer Dinge wird in Folge von den Autoren eine eigene Plattform, namens *WoTCloud*, in Form eines Prototypen vorgeschlagen und entwickelt.

Neben einer allgemeinen Erläuterung der Architektur und Implementierung werden im zweiten Teil der vorliegenden Arbeit spezifische Aspekte der Plattform näher erläutert. Es wird geklärt, warum Web-of-Things-Plattformen auf einen *REST-basierten Architekturstil* setzen. Weiters werden die zugrundeliegenden Services von WoTCloud, die sowohl zur Integration von Sensordaten und Aktuatoren, sowie als API für darauf aufbauende Anwendungen dienen, vorgestellt. Darüber hinaus wird die Integration von *Business Rules* in WoTCloud diskutiert. Es werden sowohl datengetriebene als auch temporale reaktive Regeln vorgestellt und deren Implementierung sowie Einsatzzweck erläutert.

Abstract

Recently there has been a paradigm shift taking place in the field of computing towards miniaturized pervasive objects which are becoming more and more popular in our daily lives. Smart things are digitally enhanced physical objects with communication capabilities. *Web-of-Things* emerged from a lack of standardized communication protocols by reusing well-proven and widely accepted protocols like HTTP, XML or JSON in order to connect smart things to the Web. At the same time Web-of-Things platforms have risen as central data hubs for things offering light-weight integration via REST services. Moreover, vending machines are becoming more attractive and are of great interest in the context of Web-of-Things due to obvious reasons.

The objective of the thesis is to investigate how complex things like vending machines could be integrated into the Web-of-Things. Thus, market players have been interviewed about current shortcomings and challenges resulting in notional requirements concerning a vending operator's business. Subsequently, the authors developed a smart vending scenario and evaluated the suitability of state-of-the-art Web-of-Things platforms. After evaluating four platforms with derived criteria the authors found that all of them are not capable of covering the smart vending scenario sufficiently. Especially the feature to represent and integrate complex things was missing. As a consequence, the authors went further and agreed on building a custom Web-of-Things-platform as part of the thesis.

Apart from a profound introduction of the platform called *WoTCloud*, the thesis highlights two important issues. First, *REST* is discussed as well as the motivation behind Web-of-Things-platforms to be grounded on this architectural notion. Moreover, we discuss how sensor and actuators can be integrated into WoTCloud as well as how further applications can benefit from REST-based webservices. Lastly, different kind of *business rules* are integrated into WoTCloud. As a consequence, vending machine operators can benefit from automating parts of their business processes by creating and maintaining reactive business rules.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

6LowPan	IPv6 over Low power Wireless Personal Area Network
API	Application Programming Interface
CSS	Cascading Style Sheets
CSV	Comma-separated Values
ECA	Event-Condition-Action
EDIFACT	Electronic Data Interchange For Administration, Commerce and Transport
EPC	Electronic Product Code
ERP	Enterprise Resource Planning
EXI	Efficient XML Interchange
IETF	Internet Engineering Task Force
GPS	Global Positioning System
GSM	Global System for Mobile Communications
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet-of-Things
LINQ	Language Integrated Query
JSON	JavaScript Object Notation
NFC	Near Field Communication
RDFS	Ressource Description Framework Schema

REST Representational State Transfer

RFID Radio Frequency Identification

RPC Remote Procedure Call

ROA Resource Oriented Architecture

SAAS Software-as-a-Service

SAC Social Access Controller

SBVR Semantics of Business Vocabulary and Business Rules

SOAP Simple Object Access Protocol

UDDI Universal Description, Discovery and Integration

UMTS Universal Mobile Telecommunications System

URI Uniform Ressource Identifier

URL Uniform Ressource Locator

WoT Web-of-Things

WCF Windows Communication Foundation

WSDL Web Service Description Language

WWW World Wide Web

XSLT Extensible Stylesheet Language Transformation

XML Extensible Markup Language

Kapitel 1

Einleitung

Dieses Kapitel dient der Hinführung zum Thema und der Motivation der Relevanz des Internet der Dinge. Es wird der Kernaspekt und die Zielsetzung der Arbeit vorgestellt sowie ein Ausblick auf die einzelnen Kapitel und die Struktur der Arbeit gegeben.

1.1 Motivation

Das Internet hat sowohl das berufliche als auch das private Leben vieler Menschen auf diesem Planeten umfassend verändert und wird dies auch noch zukünftig tun. Web 2.0, soziale Netzwerke und der Internetzugriff auf mobilen Geräten, wie Smartphones und Tablets, sind nur einige Schlagworte der letzten Jahre. Die Anzahl der Nachrichten und Informationen, die tagtäglich über die unterschiedlichsten Geräte und Kanäle ausgetauscht werden, nimmt rasant zu. Die Entwicklung des *Internet-of-Things*[1] *(IoT)*, einer der Top 10 strategischen Technologietrends 2015 [24], ist eine logische Konsequenz der laufenden Weiterentwicklung von Informations- und Kommunikationstechnologie, Anforderungen aus der Logistik und den Geschäftsmodellen der heutigen Zeit.

Eine Vielzahl von Objekten wird zukünftig eine Identität im Internet haben. Alltägliche Gegenstände werden zu *intelligenten Objekten*, die eigenständig mit ihrer Umwelt interagieren und somit den Menschen in den unterschiedlichsten Lebensbereichen und Arbeitsumgebungen unterstützen. So plant beispielsweise Samsung[2] bis 2020 alle Haushaltsgeräte (Kühlschränke, Waschmaschinen, etc.) nur mehr als intelligente, vernetzte Produkte anzubieten. Die Anwendungsmöglichkeiten des Internets der Dinge sind vielfältig:

[1] Die Begriffe Internet-of-Things und Internet der Dinge werden im Laufe der Arbeit synonym verwendet.

[2] http://futurezone.at/produkte/samsung-will-nur-mehr-vernetzte-haushaltsgeraete-anbieten/111.778.305

vom Smart Home[3] über die Analyse und Steuerung von Industrieanlagen[4] bis hin zu smarten Warenautomaten[5].

Die technische Vision des Internet der Dinge ist seit einigen Jahren Ausgangspunkt für (Jung-)Unternehmen zur Entwicklung von intelligenten Produkten bzw. ein treibender Faktor für viele Projekte in Forschungseinrichtungen. Bereits 2015 soll es laut Gartner [25] 4.9 Milliarden vernetzte Dinge geben. 2020 sollen es gar 20 Milliarden - 2/3 davon im Konsumentenbereich (B2C) - sein, bei einem geschätzten Marktvolumen von 7.1 Billionen Dollar[6]. Ausgangspunkt des Internet-of-Things war vor allem die Etablierung von Near Field Communication (NFC) und Radio Frequency Identification (RFID) Technologien im Konsumentenmarkt [59], kombiniert mit der ständig steigenden Verfügbarkeit als auch Verbreitung von mobilem Internet (vor allem durch Smartphones und Tablets). Durch diese Entwicklungen und den skalierbaren Austausch von Informationen (z. B. Facebook, Twitter), der durch soziale Medien ermöglicht wird, wurde ein enormer Platz für Innovationen im Bereich der Endbenutzer bzw. der Entwicklung von benutzerzentrierten Produkten geschaffen. Die Annäherung von Menschen und Dingen nimmt stetig zu.

Nichtsdestotrotz gibt es in diesem Bereich noch offene Forschungsthemen [2]. So ist neben der fehlenden Standardisierung vor allem auch Sicherheit ein großes Thema, da das Bedrohungspotenzial von 20 Milliarden Geräten enorm ist. Ein logischer Schritt hin zu einer möglichen Standardisierung der Kommunikation von und mit smarten Dingen ist das Web-of-Things (WoT): Die Nutzung des World Wide Web und aller assoziierten Technologien als Basis für die Kommunikation und den Datenaustausch.

1.2 Problemstellung

Das Web-of-Things besteht aus einer Vielzahl von smarten Dingen, die über Webschnittstellen erreichbar sind. Durch die Nutzung von weit verbreiteten und interoperablen Standards wie dem Hypertext Transfer Protocol (HTTP), Extensible Markup Language (XML) oder JavaScript Object Notation (JSON) erlangte das Web-of-Things schnell große Bedeutung, da der Fokus im Vergleich zum Internet-of-Things nicht in der Schaffung von Kommunikationsprotokollen zur Vernetzung, sondern in der Orchestrierung und Nutzung der Daten für weiterführende Anwendungsgebiete liegt. Neben der Vernetzung einzelner physischer Geräte gibt es auch zahlreiche (wissenschaftliche) Projekte (z. B. [8]), die eine Plattform als zentralen Datenhub

[3] http://www.wired.com/2015/03/internet-anything-open-source-smart-home-control/

[4] http://www.linemetrics.com

[5] http://blogs.sap.com/innovation/innovation/smart-vending-machine-the-future-of-technology-01253903.com

[6] http://www.forbes.com/sites/gilpress/2014/08/22/internet-of-things-by-the-numbers-market-estimates-and-forecasts/

für Dinge entwickelten. Hauptfokus der meisten Plattformen ist eine einfache Integration von Dingen und deren Daten über REST-Schnittstellen (Representational State Transfer), sowie die Erstellung von Mashups und das Teilen von Informationen auf sozialen Plattformen. Meist können auf diesen Plattformen simple Sensoren und teilweise auch Aktuatoren mit einigen Datenfeldern angelegt werden. Diese "einfachen" Sensoren sind aber oft nicht ausreichend für komplexere Anwendungsfälle der realen Welt.

Der zentrale Anwendungsfall dieser Arbeit - Smart Vending - stellt solch ein komplexes Szenario dar. Die Bedeutung von Web-of-Things für Märkte wie das Vending ist zweifellos groß[7]. Branchengrößen wie SAP präsentierten jüngst Prototypen von Automaten, die durchgängig neue, intelligente und vernetzte Konzepte umsetzen[8]. Bestehende WoT-Plattformen sind aber nicht dafür konzipiert, beispielsweise einen Getränkeautomaten mit all seinen Sensoren (z. B. Füllstandsensoren pro Schacht, generelle Sensoren) und Aktuatoren (z. B. Preisänderung, Getränk ausgeben) abzubilden um den Anforderungen von Betreibern solcher Automaten gerecht zu werden.

1.3 Zielsetzung und Abgrenzung

Das Ziel dieser Arbeit ist die Abbildung und Integration von komplexen Dingen in einer Web-of-Things-Plattform am Beispiel eines Getränkeautomaten. Spezifisch werden Anforderungen für solch ein Szenario erhoben, eine Auswahl an bestehenden Plattformen hinsichtlich ihrer Eignung zur Abbildung eines solchen Szenarios evaluiert und schließlich eine Architektur für eine eigene Plattform vorgeschlagen und implementiert. In diesem Zusammenhang beschäftigen wir uns mit folgenden Forschungsfragen:

Komplexe Dinge: *Was benötigt eine Web-of-Things Plattform, um verschiedenste Getränkeautomaten abbilden zu können?* Die Autoren adressieren diese Frage in Bezug auf komplexe Dinge (z. B. einen Getränkeautomat), im Vergleich zu "simplen" Dingen (wie beispielsweise einem Lichtschalter oder einem Temperatursensor) und deren Anforderungen. Ziel ist die Erhebung, welche Features eine Plattform benötigt, um komplexe Dinge am Beispiel eines Smart Vending Szenarios verwalten zu können. Basierend darauf soll eine Architektur und eine prototypische Implementierung durchgeführt werden.

Evaluierung von Plattformen: *Unterstützen bestehende WoT-Plattformen das entwickelte Smart Vending Szenario?* Es sollen bestehende Plattformen hinsichtlich ihrer Eignung für ein Smart Vending Szenario evaluiert werden. In diesem Zusammenhang sollen auch weiterführende Szenarien, die

[7] http://webofthings.org/2012/10/15/i-saw-the-future-of-m2m-in-budapest-smart-vending-machines/

[8] http://blogs.sap.com/innovation/innovation/smart-vending-machine-the-future-of-technology-01253903.com

eine Ende-zu-Ende Integration der Geschäftsprozesse von Automatenbetreibern erlauben, basierend auf den Daten der Plattform, ermöglicht werden.

Spezifische Aspekte: *Wie können spezifische Aspekte, die zur Integration von komplexen Dingen benötigt werden, umgesetzt werden?* Aufgrund der Tatsache, dass bestehende Plattformen nicht für Smart Vending geeignet sind, wird im Zuge dieser Masterarbeit eine eigene Plattform vorgestellt. Hauptaspekt ist die Berücksichtigung besonderer Aspekte, die für die Integration und Abbildung von komplexen Dingen notwendig ist. Die Generizität dieser Plattform soll insofern gegeben sein, sodass unterschiedlichste Arten und Typen von Getränkeautomaten berücksichtigt werden sollen. Ein besonderer Fokus soll auf die Architektur und die technische Umsetzung gelegt werden. In dieser Arbeit soll inbesondere geklärt werden, wie eine Web-of-Things-Plattform im Sinne einer *Representational State Transfer-basierten Architektur* realisiert und Things integriert werden können. Darüber hinaus gilt es zu klären, wie durch *Business Rules* ein dynamisches Verhalten ermöglicht wird. In der begleitenden Arbeit von Andreas Neuhauser [44] werden Templates und Multi-Tenancy behandelt.

1.4 Aufbau der Arbeit

Diese vorliegende Masterarbeit ist wie folgt strukturiert:

- *Kapitel 2* beschreibt die grundlegenden Begriffe und Zusammenhänge und führt in die Thematik von Internet- bzw. Web-of-Things ein.
- *Kapitel 3* dient der Beschreibung des Rahmenbeispiels, welches für die Umsetzung der WoT-Plattform relevant ist. Insbesondere werden Unzulänglichkeiten von bestehenden Smart Vending Lösungen diskutiert.
- *Kapitel 4* vergleicht bestehende WoT-Plattformen hinsichtlich ihrer Einsatztauglichkeit für die Abbildung komplexerer Objekte. Basierend auf einem Kriterienkatalog wird eine Auswahl an Plattformen evaluiert und verglichen.
- *Kapitel 5* fokussiert die Anforderungen und die Systemarchitektur der implementierten WoT-Plattform. Es werden die einzelnen Schichten erklärt und mögliche Interaktionsszenarien vorgestellt.
- *Kapitel 6* diskutiert die wesentlichen Konzepte eines REST-basierten Architekturstils von Webservices. Darüber hinaus wird die Implementierung der ressourcenorientierten Architektur für Smart Vending bzw. für die Web-of-Things-Plattform WoTCloud vorgestellt.
- *Kapitel 7* beschäftigt sich mit der Integration von Geschäftsregeln in die Web-of-Things-Plattform. Es werden die Konzepte und Implementierungsdetails von einfachen, reaktiven Regeln vorgestellt.

Teil I

Grundlagen

Kapitel 2

Vom Internet-of-Things zum Web-of-Things

Our vision is to create a "Smart World," that is, an intelligent infrastructure linking objects, information and people through the computer network. This new infrastructure will allow universal coordination of physical resources through remote monitoring and control by humans and machines. Our objective is to create open standards, protocols and languages to facilitate worldwide adoption of this network – forming the basis for a new "Internet of Things". [10]

2.1 Internet-of-Things

2.1.1 Historie

Der Begriff *Internet-of-Things (IoT)* bekam 2003 vermehrte Aufmerksamkeit, als das Auto-ID Center des Massachusetts Institute of Technology (MIT) ihre Vision eines Electronic Product Code (EPC) Netzwerkes zur automatischen Identifikation und Verfolgung der Warenflüsse von Gütern in Supply-Chains vorstellte [59]. Der Begriff Auto-ID subsumiert eine breite Klasse an Identifizierungstechnologien, die in der Industrie zur Automatisierung, zur Reduktion von Fehlern und zur Erhöhung der Effizienz benutzt werden. Diese Technologien umfassen unter anderem Barcodes, Smart Cards, Spracherkennung und biometrische Verfahren. Seit 2003 ist aber vor allem RFID einer der größten Treiber gewesen. David Brock [10] verwendete den Begriff Internet-of-Things jedoch schon 2001 in einem Whitepaper des Auto-ID Centers über ein Namensschema für physische Objekte. In Folge dessen wurde der Begriff in zahlreichen Publikationen, Konferenzbeiträgen und Büchern verwendet. Mattern [40] verweist auch auf das populärwissenschaftliches Buch *„Wenn Dinge denken lernen"* von Neil Gershenfeld, der

darin in sinngemäßer Weise den Begriff verwendet:

> *Es kommt mir so vor, als sei das rasante Wachstum des WWW nur der Zündfunke einer viel gewaltigeren Explosion gewesen. Sie wird losbrechen, sobald die Dinge das Internet nutzen. [26]*

2.1.2 Vision

Das Internet der Dinge ist ein neuartiges Paradigma, welches im Kontext moderner Telekommunikation über Drahtlosverbindungen einen stetigen Zuwachs verbucht. Die grundsätzliche Idee [2] hinter *Internet-of-Things* ist die Präsenz zahlloser meist ortsunabhängiger Geräte oder Objekte, welche meist als Dinge (Things) bezeichnet werden, wie beispielsweise RFID-Tags, Sensoren, Aktuatoren, Mobiltelefone, etc. Diese Dinge werden über ein eindeutiges Adressschema [2] adressiert und somit in die Lage versetzt, mit sich selbst und anderen Geräten/Menschen in ihrer Umgebung zu interagieren, um definierte Ziele zu erreichen. Mit diesem Konzept verschwimmt die Grenze zwischen virtueller und realer Welt zunehmend, da virtuelle Informationen nahtlos mit physischen Objekten verknüpft werden und daraus neuartige Möglichkeiten bzw. Anwendungsfelder geschaffen werden [59]. Uckelmann et al. führen weiter an, dass der Zugriff auf möglichst umfassende Informationen durch Informations- und Kommunikationstechnologien in Echtzeit in einer neuen Art und Weise erfolge [59]: unabhängig von Ort („anywhere“) und Zeit („anytime“). Dies erfordere aber eine offene, skalierbare, sichere und standardisierte Infrastruktur. Ein Umstand, der derzeit (noch) nicht vollständig existiere.

Laut Santucci verweist der Begriff Internet-of-Things - das *Internet der Dinge* - auf eine Vision der Maschinen der Zukunft:

> *in the nineteenth century, machines learned to do; in the twentieth century, they learned to think; and in the twenty-first century, they are learning to perceive – they actually sense and respond. [51]*

Die zentrale Rolle in dieser Vision nehmen *smarte Dinge* ein, die mit Informations- und Kommunikationstechnologie ausgestattet sind und über das Internet sich sowohl miteinander vernetzen, mit Menschen interagieren, als auch durch Sensoren ihren eigenen Kontext wahrnehmen können. Diese digitale Aufrüstung von alltäglichen Gegenständen eröffne laut Mattern [40] viele neue Möglichkeiten und erlaube sowohl die Optimierung bestehender Geschäftsprozesse als auch die Bereitstellung neuer Dienste, die durch „zeitnahe Interpretation von Daten aus der physischen Welt ökonomischen und gesellschaftlichen Nutzen stiften“ [40]. Durch die Anbindung an das Internet lasse sich zudem der Zustand von Dingen in bisher unerreichter Granularität und Zeit zu „fast verschwindenden Kosten [messen]“ und somit könne man in vielen Aspekten der realen Welt einen Nutzen stiften.

2.1.3 Evolution

Bereits in der Vergangenheit gab es viele Anwendungen von Auto-ID Technologien in Produktion und Logistik um sowohl unternehmensintern als auch unternehmensextern Informationen auszutauschen. Die meisten RFID-Installationen könnten in diesem Sinne als Intranet oder Extranet der Dinge bezeichnet werden. Während mit Intranet der Informationsfluss im Unternehmen gemeint ist, stehen beim Extranet der Dinge [59] traditionelle Kommunikationsmechanismen im Fokus, wie beispielsweise Electronic Data Interchange For Administration, Commerce and Transport (EDIFACT), welches vor allem dazu verwendet wird, um mit einigen wenigen definierten Partnern kommunizieren zu können. Die Autoren beschreiben das Internet der Dinge als logische Evolution des Extranets der Dinge. Basierend auf den bestehenden Ansätzen wurde die Skalierbarkeit durch das Internet und Web 2.0 entscheidend erhöht. Mit dem Internet der Dinge wird ermöglicht, dass global Informationen ausgetauscht und verwendet werden. Durch zunehmend neue Applikationen und eine breitere Akzeptanz wird auch die Pervasivität erhöht. Abbildung 2.1 veranschaulicht die Phasen vom Intranet der Dinge zu einem Internet der Dinge und versucht, auch eine zukünftige Integration von Dingen und Menschen zu verdeutlichen. Laut Uckelmann et al. [59] müssen vor allem aber die Kosten in Relation zum Nutzen für den jeweiligen Geschäftsfall stimmen sowie die Benutzerfreundlichkeit erhöht werden, um breitere Akzeptanz zu schaffen, sodass eine einfache Integration und Kommunikation von Ding und Mensch ermöglicht wird.

2.1.4 Technische Grundlagen

Das Internet der Dinge basiert laut Mattern [40] nicht auf einer einzelnen konkreten Technologie bzw. Funktionalität, sondern vielmehr auf einer Kombination und Ergänzung von teilweise konvergierenden Technikentwicklungen, die in ihrer Gesamtheit neuartig sind. Zu diesen Funktionen gehören [40]:

- *Kommunikation und Kooperation:* Damit ist die Vernetzung der Objekte zum Zweck des Datenaustausches bzw. zur Nutzung von Diensten in lokalen Netzen oder über das Internet gemeint. Voraussetzung dafür sind funkbasierte Technologien (z. B. Global System for Mobile Communications (GSM), Universal Mobile Telecommunications System (UMTS), Wifi, Bluetooth) und spezielle Weiterentwicklungen in diesem Bereich (z. B. Zigbee, IPv6 over Low power Wireless Personal Area Network (6LowPan)), die auf ressourcenarme Geräte/Dinge abzielen.
- *Adressierbarkeit:* Dinge sind nur dann relevant, wenn sie auch gefunden und deren Daten aus der Ferne konsumiert werden können. Folglich ergibt sich ein Bedarf nach einem (einheitlichen) Discovery-, Lookup-

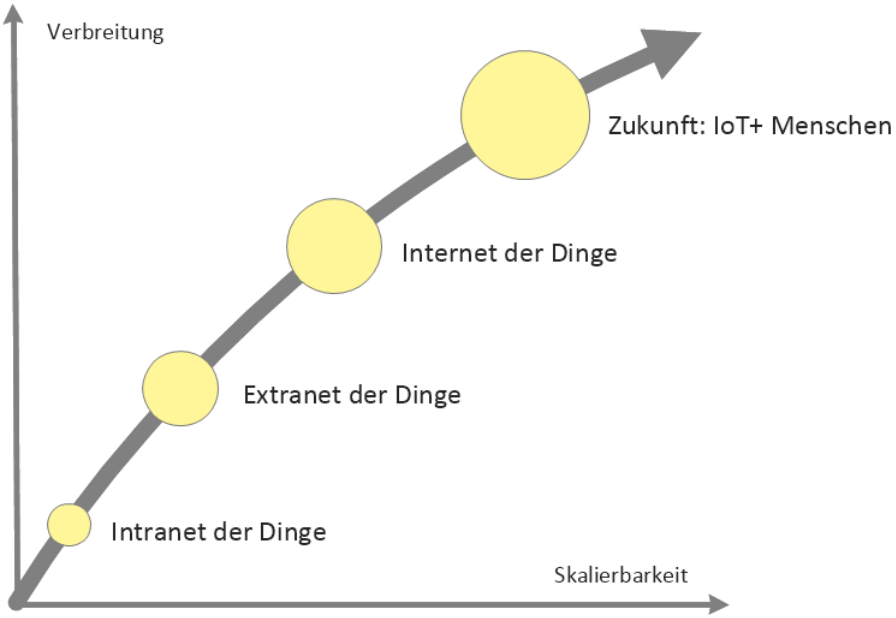

Abbildung 2.1: Die Evolution von Intranet der Dinge zu einer zukünftigen Vision des Internet der Dinge und Menschen (angelehnt an [59]).

oder Namensdienst.

- *Identifikation:* Ein wesentliches Kriterium ist die eindeutige Identifizierbarkeit von Dingen. Alleine aufgrund des historischen Kontexts von Internet-of-Things stellen RFID, NFC oder optische Barcodes Beispiele für relevante Technologien dar. Oft ist ein Vermittler (RFID-Leser), der auch passive Dinge identifiziert, beteiligt. Durch die Identifikation ist eine Verknüpfung von Informationen mit Objekten möglich, auch wenn diese wie im Fall von passiven RFID-Tags nur über den Vermittler mit einem Server verbunden sind.
- *Sensorik:* Objekte sammeln Daten über Ereignisse ihrer Umgebung durch Sensoren. Anschließend werden diese Daten aufgezeichnet oder weiterverarbeitet.
- *Effektorik*: Durch Aktuatoren können Objekte auf ihre Umwelt einwirken und somit sind auch Szenarien denkbar, die eine Beeinflussung und Steuerung von Dingen aus der Ferne (Remote) zulassen.
- *Eingebettete Informationsverarbeitung:* Durch Prozessoren, Mikrocontroller und Speicher werden Objekte zu Smart Things. Diese Bauteile ermöglichen eine Verarbeitung und Speicherung der Sensordaten. Smart Things können also auch ein Gedächtnis besitzen.
- *Lokalisierung:* Durch Global Positioning System (GPS), Mobilfunknetze oder ähnliche Technologien (Ultraschallmessungen, optische Tech-

nologien, ...) können Dinge ihre physische Lokation bestimmen und sind auch für andere auffindbar.

- *Benutzungsschnittstelle:* Smarte Objekte ermöglichen die direkte oder indirekte (beispielsweise über Smartphones) Kommunikation mit Menschen. Wichtig ist dabei, dass die Interaktion in möglichst einfacher und intuitiver Art und Weise erfolgt. Denkbar sind auch innovative Interaktionsparadimgen wie beispielsweise „tangible user interfaces“ oder aber auch Methoden aus den Bereichen Sprach-, Bild- und Gestenerkennung.

2.1.5 Standardisierung und Interoperabilität

Einer der größten Kritikpunkte der vielen Internet-of-Things Entwicklungen der letzten Jahre war und ist die fehlende Standardisierung der Kommunikation, die vor allem durch vergleichsweise geringe vorhandene Ressourcen und Leistungen der Things erschwert wird. Obwohl es einige Bemühungen und Initiativen in diese Richtung gibt, basieren die meisten Projekte auf den unterschiedlichsten Ausprägungen von Hard- und Software, die untereinander inkompatibel sind. Folglich koste laut Guinard & Trifa [31] auch die Entwicklung von simplen Anwendungen aufgrund der heterogenen Systeme noch sehr viel Zeit und benötigt tiefgehende Kenntnisse in spezialisierten Bereichen. Für jedes Deployment müsse man weiters sowohl Basisfunktionen als auch applikationsspezifische Interfaces anpassen, anstatt sich zur Gänze der Applikationslogik widmen zu können.

Aufgrund dessen, dass es noch keine klar spezifizierten, standardisierten, verbreiteten und interoperablen Kommunikationsprotokolle gab, entwickelte sich 2008 ein neues Paradigma, das vollständig auf Webstandards setzt, um Things untereinander und mit ihrer Umwelt zu vernetzen: das *Web-of-Things*.

2.2 Web-of-Things

Die zunehmende Vernetzung von Geräten über das Internet führte dazu, dass als logische Konsequenz - dem Web-of-Things - das World Wide Web und assoziierte Technologien als technische Basis für Smart Things dienen [32]. Das Web ist ein überzeugendes Beispiel für ein skalierbares weltweites Computernetz, in dem heterogene Hardware- und Softwareplattformen ohne Integrationsprobleme zusammenarbeiten. Bereits 2002 wurden physische Objekte über Barcodes mit Webseiten verlinkt, um zusätzliche Informationen und Services abzurufen [36]. Um diesen Ansatz zu erweitern, kann man Smart Things in standardisierte Webservice-Architekturen, durch Nutzung von Standards wie Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) & Universal Description, Discovery and Inte-

gration (UDDI) einbetten [32]. Dieser Ansatz ist in den meisten Fällen aber sehr schwierig umzusetzen, da die genannten Protokolle für die Anwendung mit simplen, ressourcenarmen Geräten viel zu schwergewichtig und komplex sind [33] und in puncto Bandbreite, CPU und Speicherplatz die Ressourcen der Geräte nicht ausreichen. Aus diesem Grund hat sich vor allem der REST-basierte Architekturstil [19] mit lose gekoppelten Webservices und somit die virtuelle Abbildung von physischen Dingen als ressourcenorientierte Architektur (Resource Oriented Architecture (ROA)) angeboten [31, 32].

2.2.1 Web-of-Things Schichtenmodell

Um die Integration von smarten Dingen mit bestehenden Services im Web zu erleichtern, entwickelte Dominique Guinard in seiner Dissertation [29] eine vierschichtige Applikationsintegrationsarchitektur, die die Entwicklung von Anwendungen mit smarten Dingen vereinfacht (siehe Abbildung 2.2). Diese Architektur soll vor allem die Eintrittsbarriere für Entwickler und technisch versierte Anwender verringern, einen direkten Zugriff über einen Webbrowser oder einen HTTP-Client ermöglichen und einen leichtgewichtigen Zugang zu Daten anbieten.

Obwohl Applikationen auf allen 4 Schichten aufgesetzt werden können, bietet jede höhere Schicht breitere Zugangsmöglichkeiten als ihre untergeordneten. So ist die Verfügbarkeitsschicht als Basisschicht zu verstehen und die Kompositionsschicht logisch gesehen als höchste Schicht, die von Anwendungen benutzt werden kann. In der Folge werden die einzelnen Schichten und ihre Grundgedanken kurz beschrieben.

Device Accessibility (Verfügbarkeit)

Die Verfügbarkeitsschicht (Device Accessibility) dient als Basisschicht und adressiert alle Themen rund um einen konsistenten Zugriff auf smarte, vernetzte Objekte. Guinard bedient sich dazu einer ressourcenorientierten Architektur [29] und behandelt Smart Things somit als „first-class citizen“ analog zu Webseiten. Physische Dinge werden als logische Ressourcen hierarchisch modelliert und sind über eine Uniform Ressource Identifier (URI)-Struktur eindeutig adressierbar [31, 32]. Darüber hinaus dienen diese URIs aber auch der Vernetzung von Dingen. Durch die Benutzung von HTTP als Applikationsprotokoll und nicht nur als Transportprotokoll, wie beispielsweise in WS*-Szenarien, erfolgt die Kommunikation zustandslos und verfügt neben der HTTP-Verben GET, POST, PUT und DELETE auch über ein einheitliches Interface [29, 61]. Die konkrete Repräsentation der Ressourcen wird üblicherweise von Client und Server ausgehandelt (Content Negotation) und erfolgt meist in XML oder JSON.

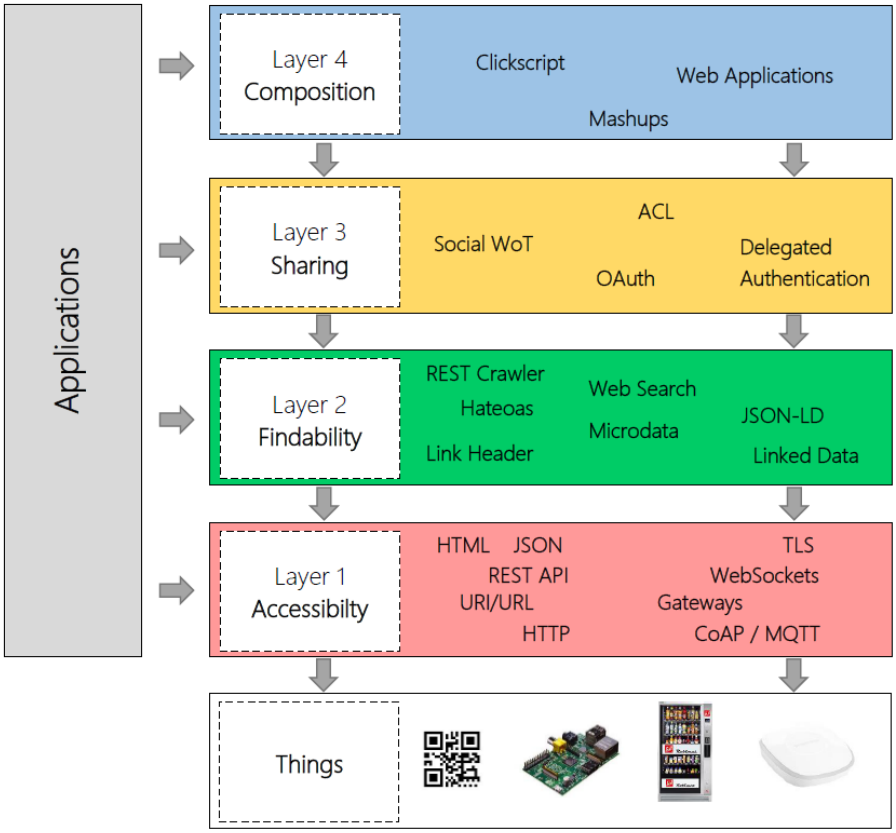

Abbildung 2.2: Die 4 Schichten einer Web of Things Architektur (angelehnt an [29, 30]).

Findability (Auffindbarkeit)

Ausgehend von der Verfügbarkeit einer REST-Schnittstelle von Smart Things gilt es natürlich, deren Services auch zu finden und zu integrieren. Guinard [29] propagiert in diesem Zusammenhang ein Beschreibungsmodell für Smart Things und deren Services basierend auf Metadaten. Nutzvolle Metadaten könnten laut ihm beispielsweise aus der ressourcenorientierten Abbildung der Dinge durch die REST-Architektur extrahiert werden. Anhand einer Implementierung mit Microformats zeigt er auf, dass auch eine Integration in bestehende Suchmaschinen ermöglicht werden kann. Ein großer Nachteil von mobilen Geräten sei jedoch, dass sich der Standort ständig ändern könne. Aus diesem Grund beschreibt Guinard auch eine weborientierte Lookup-Infrastruktur für das Auffinden von Dingen, die auf mehreren lokalen Lookup-Infrastrukturen basiert [29]. Das Ziel der Auffindbarkeit ist also sowohl Menschen als auch Applikationen zu ermöglichen, Services von je-

nen Smart Things über das Web zu konsumieren, die sie in ihrem aktuellen Kontext benötigen.

Sharing (Teilbarkeit)

Basierend auf den ersten beiden Schichten werden die smarten Geräte theoretisch ohne Restriktionen öffentlich über das Web zugänglich. Dies führt dazu, dass auch das Teilen von Informationen (z. B. Sensordaten) erleichtert wird. So könnte beispielsweise der Energieverbrauch durch Sensoren im Haus mit einer Community geteilt werden. Diese Umstände führen jedoch zu gravierenden Eingriffen in die Privatsphären und erfordern deshalb gewisse Restriktionen. Guinard [29] beschreibt deshalb einen selektiven Mechanismus zum Teilen von gewünschten Informationen über einen Social Access Controller (SAC). Dieser ermöglicht auch die Integration von sozialen Netzwerken (z. B. für Werbezwecke) und führt somit zu einem *Social Web-of-Things*. Ein weiteres Anwendungsfeld ist die Aggregation von Daten bzw. Ereignissen zu Feeds. In diesem Zusammenhang kann der SAC auch zur Syndizierung von Daten mehrerer Dinge dienen.

Composition (Komposition)

In der Kompositionsschicht stellt Guinard das Konzept von physischen Mashups vor. Er definiert dazu 3 verschiedene Entwicklungsansätze [29]:

1. *Manuelle Entwicklung von Mashups:* Der erste Ansatz, das manuelle Entwickeln von Mashups, erlaubt es Entwicklern basierend auf Webtechnologien und den Application Programming Interfaces (APIs) & Daten der Smart Things in einfacher Art und Weise Applikationen zu erstellen, die die ersten 3 Ebenen des Web-of-Things-Architekturmodells ausnutzen.
2. *Widget-basierte Entwicklung von Mashups:* Bei der widget-basierten Entwicklung von Mashups erfolgt die Kommunikation mit den smarten Dingen zentral über ein eigenes Software-Framework, welche als Black Box angesehen werden kann, die die Kommunikation abstrahiert und die Daten transparent liefert. Durch Widgets (meist in Form einer Kombination aus Hypertext Markup Language (HTML) und Javascript-Code) können Domänen-Experten nun Anwendungen erstellen. Bei diesem Ansatz sind keine tiefgreifenden Software-Entwicklungskenntnisse notwendig.
3. *Entwicklung durch Endbenutzer über Mashup-Editoren*: Der dritte Ansatz stellt bereits die Endbenutzer in den Mittelpunkt und erlaubt, dass sich Benutzer über Mashup-Editoren eigene kleine Applikationen zusammenstellen. Diese Mashups werden über visuelle Metaphern und Regeln erstellt und nutzen Webseiten bzw. Smart Things als Datenquellen.

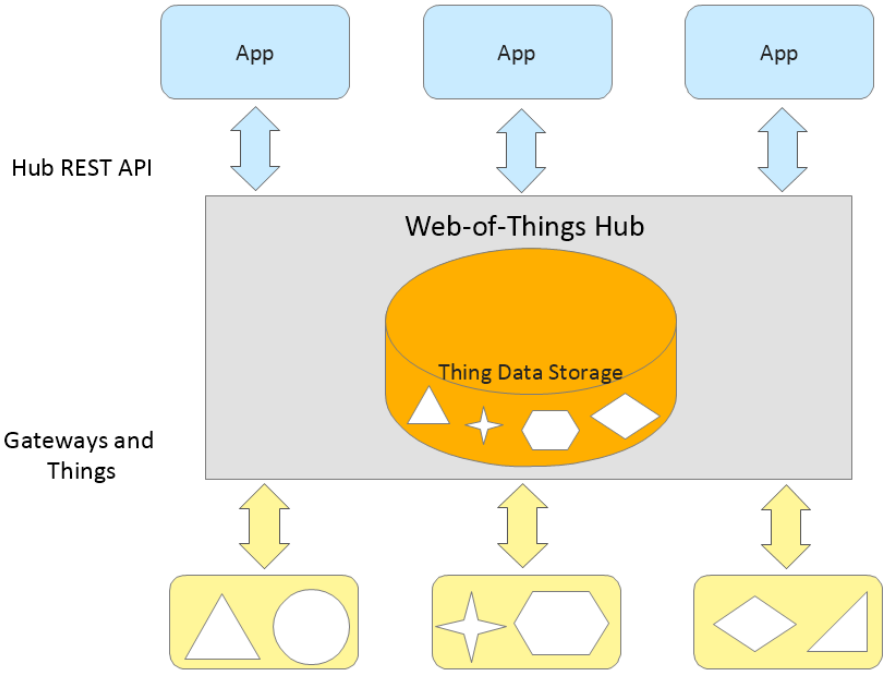

Abbildung 2.3: Konsistente Schnittstellen durch Web-of-Things Hubs (angelehnt an [6]).

2.2.2 Web-of-Things-Plattformen

Während Guinard in seinem Web-of-Things-Architekturmodell (siehe 2.2.1) prinzipiell davon ausgeht, dass jedes Thing einen eigenen Webserver hostet und via REST-API als Datenzugriffspunkt dient, gibt es auch noch andere mögliche Umsetzungsvarianten. Auch die Tatsache, dass Smart Gateways als Mediatoren zwischen Thing und dem Konsumenten dienen, ist in gewissen Anwendungsfällen nicht ausreichend. Ein Kritikpunkt dieser Architektur ist, dass sämtliche Zugriffe direkt auf das ressourcenarme Gerät erfolgen und auch gewisse Security- und Privacy-Mechanismen auf jedem Gerät berücksichtigt werden müssten. Eine weitere Schwachstelle ist der Umstand, dass, obwohl sämtliche Kommunikation über Webstandards erfolgt, es noch keine standardisierte Vorgehensweise gibt, wie man physische Objekte konkret in das Web integriert [6]. Damit ist gemeint, dass Applikationen, die auf den Daten vieler Dinge basieren, auch eine konsistente und interoperable Schnittstelle auf allen Things erfordern würde.

Diese Überlegungen führen zu einem anderen Architekturmodell: Anstatt Smart Things direkt anzusprechen, greift man auf das Konzept eines „Hubs“ zurück, der in Form einer zentralen Plattform die Daten vieler Dinge bündelt [6, 7]. In diesem Zusammenhang ist es auch möglich, dass Smart

Things Daten (z. B. bei einem Ereigniseintritt oder aber auch periodisch) an die Plattform liefern, anstatt über Polling oder Publish-/Subscribe Ansätze abgefragt zu werden. Diese zentralen Plattformen dienen als Basis für spätere Anwendungen und sind im Sinne des Web-of-Things selbst REST-basierte Webanwendungen [7, 8]. Ein Vorteil dieses Ansatzes ist die Reduzierung der Zugriffe auf Smart Things, die nur zentral durch die Plattform oder im umgekehrten Sinn vom Thing zur Plattform erfolgt. Weiters bietet die Plattform eine einheitliche Schnittstelle für jegliche Anwendungen, die auf den Daten vieler Things basiert. Darüber hinaus bieten diese Plattformen vielfältigere Möglichkeiten hinsichtlich Datenpersistierung, zeitbasierten Abfragen und Integration zwischen Anwendungen und Things.

2.3 Zusammenfassung

Die zunehmende Bedeutung der Integration von digitalen Artefakten mit der physischen Welt hat dazu geführt, dass Menschen und Applikationen immer öfter mit „smarten“ Dingen interagieren. Im Rahmen des *Internet der Dinge* wird folglich nach Möglichkeiten gesucht, um diese smarten Dinge miteinander zu vernetzen [29]. Das Web-of-Things nutzt das World Wide Web und den REST-Architekturstil um eine Integration und Kommunikation von smarten Dingen in standardisierter Art und Weise zu ermöglichen. Neben der Einführung und Beschreibung von Internet-of-Things und Web-of-Things wurde in diesem Kapitel auch eine vierschichtige Applikationsintegrationsinfrastruktur beschrieben. Darüber hinaus wurde der Unterschied zwischen der Betrachtung von Smart Things als drahtlose Sensorknoten und der Integration dieser in Web-of-Things-Plattformen, die als zentraler Hub dienen, vorgestellt.

Kapitel 3

Smart Vending

Die physische Beschaffenheit vieler Dinge ist oft sehr komplex. Folglich sollten Web-of-Things-Plattformen auch in der Lage sein, solche komplexen Dinge abbilden zu können. Da der Verkauf von Waren und Dienstleistungen mittels Automaten immer populärer wird, wird im Rahmen dieser Arbeit beispielhaft die Integration von Verkaufsautomaten ins Web-of-Things diskutiert.

Der in Abbildung 3.1 gezeigte Getränkeautomat ist dabei ein ideales Beispiel für ein komplexes Thing. Aus der Produktbeschreibung [55] geht hervor, dass der Automat aus 6 Ebenen mit je 8 Verkaufsschächten besteht, also insgesamt 48 verschiedene Produkte anbieten kann.

Um ein besseres Verständnis vom Vending-Geschäft aufbauen zu können, haben die Autoren der vorliegenden Arbeit Gespräche mit Automatenbetreibern, sowie mit einem Hersteller von Telemetrielösungen gesucht. Dabei konnten Bedürfnisse bzw. aktuelle Problemfelder identifiziert werden.

3.1 Bedeutung

Der Bundesverband der deutschen Automatenwirtschaft definiert Vending als „Verkauf von Waren und Dienstleistungen durch Automaten“ [13]. Die Automaten werden von sogenannten „Operatoren“ aufgestellt und von diesen betrieben, befüllt, gereinigt und gewartet. Darüber hinaus existiert der Begriff des „Public Vending“, welcher den Verkauf von Waren und Dienstleistungen an öffentlichen Plätzen wie Schulen, Universitäten, Museen, Banken, Baumärkten, usw. umfasst [13].

Um die Bedeutung der Vending-Industrie zu unterstreichen, führt die European Vending Association an, dass 295 Millionen Kunden mindestens einmal pro Woche einen Verkaufsautomaten verwenden [17]. Insgesamt erzielt diese Branche einen Umsatz von 11,3 Milliarden € pro Jahr mit rund 3,77 Millionen Verkaufsautomaten in Europa [17].

Abbildung 3.1: Aktueller Kaltgetränkeautomat der Serie Robimat von Sielaff [55].

3.2 Befragung von Marktteilnehmer

Die nachfolgend angeführten Unternehmen sowie Ansprechpartner erklärten sich bereit, den Autoren Einblick in die Vending-Branche, sowie in aktuelle Probleme und Herausforderungen zu gewähren.

3.2.1 Rudolf Wagner KG

Weinstraße 31
A-4664 Oberweis
Ansprechpartner: GF Ruldolf Wagner

Die von den Autoren befragte Rudolf Wagner KG ist mit etwa 4000 eingesetzten Automaten im Heiß- sowie im Kaltgetränkebereich eine der größten Automatenbetreiber Österreichs. Darüber hinaus ist das Unternehmen auch als selbstständiger Kantinen- und Buffetbetreiber tätig. Die Firma Rudolf Wagner KG bietet seinen Kunden drei verschiedene Aufstellvarianten für Automaten an, die nachfolgend erläutert werden und stellvertretend als Ge-

schäftsmodelle für das Vending Geschäft stehen.

- *Operating:* Im Falle von Operating wird der Automat leihweise von der Firma Wagner zur Verfügung gestellt und auch ordnungsgemäß gewartet und regelmäßig befüllt. Die Automaten bleiben dabei im Eigentum des Betreibers. Die Abrechnung wird ebenfalls vom Automatenbetreiber übernommen. Die etwaige Meldung von Störungen obliegt dem Kunden.
- *Miete:* Beim Mietmodell wird gegen eine monatliche Miet- und Servicekostenpauschale der gewünschte Automatentyp zur Verfügung gestellt und vom Betreiber gewartet. Der Unterschied zum Operating besteht darin, dass die Betreuung, Reinigung und Abrechnung durch den Kunden zu erfolgen hat. Die Füllprodukte dürfen dabei meistens nur beim Automatenbetreiber bezogen werden.
- *Kauf:* Wird der Kauf eines Automaten bevorzugt, so hat der Kunde sämtliche Betreuungs-, Reinigungs-, Befüllungs-, und Abrechnungstätigkeiten selbst zu erledigen. Bei Kauf des Automaten besteht keine Bindung bzgl. der Füllprodukte vom Operator.

Herr Wagner sieht im Vending noch großes Potenzial, aber auch Herausforderungen, die es zu lösen gilt. Die meisten Kunden der Rudolf Wagner KG sind Firmenkunden, die ihren Mitarbeitern Heiß- und Kaltgetränkeautomaten zur Verfügung stellen wollen. Laut Wagner gibt es viele proprietäre Telemetrielösungen, die aber ein unzureichendes Kosten-Nutzen-Verhältnis aufweisen und sich nicht in bestehende Warenwirtschaftssysteme bzw. vorhandene Enterprise Resource Planning (ERP)-Lösungen integrieren lassen. Weiters liefern die Telemetrielösungen noch keine Echtzeitinformationen. Public Vending ist für Wagner nicht relevant bzw. noch sehr unterentwickelt.
Darüber hinaus nennt Wagner folgende Herausforderungen:

- *Einbruch und Vandalismus*
- *Miteinbeziehung des Endkonsumenten:* Durch neue smarte Telemetrie bzw. Vending-Lösungen könnten die Endkonsumenten besser integriert werden, indem diese Füllstände bzw. Standorte von Automaten am Smartphone ermitteln können.
- *Bezahlung mittels NFC:* Derzeit sind die Betriebskosten im Zusammenhang mit NFC noch zu hoch für den flächendeckenden Einsatz. Aufgrund der Tatsache, dass sich die Bezahlung mit Quick nicht durchgesetzt hat (Akzeptanz von rund 2 Prozent), lässt NFC als Hoffnungsträger der bargeldlosen Bezahlung aufleben.

3.2.2 Brunnhofer Verpflegungsautomaten

Bahnhofstraße 9
A-8792 St. Peter-Freienstein

Ansprechpartner: Ing. Manfred Brunnhofer

Das Unternehmen Brunnhofer Verpflegungsautomaten zählt mit 3 Mitarbeitern zur Kategorie der kleinen Automatenbetreiber. Der Automatenbetreiber setzt für Eierautomaten, die sich weit vom Standort entfernt befinden, die Lösung Vendon[1] ein. Vendon ist eine Telemetrielösung bestehend aus Hard- und Software. Die Lösung ist laut Brunnhofer noch sehr teuer (190€/Automat in der Anschaffung und 9€/Monat für die Nutzung der Plattform) und daher noch nicht für den flächendeckenden Einsatz tauglich.

Allerdings sind gerade die kleinen Automatenbetreiber auf Telemetrielösungen angewiesen, da die Automaten weniger oft zyklisch angefahren werden. Große Betreiber sind bis zu zwei mal wöchentlich zur Befüllung und Reinigung beim Automaten vor Ort und daher nicht auf eine solche Lösung angewiesen. Die Automatenhersteller Dallmayr und café+co haben eigene Telemetrielösungen entwickelt, welche jedoch wiederum proprietär sind und nicht kompatibel mit anderen Herstellern sind. Automatenbetreiber setzen jedoch eine Vielzahl an Automaten unterschiedlicher Hersteller ein, was folglich zum Einsatz mehrerer heterogener Systeme und Plattformen führt.

Zuletzt ist es nicht möglich, über die Softwaresysteme schreibend auf die Automaten zuzugreifen. Eine dynamische Preisauszeichnung ist somit nicht möglich. Die Erwartungshaltung von Brunnhofer an das zukünftige Vending ist ebenfalls groß: Zukünftig soll es möglich sein, bezahlte Werbung während der Zubereitung schalten zu können, sowie die Bezahlung per Near Field Communication durchzuführen.

3.2.3 Vendidata Software Entwicklung GmbH

Swarovskistraße 15
A-6130 Schwaz
Ansprechpartner: GF Manfred Steiner

Vendidata entwickelt Verwaltungssoftware für Automaten jeglicher Art. Die entwickelte Lösung zielt dabei darauf ab, den Automaten mit einer speziellen Komponente auszustatten, welche in Verbindung mit einem Handheld via Infrarot drahtlos Information austauscht. Wird ein Automat von einem Mitarbeiter angefahren, können mittels Handheld sämtliche Daten (Anzahl und Art der gefüllten Produkte, eingesammelte Erlöse, usw.) vor Ort ausgelesen werden. Die Daten werden dann mittels Bluetooth auf das Mobiltelefon des Mitarbeiters übertragen und via GSM zu der Backoffice-Software übertragen. Mittels dieser Lösung wird die lückenlose Aufzeichnung des Tagesgeschäftes ermöglicht.

Laut Steiner ist das Hauptproblem im Vending Geschäft die Tatsache,

[1] http://www.vendon.net/

dass die Automatenhersteller die verabschiedeten Standards nicht einhalten. Deshalb beinhaltet der Handheld von Vendidata über 1000 Korrekturdateien, um die Daten verschiedenster Automaten korrekt auslesen und standardisiert übertragen zu können. Natürlich versuchen die Hersteller eigene proprietäre Lösungen anzupreisen und zur Erhöhung der Kundenbindung Telemetrie & Automat aus einer Hand anzubieten. Telemetrie sollte immer herstellerunabhängig und flächendeckend erfolgen. Steiner ist ebenfalls der Meinung, dass der Einsatz von Remote-Telemetrie für bestimmte Automatentypen wie Snack-, Eier-, Heiß- und Kaltegetränkeautomaten viele Vorteile bietet. Ausgenommen davon sind jedoch Kaffeeautomaten, aufgrund der Tatsache, dass zweimal wöchentlich eine Reinigung mit gleichzeitiger Befüllung stattfindet.

3.2.4 Sielaff Austria GmbH

Kaiser-Max-Straße 51
6060 Hall in Tirol
Ansprechpartner: GF Olf Thiele

Sielaff Austria ist ein Tochterunternehmen der deutschen Sielaff GmbH & Co. KG mit Sitz in Herrieden, Deutschland. Laut Geschäftsführer Thiele haben sich in Österreich Remote-Telemetrielösungen aktuell nicht durchgesetzt, da der Preis solcher Lösungen die zu erzielenden Kosteneinsparungen nicht rechtfertigt. Gerade bei Kaffeeautomaten ergibt sich die Situation, dass der Automat alle paar Tage angefahren werden muss, da eine Reinigung und eine Bohnenauffüllung notwendig ist.

Die Tatsache, dass die Automatenhersteller standardisierte Protokolle unterschiedlich implementieren, erschwert die Entwicklung einer einheitlichen Telemetrielösung. Ein Operator betreut jedoch meistens mehrere verschiedene Automaten von unterschiedlichen Herstellern. Der Einsatz von nur einem Remote-Telemetriesystem ist somit fast nicht möglich.

3.3 Ergebnisse der Befragung

Durch die Befragung der Marktteilnehmer konnten die Autoren wertvolle Informationen über die derzeitige Situation im Vending gewinnen. Diese Erkenntnisse werden nachfolgend zusammengefasst festgehalten.

- **Proprietäre Telemetrielösungen**
  Die Tatsache, dass Hersteller nach wie vor verabschiedete Standardprotokolle wie Multi Drop Bus[2] oder EVA-DTS[3] unterschiedlich implementieren, erschwert eine zentrale Plattform zur Verwaltung von

[2] http://www.vending-europe.eu/en/standards_protocols/mdb-icp.html
[3] http://www.vending-europe.eu/en/standards_protocols/eva-dts.html

allen Automaten eines Operators. Die Hersteller drängen daher eher auf eigene proprietäre Plattformen zur umfassenden Verwaltung. Die Betreiber müssen dadurch je nach Umfang der Automatenflotte eine Vielzahl an Systemen einsetzen.

- **Fehlende Echtzeitinformationen**
  Derzeit verfügbare Telemetriesysteme sind nicht in der Lage, Informationen wie bspw. Füllstand oder Fehlercodes in Echtzeit an die Betreiber zu übermitteln. Gerade aber im Vending ist die Verfügbarkeit von Informationen über den aktuellen Zustand erfolgskritisch, denn Stillstand resultiert im fehlenden Umsatz.
- **Unausgewogenes Kosten-Nutzen-Verhältnis**
  Auch die Kosten von verfügbaren Telemetriesystemen rechtfertigen aufgrund fehlender Features die Kosten von Hard- und Software noch nicht.
- **Unzureichende Integrationsmöglichkeiten**
  Weiters beklagen Automatenbetreiber die fehlende Integration solcher Systeme in eingesetzte Warenwirtschaftssysteme, welche Lagerstände kontrollieren bzw. Bestellanforderungen verschicken. Auch eine Einbindung des Endkonsumenten ist derzeit nicht vorhanden.
- **Regelbasierte Aktionen**
  Zuletzt wäre es auch wünschenswert aufgrund gesammelter Statusinformationen Aktionen am Automaten einzuleiten. Beispielsweise wäre es wünschenswert, bei Unterschreiten eines gewissen Füllstands den nächst gelegenen Refiller zu benachrichtigen/beauftragen. Diese regelbasierten Automatismen sind derzeit ebenfalls nicht verfügbar.

3.4 Smart Vending Szenario

Auf Basis der von den Marktteilnehmern kommunizierten Unzulänglichkeiten haben die Autoren ein simples Smart Vending Szenario entworfen, die den Einsatz von Telemetrielösungen attraktiv machen würde. Nachfolgend werden die wichtigsten Anwendungsfälle skizziert:

- **Füllstandsbenachrichtigung**
  Durch die Vernetzung eines Automaten sollen aktuelle Füllstände automatisch an den Betreiber übermittelt werden. Weiters soll im Falle der Erreichung eines kritischen Füllstandes automatisch eine Nachlieferung bei einem Refiller angeordnet werden.
- **Analyse des Konsumverhalten**
  Durch die Nutzung des Automaten entstehen für den Betreiber wertvolle Informationen über das Konsumverhalten. Diese Informationen sollen ausgewertet werden können, um Trends im Konsumverhalten feststellen sowie Strategieanpassungen vornehmen zu können.

- **Suche nach bestimmten Automaten**
  Kunden sollen vorab die Verfügbarkeit von bestimmten Produkten im Automaten ermitteln können. Sollte ein bevorzugtes Produkt nicht mehr erhältlich sein, soll der Kunde zum nächst gelegenen Automaten verwiesen werden.
- **Einfaches Anlegen von Automaten unter Wiederverwendung**
  Zur einfacheren Verwaltung der gesamten Automatenflotte sollen die einzelnen Automatentypen (z. B. Kaltegetränkeautomat Sielaff Robimat 75 mit allen Schächten bzw. Sensoren) als Templates global definiert werden können. Die Neuanlage eines konkreten Automats kann dann auf Basis dieser definierten Vorlagen erfolgen. Die Vermeidung dieser repetitiven Tätigkeiten führt zur effizienteren Anlage von Automaten.
- **Geschäftsregeln**
  Zuletzt soll die Plattform die Definition von einfachen Geschäftsregeln auf Basis von Event-Condition-Action erlauben. Damit könnte der Automatenbetreiber beispielsweise bei Unterschreitung der Boilertemperatur eines Kaffeevollautomaten benachrichtigt werden. Auch das Setzen von automatischen Maßnahmen mittels Aktuatoren soll ermöglicht werden, sodass z. B. Preise auf Basis der Außentemperatur gesetzt werden können.

3.5 Zusammenfassung

Die zunehmende Bedeutung von Vending und Web-of-Things veranlasste die Autoren ein Smart Vending Szenario zu entwickeln, auf Basis dessen analysiert werden soll, ob und wie eine Integration von Automaten in das Web-of-Things erfolgen kann. Hierfür nahmen die Autoren Kontakt mit Betreibern und Herstellern auf, um Bedürfnisse und Probleme bisheriger Lösungen feststellen zu können. Auf Basis der durchgeführten Interviews konnten die Autoren feststellen, dass derzeit noch eine Vielzahl an Problemen existiert, welche den flächendeckenden Einsatz von Remotetelemetrielösungen verhindern. Einerseits bereitet die Heterogenität der Automaten durch die unterschiedliche Implementierung von Protokollen große Probleme und andererseits rechtfertigt der Nutzen nicht die Kosten von derzeit verfügbaren Telemetrielösungen. Daher soll im nachfolgenden Kapitel untersucht werden, ob aktuell verfügbare Web-of-Things Plattformen verwendet werden können, um Automaten ins Web integrieren zu können.

Das in diesem Kapitel entwickelte Szenario wird in Folge benötigt, um eine Vorteilhaftigkeit für Automatenbetreiber sicherstellen zu können. In Kapitel 4 wird nun anhand der entwickelten Anwendungsfälle untersucht, ob aktuelle Web-of-Things Plattformen dazu verwendet werden können, die genannten Anwendungsfälle abdecken zu können.

Kapitel 4

Evaluierung bestehender Plattformen

In diesem Kapitel werden nun aktuelle Web-of-Things Plattformen hinsichtlich ihrer Eignung in Bezug auf das entwickelte Smart Vending Szenario aus Abschnitt 3.4 evaluiert. Um eine Evaluierung durchführen zu können, wurden zuerst Evaluierungskriterien aus dem entworfenen Szenario abgeleitet. Anschließend wurden vier WoT-Plattformen ausgewählt und untersucht. Aufgrund der Vielzahl an existierenden WoT-Plattformen wurden vier Vertreter ausgewählt und evaluiert. Diese Evaluierung wurde bereits im Rahmen einer Seminararbeit am Institut für Data & Knowledge Engineering im Juli 2014 durchgeführt. Jedoch wurde zu dieser Zeit, aufgrund fehlendem Kenntnisstand im Vending-Umfeld, keine Evaluierung anhand komplexer Things, Business Rules, Templates und Erweiterbarkeit durchgeführt. Aus diesem Grund wurde die Evaluierung mit den unten stehenden Kriterien auszugsweise neu durchgeführt.

4.1 Kriterien

In Folge werden die für die Evaluierung benötigten Kriterien erläutert. Diese wurden von dem in Abschnitt 3.4 entwickelten Szenario abgeleitet.

4.1.1 Abbildung komplexer Things

Das Kriterium „Abbildung komplexer Things“ soll feststellen, ob es möglich ist ein komplexes Thing strukturiert und realitätsnah abbilden zu können. Struktuiert meint dabei die Fähigkeit Sensoren und Aktuatoren so zusammenzufassen bzw. zu gruppieren, damit diese der Realität bestmöglich entsprechen. Aktuatoren stellen dabei Zugriffspunkte auf das Thing dar, die gewisse Aktionen bzw. Veränderungen hervorrufen. Als Beispiel kann ein Getränkeautomat angeführt werden, welcher aus verschiedensten Schächten

mit untergeordneten Sensoren besteht.

4.1.2 Business Rules

Unter Business Rules meinen die Autoren eine Unterstützung von regelbasierten Automatismen. Konkret soll untersucht werden, inwiefern Regeln auf Basis von Sensorwerten definiert bzw. ausgeführt werden können.

4.1.3 Erweiterbarkeit

Das Kriterium Erweiterbarkeit soll die Option einer Plattformerweiterung, in welcher Form auch immer, untersuchen. Der Zugriff auf eine angebotene Web API stellt dabei keine Erweiterbarkeit in unserem Sinne dar.

4.1.4 Templates

Um den Operator wesentlich zu unterstützen, wird untersucht ob eine Plattform fähig ist Templates anzulegen bzw. zu verwalten. Eine Plattform erfüllt dieses Kriterium, wenn es die Anlage von Things auf übergeordneter Ebene ermöglicht, die bei einer Neuanlage zu einem späteren Zeitpunkt wiederverwendet werden können. Selbst die Verfügbarkeit eines einfachen Copy & Paste-Mechanismus auf Ebene eines gesamten Thing erfüllt dieses Kriterium.

4.1.5 Unterstützte Datenformate

Bei den unterstützten Formaten für Daten soll geklärt, welches Datenformat für Requests und Responses in HTTP verwendet werden kann. Zur Auswahl stehen dabei XML, JSON und Efficient XML Interchange (EXI). EXI[1] ist dabei eine effizientere und binäre Repräsentation von XML, welche weniger Bandbreite beim Transfer in Anspruch nimmt.

4.1.6 Visualisierung

Unter diesem Punkt fallen mögliche Darstellungsformen der erfassten Sensorwerte. Es wird untersucht, ob die Werte anhand von Listen und/oder in Form von Diagrammen dargestellt werden können. Weiters soll untersucht werden, ob Standorte von Things in Kartenform dargestellt werden können, um sofort den Aufenthaltsort eines Things ermitteln zu können.

4.1.7 Web Services

Unter diesem Punkt soll geklärt werden, ob die zu untersuchende Plattform entweder klassische Webservices via SOAP oder schlankere REST-basierte

[1] http://www.w3.org/TR/exi

Webservices unterstützt.

4.2 Untersuchte Plattformen

Insgesamt wurden vier WoT-Plattformen zur Evaluierung herangezogen. Aufgrund der Vielzahl an existierenden WoT-Plattformen wurden zwei Vertreter (Evrythng und WoTKit) aufgrund des Bekanntheitsgrades und zwei Plattformen (ThingSpeak und Paraimpu) willkürlich ausgewählt.

4.2.1 Evrythng

EVRYTHNG[2] ist eine von der Evrythng Ltd. entwickelte Plattform zur Integration jeglicher Konsumprodukte in das Web. Einer der Gründer ist unter anderem der Web-of-Things-Begründer Dominique Guinard. Im Zentrum steht die sogenannte „Smart Product Active Digital Identity“, die jedem Ding oder Gegenstand eine eindeutige Identität zuweist, um damit interagieren zu können. Die cloudbasierte Plattform unterstützt die Bereitstellung von Informationen in Echtzeit sowie Erweiterbarkeit über sogenannte „Applications“. Die Plattform bietet darüber hinaus viele Features sowie eine mächtige REST-Schnittstelle, Client Libraries und Wrapper-Bibliotheken.

4.2.2 WoTKit

WoTKit[3] ist ein Web-of-Things Framework, welches von Michael Blackstock und Rodger Lea am Media and Graphics Interdisciplinare Center an der University of Columbia entwickelt und nun durch ihre Firma, Sense Tecnic System, verbreitet wird. Der Fokus von WoTKit liegt auf der Bereitstellung eines leichtgewichtigen Toolkits für die Entwicklung von IoT-Applikationen mittels Web Technologien und Rapid Application Development. Angelegt als Platform-as-a-Service (PaaS), dient das System als Aggregator für Sensordaten, als Dashboard, zur entfernten Steuerung und um Daten aufzubereiten. Darüber hinaus kann die integrierte RESTful API auch für die Entwicklung von eigenen Applikationen verwendet werden.

4.2.3 ThingSpeak

ThingSpeak[4] versteht sich als eine Open Data Plattform zur einfachen Integration von Dingen ins Internet. Die Plattform wurde von ioBridge Inc. entwickelt. Um mit ThingSpeak kommunizieren zu könnnen, muss zuerst ein sogenannter „Channel“ angelegt werden. Über diesen „Channel“ kann dann

[2] https://evrythng.com
[3] http://wotkit.sensetecnic.com/wotkit
[4] https://thingspeak.com

ein Device, eine App oder ein Thing Daten an ThingSpeak senden. Während es möglich ist mittels Plugins Erweiterungen innerhalb der Plattform im Sinne einer Anwendung zu entwickeln, gibt es auch bereits einige vorinstallierte Anwendungen wie ThingTweet, welche den aktuellen Sensorwert und Status eines Things auf Twitter postet.

4.2.4 Paraimpu

Paraimpu[5] ist ein soziales Tool zur Integration von Dingen und zur Erstellung von personalisierten WoT-Anwendungen. Es erlaubt physische oder virtuelle Dinge mit der Plattform zu verbinden, verwenden und zu teilen. Der Name Paraimpu ist aus dem sardinischen Wort Paralimpu abgeleitet. Ein Paralimpu bezeichnet eine Person, welche als Vermittler eine Hochzeit zwischen zwei Personen arrangiert. Im Kontext von Web-of-Things meint Paraimpu einen Vermittler, um unterschiedliche „Entitäten“ zu verbinden. Im Gegensatz zu den bisher vorgestellten Plattformen steht bei Paraimpu, neben der einfachen Einbindung von Dingen, die Möglichkeit zum Teilen von Dingen auf sozialen Plattformen wie Facebook oder Twitter im Vordergrund.

4.3 Ergebnis

Die Tabelle 4.1 fasst das Ergebnis der Evaluierung in Form eines Vergleichs, anhand der von den Autoren identifizierten Kriterien, zusammen. Nachfolgend werden die Ergebnisse anhand der einzelnen Kriterien detaillierter erläutert.

4.3.1 Abbildung komplexer Things

Als eines der wichtigsten Kriterien für die Evaluierung galt die Abbildung komplexer Things zur realitätsnahen Darstellung von Automaten. Die Autoren der vorliegenden Arbeit mussten feststellen, dass keine der untersuchten Plattformen eine Möglichkeit zur Strukturierung von Sensoren und Aktuatoren anbietet. Trotzdem unterstützen Paraimpu und WoTKit die Definition von Aktuatoren. WoTKit erlaubt mit der Actuator Control API angebundene Things mittels URL Callbacks, LongPolling oder WebSockets anzusteuern. Als soziale Web-of-Things-Plattform versteht Paraimpu darüber hinaus unter Aktuatoren auch Ausgabekanäle zur Weitergabe von Informationen an Twitter, Facebook, usw.

4.3.2 Business Rules

WoTKit und Paraimpu erlauben die Definition von einfachen Geschäftsregeln auf Basis von Event-Condition-Action Regeln. Paraimpu erlaubt die

[5] https://www.paraimpu.com

	Evrythng	*ThingSpeak*	*Paraimpu*	*WoTKit*
Komplexe Things	Nein	Nein	Nein	Nein
Business Rules	Nein	Nein	Ja	Ja
Erweiterbarkeit	Applications	Plugins	Nein	Processor
Templates	Nein	Nein	Nein	Nein
Datenformate	JSON	JSON	JSON, CSV, XML	JSON
Visualisierung	Ja	Charts	Ja	Ja
Web Services	REST	REST	REST	REST

Tabelle 4.1: Ergebnisse der Evaluierung in tabellarischer Form.

Definiton einer Geschäftsregel unter Angabe des Sensors, Aktuators sowie der Verbindung. Innerhalb dieser Verbindung können mittels Filterbedingungen einfache Regeln definiert werden. WoTKit hingegen ist mit einem ereignisgesteuerten Verarbeitungssystem ausgestattet, sodass mittels Pipes und Modules Sensordaten verarbeitet, gefiltert und weitere Aktionen ausgeführt werden können. Evrythng und ThingSpeak erlauben keine regelbasierte Verarbeitung.

4.3.3 Erweiterbarkeit

Geht es um die Erweiterbarkeit, so bieten drei von vier Plattformen die Möglichkeit eigene Bedürfnisse abzubilden. Evrythng kann insofern erweitert werden als Applications darin erstellt werden können. Mit diesem Feature ist es möglich externen nativen oder webbasierten Anwendungen Zugriff auf die Produkte und Things zu ermöglichen. ThingSpeak erlaubt ebenfalls die Entwicklung eigener Anwendungen mit HTML, CSS und JavaScript. Bei WoTKit ist die Erweiterung mithilfe von Apps, Pipes und Connectors möglich. Paraimpu erlaubt momentan keine Erweiterungen.

4.3.4 Templates

Betrachtet man hingegen die Unterstützung von Templates, so muss man feststellen, dass keine der Plattformen auch nur ansatzweise eine Unterstützung bietet. Abgesehen von Templates ist auch eine einfache Wiederverwendung eines Things mit Copy & Paste bei allen Plattformen nicht möglich.

4.3.5 Unterstützte Datenformate

Als Datenformat für die Übertragung mittels HTTP erlauben alle Plattformen die Anwendung von JSON. Vorreiter ist dabei Paraimpu mit einer zusätzlichen Unterstützung für Comma-separated Values (CSV) und XML.

4.3.6 Visualisierung

Positiv hervorzuheben ist die Tatsache, dass beinahe alle Plattformen Sensorwerte sowohl in Listenform als auch Diagrammform darstellen. Einzig und alleine ThingSpeak ermöglicht nur eine Darstellung in Diagrammform.

4.3.7 Web Services

Bei Betrachtung der angebotenen Web Services oder Web APIs lässt sich eine Gemeinsamkeit feststellen: Alle Plattformen setzen auf die leichtgewichtige Implementierung von REST-fähigen HTTP-Services. SOAP Services sind dabei für alle Plattformen keine Alternative. Die Plattform Evrythng bietet zusätzlich Wrapper-Bibliotheken für mehrere Programmiersprachen (JavaScript, Node.js, Java und Android) zur einfacheren Interaktion mit den Services.

4.4 Schlussfolgerung

Die Autoren sind nach Durchführung der Evaluierung der Auffassung, dass das entwickelte Szenario aus Abschnitt 3.4 durch aktuelle WoT-Plattformen nicht ausreichend unterstützt wird. Den Autoren zufolge wurden die bestehenden WoT-Plattformen intentional nicht für komplexe Things entworfen. Aufgrund der festgestellten Unzulänglichkeiten, vor allem der fehlenden Unterstützung zur Abbildung komplexer Things, entschlossen sich die Autoren zur Entwicklung einer eigenen generischen Plattform in Form eines Prototypen, welche das vorgeschlagene Szenario adäquat unterstützen soll.

In Abschnitt 5.2 werden hierfür von den Autoren Ziele und Anforderungen definiert und umgesetzt. Die entwickelte Plattform erhebt dabei nicht den Anspruch auf Vollständigkeit. Die zu entwickelnde Plattform soll lediglich als prototypische Implementierung die mögliche Umsetzung einer solchen Smart Vending Plattform aufzeigen.

4.5 Zusammenfassung

Um eine Aussage über die Unterstützung des Smart Vending Szenarios in aktuellen WoT-Plattformen treffen zu können, definierten die Autoren zunächst Evaluierungskriterien, die für eine akzeptable Smart Vending Lösung erforderlich sind. Anhand dieser Kriterien wurden dann vier Plattformen

(Evrythng, ThingSpeak, Paraimpu und WoTKit) ausgewählt und hinsichtlich Kriterienerfüllung analysiert. Die Ergebnisse der Evaluation ließen die Autoren zu dem Punkt kommen, dass heute verfügbare WoT-Plattformen für das entwickelte Smart Vending Szenario nicht geeignet sind. Deshalb entschieden sich die Autoren eine eigene prototypische Plattform zu entwerfen, die das Smart Vending Szenario ausreichend abbildet. Diese Eigenentwicklung mit den von den Autoren definierten Zielen und Anforderungen ist Gegenstand des nachfolgenden Kapitel 5.

Kapitel 5

Überblick über die Web-of-Things-Plattform WoTCloud

Die im vorigen Kapitel durchgeführte Evaluation bestehender WoT-Plattformen ließ die Autoren zum Schluss kommen, dass diese Plattformen ungeeignet sind für das entwickelte Smart Vending Szenario. Die Autoren entschieden sich für die Entwicklung einer eigenen, prototypischen Plattform, um die genannten Unzulänglichkeiten zu berücksichtigen. Dieses Kapitel ist der Beschreibung der Ziele, Anforderungen sowie der technischen Umsetzung gewidmet.

5.1 Ziele und Nicht-Ziele

Auf Basis des in Kapitel 3 entwickelten Smart Vending Szenarios und der in Kapitel 4 durchgeführten Evaluierung definierten die Autoren übergeordnete Ziele für den zu entwickelnden Prototyp, namens "WoTCloud":

- Prototypische Implementierung einer generischen Web-of-Things Plattform zur Überwachung und Steuerung von Automaten
- Implementierung als Software-as-a-Service Lösung auf Basis von Microsoft Azure
- Anwendung von aktuellen Webtechnologien
- Abbildung des Smart Vending Szenarios aus Abschnitt 3.4 mit den unten genannten Einschränkungen
- Unterstützung der Aspekte: Abbildung von komplexen Things, Templates, Business Rules und Multi-Tenancy

Unter generisch verstehen die Autoren dabei die Möglichkeit unterschiedlichste Automaten abbilden bzw. einbinden zu können. Auf Basis der oben genannten Ziele werden nun in Abschnitt 5.2 Anforderungen abgeleitet bzw.

definiert.

Im Sinne einer Systemabgrenzung werden nachfolgend auch explizit die Nicht-Ziele des zu entwickelnden Prototyps aufgelistet.

- Umsetzung einer voll funktionsfähigen, vollständigen und marktreifen WoT-Plattform
- Einbindung des Endkonsumenten (beispielsweise mittels Smartphone App)
- Setzen von Security-Maßnahmen
- Garantierte Performance
- Physische Anbindung von Automaten an die Plattform inkl. Auseinandersetzung mit Automatenhardware bzw. -protokollen

5.2 Anforderungen

Die nun folgenden Anforderungen wurden von den Autoren zur Umsetzung des Smart Vending Szenarios definiert und hierfür natürlichsprachig dokumentiert. Die Konstruktion einer Anforderung erfolgte dabei auf Basis der von Klaus Pohl [48] vorgestellten Satzschablone.

Eine Satzschablone ist "ein Bauplan für die syntaktische Struktur einer einzelnen Anforderung" [48]. Die Anforderungsdokumentation mittels Satzschablonen ist einfach anzuwenden und liefert eine hohe Qualität in Bezug auf Vollständigkeit. Die einhergehenden Nachteile der Dokumentation mittels natürlicher Sprache können im Falle der vorliegenden Masterarbeit vernachlässigt werden, da die Plattform ohnehin nur von den beiden Autoren entwickelt wurde.

A01 - Registrierung als Tenant Die Plattform muss einem nicht registrierten Operator die Möglichkeit bieten sich am System zu registrieren und dabei folgende Daten aufnehmen:

- Firmenname
- Benutzername
- Passwort

Der angelegte Benutzer ist aus Gründen der Einfachheit der einzige Benutzer eines Automatenbetreibers.

A02 - Login als Tenant Die Plattform muss einem registrierten Operator die Möglichkeit bieten sich im System einzuloggen. Zur Authentifizierung sollen Benutzername und Passwort verwendet werden.

A03 - Thing Übersicht Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten im System existierende Things übersichtlich anzeigen zu können.

A04 - Thing anlegen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten im System Things anlegen zu können. Bei der Anlage von Dingen müssen folgende Felder eingegeben werden können:

- Name
- Beschreibung
- Längengrad des Standorts
- Breitengrad des Standorts

Zusätzlich muss es bei der Anlage möglich sein, komplexe Things, Sensoren (Anforderung A08) und zugehörige Aktuatoren (Anforderung A15) zu spezifizieren. Unter komplexen Things versteht man jene Things, die selbst wiederum aus Things bestehen können. Die Anlage muss dabei auf Basis eines Templates oder als Neuanlage erfolgen können.

A05 - Thing editieren Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten im System existierende Things editieren zu können. Es können alle Felder, die bei der Anlage spezifiziert wurden, geändert werden.

A06 - Thing löschen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten zugehörige Things löschen zu können.

A07 - Thing anzeigen Die Plattform muss einem angemeldeten Operator die Möglichkeiten bieten bei Auswahl eines Things die dazugehörigen Detailinformationen abrufen zu können. Hierbei soll die Beschreibung, eine Map mit der geografischen Position, sowie alle zugehörigen Sensoren angezeigt werden.

A08 - Sensor anlegen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten zu einem (komplexen) Thing einen Sensor hinzuzufügen. Hierfür werden folgende Felder benötigt:

- Name
- Datentyp (bool, int, decimal oder String)

A09 - Sensor löschen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten einen Sensor löschen zu können.

A10 - Sensorwerte anzeigen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten die letzten 30 Sensorwerte in Form einer Liste und als Liniendiagramm aufbereiten bzw. darstellen zu können.

A11 - Sensorwert erzeugen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten einen Sensorwert hinzuzufügen.

A12 - Template anlegen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten private sowie öffentliche Vorlagen erstellen zu können. Die benötigten Felder bzw. Funktionen ergeben sich analog aus Anforderung A04.

Bedeutung: Templates sollen dazu verwendet werden komplexe Dinge wie Automaten mit unterschiedlichen Schächten und Sensoren (z.B. Sielaff Robimat XL) vordefinieren zu können, um diese anschließend mehrfach wiederverwenden zu können. Wird eine Vorlage als öffentlich definiert, können andere Operatoren diese Vorlage ebenfalls wiederverwenden.

A13 - Templateübersicht Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten private sowie öffentliche Vorlagen übersichtlich anzeigen zu können.

A14 - Template löschen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten private sowie öffentliche Vorlagen löschen zu können.

Hinweis: Aus Gründen der Einfachheit kann ein Operator seine eigenen (privaten) als auch öffentliche Vorlagen jederzeit löschen.

A15 - Aktuator anlegen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten zu einem (komplexen) Thing einen Aktuator hinzuzufügen. Hierfür werden folgende Felder benötigt:

- Name
- URI

Bedeutung: Mittels Aktuatoren kann schreibend auf ein Thing zugegriffen werden. Aktuatoren werden unter anderem für die Anforderung A18 benötigt.

A16 - Aktuator ausführen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten zu einem (komplexen) Thing zugehörigen Aktuator unter Angabe eines Wertes auszuführen. Der spezifizierte Wert wird dabei nicht von der Plattform validiert, da es sich sowohl um elementare Typen (string, int, bool, usw.) aber auch komplexe Datenstrukturen (XML, JSON, usw.) handeln kann.

A17 - Aktuator löschen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten einen Aktuator löschen zu können.

A18 - Geschäftsregel anlegen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten zu einem (komplexen) Thing eine Geschäftsregel hinzuzufügen. Hierfür werden folgende Felder benötigt:

- Angabe ob Regel datengetrieben oder eventgetrieben
- Zeitangabe (hh:mm) bei eventgetriebener Regel
- Auswahl eines Sensors
- Operator (kleiner, kleiner gleich, größer, größer gleich, gleich und ungleich)
- Vergleichswert
- Angabe ob Aktion, Aktuator oder E-Mail
- Auswahl eines Aktuators bei Aktion Aktuator
- Übermittlungswert bei Aktion Aktuator
- Empfängeradresse bei Aktion E-Mail
- Textnachricht bei Aktion E-Mail

Nach erfolgreicher Erstellung ist die Regel aktiv und kann nur über eine Löschung deaktiviert werden.

A19 - Geschäftsregel löschen Die Plattform muss einem angemeldeten Operator die Möglichkeit bieten eine Geschäftsregel löschen zu können.

5.3 Konzeptuelles Modell für komplexe Dinge

Einer der Hauptaspekte der Plattform ist die Abbildung von komplexen Dingen. Ein komplexes Ding könnte beispielsweise ein Getränkeautomat (Sielaff Robimat) sein. Dieser Getränkeautomat besteht neben allgemeinen Sensoren (z. B. Außentemperatur, Lokation,...) aus bis zu 48 Schächten. Jeder dieser Schächte kann wiederum als eigenes Ding angesehen werden und besteht aus Sensoren (z. B. Füllstand, Schachttemperatur, Preis,...) und Aktuatoren. Aus dieser Darstellung ergibt sich eine Unterscheidung zwischen komplexen und simplen Dingen, wobei ein komplexes Ding mehrere simple Dinge subsumiert (siehe Abbildung 5.1). Prinzipiell könnte eine beliebige hierarchische Struktur ermöglicht werden, indem ein komplexes Ding eine Vielzahl an abstrakten Dingen als Selbstreferenz zulassen würde. Die Autoren haben aber die Hierarchie auf zwei Stufen begrenzt, da die Entwicklung eines generischen Webclients zur Erfassung und Verwaltung von Dingen ansonsten den Rahmen dieser Arbeit sprengen würde.

Auf dieser Grundlage ergibt sich das in Abbildung 5.1 dargestellte konzeptuelle Modell. Ausgehend von einem Kunden (Tenant), der im Smart

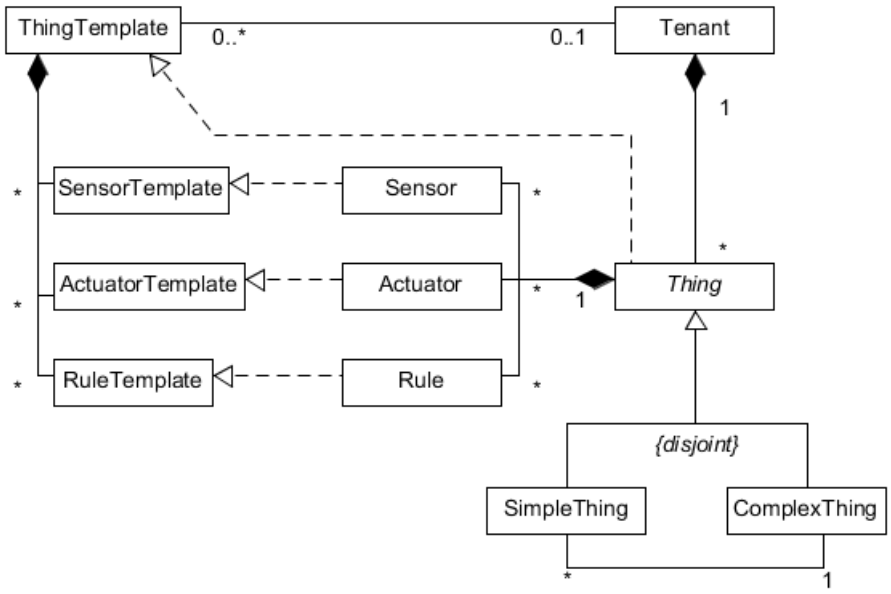

Abbildung 5.1: Konzeptionelles Modell von WoTCloud.

Vending Szenario einen Operator darstellen würde, gibt es eine abstrakte Basisklasse Thing und zwei konkrete, disjunkte Spezialisierungen: *SimpleThing* und *ComplexThing*. Ein komplexes Ding (*ComplexThing*) kann wiederum aus einer Menge von simplen Dinge bestehen. Jedes Ding ist eindeutig genau einem Tenant zugewiesen und kann nur in Kombination mit diesem Tenant existieren. Sensoren, Aktuatoren und Rules sind der abstrakten Basisklasse Thing zugeordnet. Folglich kann sowohl für simple als auch für komplexe Dinge eine Menge an Sensoren, Aktuatoren und Regeln definiert werden, die wiederum eine Existenzabhängigkeit zu den Things aufweisen. Darüber hinaus gibt es auch noch Templates, die entweder von einem spezifischen Tenant erstellt wurden oder öffentlich zugänglich sind und eine Art Schablone für Things darstellen. So können auch für Templates Sensoren, Aktuatoren und Regeln definiert werden.

5.4 Spezifische Aspekte

In Folge werden kurz die wesentlichen, besonderen Aspekte der Plattform beschrieben. Diese Konzepte werden im zweiten Teil bzw. der begleitenden Masterarbeit (siehe Abschnitt II) detailliert erläutert.

5.4.1 Business Rules

Geschäftsregeln dienen der Abbildung von Geschäftslogiken eines Unternehmen. Sie erlauben es zeitgerecht auf eintretende Ereignisse in einer vordefinierten Art und Weise zu antworten und somit eine Verbindung von passiven Datenbeständen und der dynamischen Welt zu ermöglichen. Unsere WoTCloud-Plattform erlaubt das Erstellen von einfachen, atomaren Event-Condition-Action(ECA)-Regeln, um Teile der Geschäftslogiken (z. B. die Anforderung eines Vending-Operators über niedrige Füllstandsmengen informiert zu werden) abzubilden. Die Basis für diese reaktiven Regeln bilden die Sensordaten, die an die Plattform geliefert werden. Regeln können sowohl auf Things als auch auf Templates definiert werden. Eine Regel könnte beispielsweise wie folgt aussehen:

1. *Event:* Füllstandsänderung Schacht 1
2. *Condition:* aktueller Füllstand < 2
3. *Action:* Benachrichtige Operator

5.4.2 Multi-Tenancy

Das Konzept Multi-Tenancy (dt. Mehrmandantenfähigkeit) basiert auf dem Grundgedanken mehrere Kunden, Tenants oder Benutzer auf einer einzigen Software-Instanz zu hosten, um Kosteneinsparungen und Skalierungseffeke zu erzielen. Dies führt jedoch unweigerlich zu einer Erhöhung der Komplexität, da einerseits Isolationsmechanismen erforderlich sind, aber andererseits gewisse Ressourcen allen Beteiligten konsistent zur Verfügung gestellt werden müssen (Resource Sharing). Um diesen Anforderungen gerecht zu werden, sind insbesondere auf Datenbankebene besondere Vorkehrungen zu treffen. So besteht die Möglichkeit, dass jeder Tenant eine eigene dezidierte Datenbank erhält, alle Tenants in einer Datenbank abgelegt werden oder Mischformen realisiert werden. In diesem Zusammenhang werden Anforderungen evaluiert und unter Einbezug ökonomischer Überlegungen ein Multi-Tenancy Modell für das Deployment unserer WoT-Plattform vorgeschlagen und umgesetzt.

5.4.3 REST

Representational State Transfer, kurz REST, ist ein von Thomas Roy Fielding [19] beschriebener Architekturstil zur Entwicklung verteilter Anwendungen. REST zeigt Möglichkeiten auf, wie Kommunikationsprotokolle und Datenformate genutzt werden können, um möglichst einfach leichtgewichtige und lose gekoppelte Web Services realisieren zu können [41]. Im Zentrum von REST stehen Ressourcen, die über eine URL adressiert bzw. abgefragt werden. Zum Manipulieren dieser Ressourcen bedient sich REST der standardisierten HTTP-Operationen (PUT, POST und DELETE). Wie schon in

Kapitel 2 erwähnt, können REST-basierte Web Services als technologische Grundlage von Web-of-Things angesehen werden.

5.4.4 Templates

Eine wesentliche Anforderung unserer Plattform ist die einfache Integration und Verwendung von Things. In Zusammenhang mit unserem Smart Vending Szenario wird oft eine Vielzahl an gleichartigen Things angelegt. So möchte beispielsweise ein befragter Operator (siehe Abschnitt 3.2.1) seine 4000 Automaten, die in 4 unterschiedliche Automatentypen untergliedert sind, einfach und schnell anlegen. Durch sogenannte Templates können Referenzmodelle eines bestimmten Automatentyps erstellt und wiederverwendet werden. Ein Template spezifiziert also ein abstraktes Ding mit all seinen Sensoren und Aktuatoren. Weiters sollen auch Regeln bereits auf dieser abstrakten Ebene definiert werden können. Diese Templates ermöglichen es, Dinge basierend auf Vorlagen einfach und schnell anzulegen und stellen somit ein wesentliches Unterscheidungsmerkmal von WoTCloud dar.

5.5 Umsetzung

Nachfolgend wird nun die technische Realisierung von WoTCloud diskutiert bzw. ein Einblick in die drei entwickelten Schichten gegeben.

5.5.1 Systemarchitektur

Die Architektur der entwickelten Web-of-Things-Plattform basiert auf einem Client/Server Interaktionsmodell. Dieses Interaktionsmodell ist insofern gegeben, als das Web - im Sinne der Kernstandards/-protokolle HTTP, HTML und URIs - im Wesentlichen auf einem Client/Server-Modell basiert. Dieses Interaktionsmodell wurde konkret in einer 3-Layer Architektur umgesetzt (siehe Abbildung 5.2). Der Data Layer kümmert sich um den Zugriff und die Persistierung der Daten sowie der Abstraktion der dahinterliegenden konkreten Technologien. Der Service Layer bietet sämtliche Datenrepräsentation über Webservices an, welche vom Client - der Presentation Layer - entsprechend konsumiert und aufbereitet wird.

Die physische Verteilung (Tiers) dieser drei logischen Schichten (Layers) kann unterschiedlich erfolgen. Theoretisch könnten alle drei Layer auf einer physischen Instanz deployed werden (1-Tier). Im konkreten Fall sind jedoch sowohl Data- als auch Service Layer auf Microsoft's Cloud Plattform Azure getrennt deployed und können auch einzeln horizontal und vertikal skaliert werden. Somit ergibt sich ein 3-Tier Architekturszenario.

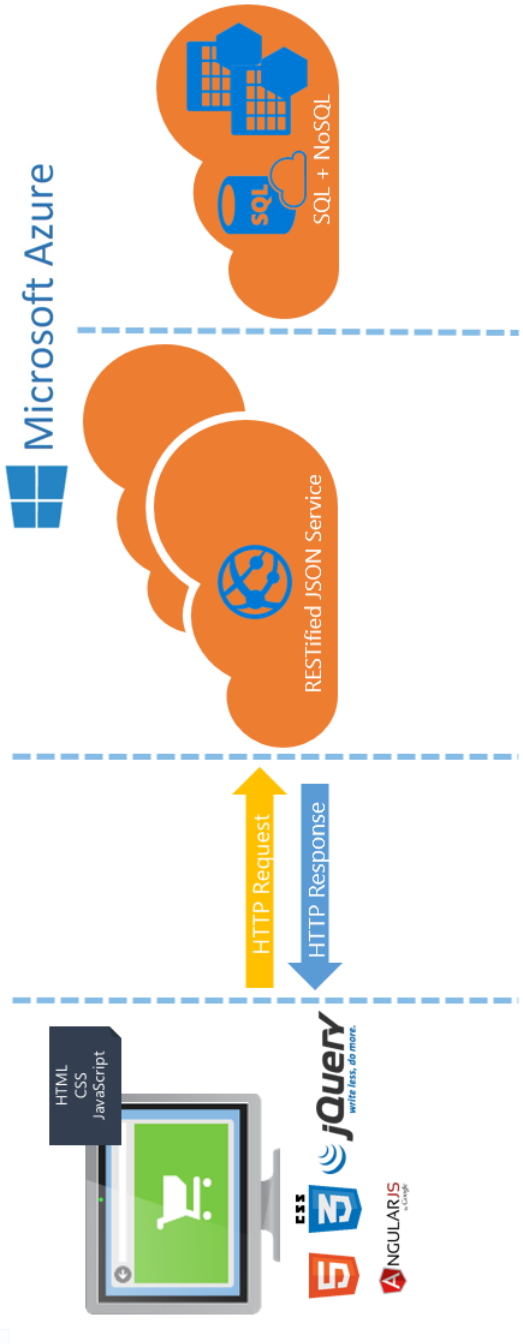

Abbildung 5.2: Systemarchitektur von WoTCloud.

5.5.2 Data Layer

Nachdem die WoT-Plattform von Beginn an vollständig als Software-as-a-Service Lösung konzipiert wurde, musste eine Entscheidung hinsichtlich Datenarchitektur und Speichertechnologien getroffen werden. Hierfür entwickelten die Autoren zuerst das nötige UML-Klassendiagramm zur Abbildung der Entitäten und Assoziationen. Anschließend wurde eine hybride Datenarchitektur auf Basis von Microsoft Azure für Datenzugriff und Datenhaltung entwickelt.

Datenmodell

Das Datenmodell entspricht dem konzeptuellen Modell aus Abschnitt 5.3. Abbildung 5.1 skiziert das entwickelte Datenmodell als Klassendiagramm der UML und stellt die Struktur der verwendeten Entitäten dar.

Hybride Datenarchitektur

Für die Umsetzung des oben genanten Entitätsmodells wurde eine hybride Datenarchitektur mit den von Microsoft Azure gebotenen Diensten realisiert. Konkret wurden die Dienste Azure SQL Database sowie Azure Table Storage für die Realisierung eingesetzt.

Microsoft Azure SQL Database ist ein relationales Datenbankmanagementsystem (RDBMS) in der Cloud und unterstützt klassische relationale ACID-Transaktionen [43]. Im Wesentlichen basiert dieser Service auf der SQL Server Engine. Azure SQL Database eignet sich somit zur Speicherung von schematisch, strukturierten Daten unter Sicherstellung der referenziellen Integrität.

Im Gegensatz dazu ist *Azure Table Storage* ein flexibler, cloudbasierter Key/Value-Speicher zur Speicherung von schemalosen Applikationsdaten [43]. Diese Speichertechnologie verzichtet auf Konzepte wie Joins, Stored Procedures und referenzielle Integrität und ermöglicht die Speicherung von Tupeln mit unterschiedlicher Struktur in einer Tabelle. Dieser Speicheransatz zielt auf die kostengünstige und performante Speicherung von großen Datenmengen ab, wobei die Daten denormalisiert sind [37].

Aufgrund der Tatsache, dass beide Ansätze jeweils beachtliche Vorteile bieten, ließ die Autoren zum Schluss kommen, eine hybride Architektur unter Verwendung beider Speicherdienste zu implementieren. Das in Abbildung 5.1 skizzierte Datenmodell könnte dabei vollständig in *Azure SQL Database* implementiert werden. Betrachtet man jedoch die Vielzahl an erwarteten Write-Transaktionen von konkreten Sensorwerten, welche durch einen angebundenen Automaten versendet werden, würde man hierbei sehr schnell an die Grenzen (max. Datenbankgröße[1]: 500 GB) stoßen. Deshalb

[1] http://azure.microsoft.com/de-de/pricing/details/sql-database

wird für die performante Speicherung und den Zugriff auf Sensorwerte *Azure Table Storage* verwendet, da die Sensorwerte ohnehin kein striktes relationales Schema mit referenzieller Integrität aufweisen. Ein weiterer Grund für den Einsatz von *Azure Table Storage* ist das kostengünstigere Preismodell für große Datenmengen.

5.5.3 Service Layer

Der Service Layer als zentraler Baustein der Plattform wurde als Sammlung von REST-basierten Webservices entwickelt. Hauptaufgabe der Services ist es, Repräsentationen von Ressourcen (z. B. Sensordaten für ein bestimmtes Thing) aufzubereiten und zur Verfügung zu stellen. Der Service Layer dient sowohl als zentraler Datenhub für Sensoren, als API für spätere Anwendungsszenarien (z. B. jegliche Form von Apps (Smartphone, Tablet, etc.), Integrationen in andere Softwarekomponenten, etc.) sowie auch als Basis für den Webclient, der selbst ausschließlich auf diesen Webservices basiert.

Technologisch wurde der Service Layer mit Microsoft's ASP.NET Web API[2] und dem .NET Framework realisiert. Als Entwicklungsumgebung für den Service Layer verwendeten die Autoren Visual Studio[3] von Microsoft. Neben C#, .NET und ASP.NET Web API wurden noch folgende Frameworks/Toolkits in Form von NuGet-Packages eingesetzt: Newtonsoft JSON, .NET Entity Framework (O/R-Mapper) und OWIN[4].

Newtonsoft JSON ist dabei ein JSON-Serialisierer für das Microsoft .NET-Framework. Entity Framework ermöglicht eine objektrelationale Zuordnung von Objekten zu relationalen Daten und kapselt somit wesentlich den Datenzugriff. OWIN umfasst Komponenten zum Entkoppeln der eigentlichen Webapplikation vom zugrundeliegenden Webserver und vereinfacht damit Portierungen.

Webservices

Wie bereits erwähnt, basiert der Server auf einzelnen, voneinander unabhängigen Webservices. Dies impliziert, dass alle Things vollständig im Sinne einer ROA als Ressourcen modelliert und in eine URI-Struktur eingegliedert wurden. So ist beispielsweise der Zugriff via HTTP GET auf ein einzelnes Thing über folgende URI möglich:

http://.../api/*{tenantId}*/things/*{thingId}*

Prinzipiell sind diese Webservices an den von Roy Fielding entworfenen Architekturstil REST angelehnt (Zustandslosigkeit, lose Kopplung, HTTP-Methoden, URIs) [19], setzen aber nicht alle Konzepte (vor allem Hypermedia

[2] http://www.asp.net/web-api/
[3] https://www.visualstudio.com/
[4] Open Web Interface for .NET - http://owin.org/

as the Engine of Application State (HATEOAS)) vollständig um. Somit handelt es sich nicht um RESTful-Webservices im engeren Sinn. Darüber hinaus basieren unsere Webservices vollständig auf HTTP, eine Tatsache, die in einem REST-Umfeld nicht zwingend nötig ist, jedoch meist impliziert wird [54]. Die Kommunikation des Clients mit diesen Webservices erfolgt ausschließlich über HTTP-Requests und HTTP-Responses (siehe Abbildung 5.2) und bietet als Datenformat neben XML bevorzugt JSON an.

Als Basistechnologie zur Umsetzung dieser auf REST und HTTP-basierten Webservices dient *ASP.NET Web API*. Dabei handelt es sich um ein sehr leichtgewichtiges Framework zur Entwicklung HTTP-basierter Services, welches vollständig auf dem .NET Framework von Microsoft basiert. Web API bietet einen direkten Zugriff auf die Möglichkeiten von HTTP und abstrahiert, im Gegensatz zur Windows Communication Foundation (WCF), nicht das darunter liegende Protokoll [54]. Die Tatsache, dass ASP.NET Web API auch um einiges leichtgewichtiger und somit auch weniger komplex als WCF ist, lässt es vor allem für jene Projekte, deren Kommunikation sich auf HTTP stützt, bevorzugt zum Einsatz kommen. Laut Steyer & Schwichtenberg [54] ermögliche dieses Framework auch eine Implementierung von RESTful Webservices. Die korrekte Implementierung jedoch liege vollständig in den Händen der Entwickler. So könne sich der Entwickler „gegen REST-ful URLs entscheiden, pro Ressource nur ein Datenformat erlauben oder auf den Einsatz von Hyperlinks im Rahmen der retournierten Daten verzichten“ [54].

5.5.4 Presentation Layer

Parallel zu den vorher genannten Schichten entwickelten die Autoren einen Webclient als Benutzerschnittstelle. Der Webclient zeichnet dabei verantwortlich für die Präsentation der gesamten Daten an der Benutzerschnittstelle.

Als Entwicklungsumgebung für den Presentation Layer verwendeten die Autoren WebStorm[5] von JetBrains. Neben HTML, CSS und JavaScript wurden für das User Interface folgende Frameworks/Toolkits bzw. Technologien eingesetzt: AngularJS, Twitter Bootstrap, jQuery, Telerik Kendo UI, Angular Google Maps, Angular UI Router und einige mehr.

Single Page Anwendung (SPA)

Der Webclient wurde als sogenannte „Single Page“-Anwendung konzipiert, sodass die Applikation im Wesentlichen aus einer einzigen HTML-Seite besteht und deren Inhalte dynamisch nachgeladen werden.

In klassischen Webanwendungen liefert der Server auf Anfragen des Clients immer vollständige HTML-Dokumente aus. Dies löst beim Webclient

[5] https://www.jetbrains.com/webstorm

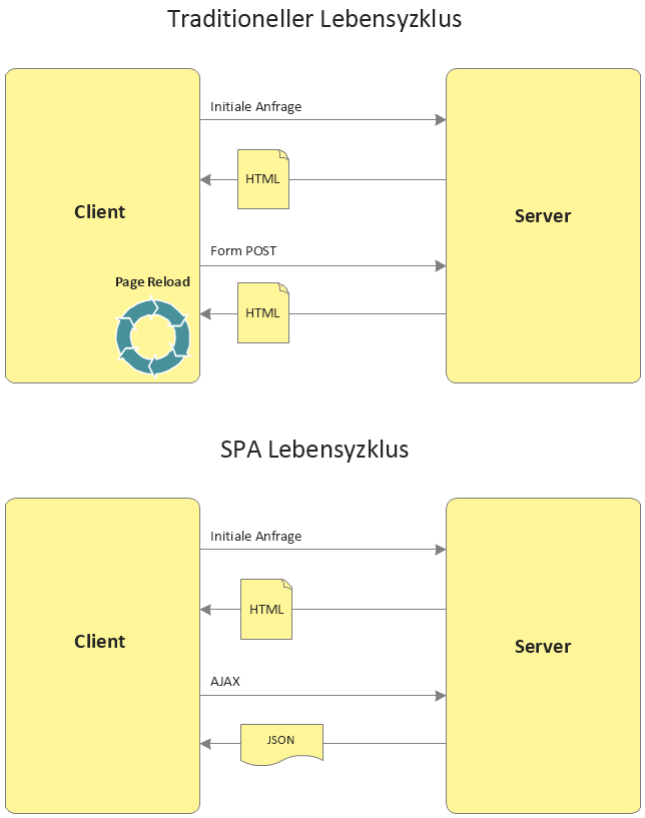

Abbildung 5.3: Der traditionelle Lebenszyklus gegenübergestellt mit dem Single-Page-App Lebenszyklus einer Seite (angelehnt an [60]).

einen sogenannten „Page Reload“ aus, der in einer Neudarstellung des retournierten HTML-Dokuments resultiert. Bei Single Page Anwendungen hingegen wird nur initial ein HTML-Dokument vollständig geladen. Alle weiteren Anfragen erfolgen über „Asynchronous JavaScript and XML“-Aufrufe. Der Server liefert anschließend eine Antwort in einem durch Content-Negotiation spezifizierten Format, in unserem Fall, als JSON-Dokument. Diese JSON-Daten werden dann vom Browser verwendet, um die Inhalte einer Seite dynamisch auszutauschen. Abbildung 5.3 stellt die Unterschiede beider Ansätze dar.

Ein wesentlicher Vorteil von SPAs ist die gesteigerte Interaktivität und Performanz der Anwendung, da anstatt von Markup-Code hauptsächlich Daten in Form von JSON ausgetauscht werden. Angular UI Router[6] stellt dabei das zentrale und unterstützende Framework zur Umsetzung dar.

[6] https://github.com/angular-ui/ui-router

5.6 Zusammenfassung

Dieses Kapitel diente der Beschreibung der grundsätzlichen Anforderungen und der Umsetzung der selbst entwickelten WoTCloud-Plattform. Es wurden Ziele definiert und konkret Anforderungen aus diesen Zielen abgeleitet, die aus den Ergebnissen der Befragungen aus Kapitel 3 und dem Vergleich aus Kapitel 4 resultieren. Darauf aufbauend entwickelten die Autoren unter Berücksichtigung komplexer Dinge ein konzeptuelles Modell, das auch als Datenmodell dient. Die konkrete Implementierung von WoTCloud basiert auf einer logischen Trennung in drei Schichten: Data Layer, Service Layer und Presentation Layer. Es wurde insbesondere die hybride Datenarchitektur des Data Layers, die REST-basierte Architektur des Service Layers und die Umsetzung des Presentation Layers als Single Page Application vorgestellt.

Teil II

Spezifische Aspekte der Web-of-Things-Plattform

Kapitel 6

REST-basierte Architekturen im Web-of-Things

Representational State Transfer, kurz REST, ist ein von Thomas Roy Fielding [19] beschriebener Architekturstil zur Entwicklung verteilter Anwendungen. REST-basierte Webservices können als architektonische Grundlage des Web-of-Things angesehen werden. Aus diesem Grund werden in diesem Kapitel die Konzepte von REST detailliert erläutert und diskutiert, warum REST gerade für das Web-of-Things von Bedeutung ist.

Weiters wird das konzeptuelle Modell aus Abschnitt 5.3 in ein Ressourcenmodell transferiert und die konkrete Umsetzung der Webservices von WoTCloud diskutiert. Abschließend werden verschiedene Deployment-Szenarien für Smart Things und Web-of-Things-Plattformen erläutert, sowie ein Simulator für das Smart Vending Szenario vorgestellt.

6.1 Einleitung/Motivation

Viele der in der Vergangenheit verfügbaren Webservices basierten auf eigenen, proprietären Schnittstellen um ihre Services zu veröffentlichen. Dieser Umstand führte zu einer Reduktion der Interoperabilität und Skalierbarkeit und in Folge dessen zu einer Erhöhung des Antwortzeitverhaltens [35]. Um hoch verfügbare und interoperable Webapplikationen zu entwickeln und zu betreiben, ist die Verwendung von gewissen Standards notwendig. Das Web - als größte verfügbare Webapplikation - ist nicht durch eine spezifische Implementierung, sondern durch seine standardisierten Schnittstellen und Protokolle definiert [35]. Es skalierte in den letzten beiden Jahrzehnten von wenigen tausenden bis hin zu Milliarden von Requests pro Tag[1].

Auch im Bereich des Internet-of-Things ist eine “gemeinsame Sprache”, die von einer Vielzahl von Geräten - vom Kühlschrank über den Fernseher

[1] http://www.internetlivestats.com/

hin zu Getränkeautomaten und Autos - verstanden und verarbeitet werden können, essentiell. Das Web-of-Things zielt genau auf diesen Umstand ab: Smart Things und Applikationen sollen über etablierte und weit verbreitete Webstandards bzw. Protokolle miteinander verbunden werden, um eine Interaktion mit und zwischen den Dingen zu ermöglichen [31]. In den meisten früheren webbasierten Ansätzen wurde das Hypertext Transfer Protocol (HTTP) nur zum Transport von Daten zwischen Geräten und nicht, wie eigentlich intendiert, als Applikationsprotokoll verwendet. Anstatt das Web als reines Transportmedium zu nutzen, werden bei einer REST-basierten Architektur hingegen Smart Things als integraler Teil des Webs angesehen und im Gegensatz zu klassichen WS-* Webservices auch HTTP als Applikationsprotokoll genutzt [31, 32]. Der große Vorteil dieser "Integration der Dinge" liege laut Wilde [61] darin, dass sie Things wie jede beliebige Ressource im Web behandle und diese folglich in den unterschiedlichsten Kontexten und Applikationen wiederverwendet werden könnten. Dies übersteige auch die Möglichkeiten von APIs, die meist nur einem begrenzten Benutzerkreis zur Verfügung stehen und nur limitierte Zugangs- und Interaktionsmöglichkeiten anbieten würden. Eine REST-basierte Architektur bietet zudem den Vorteil, dass keine zusätzliche API bzw. auch keine zusätzliche Beschreibung der Ressourcen/Funktionen notwendig ist [31].

In den letzten vier Jahren sind REST-basierte Services immer populärer geworden. So haben sie nicht nur klassische, schwergewichtige Webservices (wie beispielsweise SOAP/WS-*) überholt, sondern mit 69 Prozent Marktanteil an Internet APIs[2] bereits eine bedeutende Mehrheit erlangt. Während zu Beginn nur vereinzelte Webapplikationen REST-basierte Services konsumierten, entwickelt sich vor allem aufgrund der zunehmenden mobilen Endgeräte und dem umfangreichen Serviceangebot vieler Cloud Dienstleister neuerdings ein Trend zur Verwendung öffentlicher, wiederverwendbarer REST-Services. So hat beispielsweise auch Tesla[3] eine öffentliche REST-Schnittstelle für das Tesla Model S, welche zwar eine Authentifizierung erfordert, aber dennoch die Ansteuerung grundlegender Funktionen des Autos erlaubt (z. B. Auf-/Zusperren, Temperaturregelung, etc.).

6.2 Representational State Transfer (REST)

Jener Architekturstil, dem das Web zugrunde liegt, wird als Representational State Transfer (REST) bezeichnet. Der Usprung von REST liegt in der Dissertation von Roy Thomas Fielding aus dem Jahre 2000 [19], der auch federführend an der Spezifikation von HTTP 1.1 beteiligt war, und beantwortet den Bedarf der Internet Engineering Task Force (IETF) nach einem Modell zur Beschreibung wie das Web funktionieren *sollte* [35]. REST ist

[2] http://www.google.com/trends/explore?hl=en-US#q=soap+api,+rest+api&cmpt=q
[3] http://docs.timdorr.apiary.io/#reference/vehicle-commands/unlock-doors/unlock-doors

ein hybrider Architekturstil, der von mehreren netzwerkbasierten Architekturstilen (z. B. Client-Server, Layered System, Code on Demand) abgeleitet und mit zusätzlichen Einschränkungen hinsichtlich eines einheitlichen Interfaces angereichert worden ist [19]. Der primäre Zweck von REST ist die Etablierung eines Architekturstils für verteilte Hypermedia-Applikationen, die leichtgewichtig, wartbar und skalierbar bleiben sollen [50]. REST ist ein Architekturstil, also weder ein Standard noch ein Protokoll. Da REST von Architektur-Elementen in verteilten Hypermedia-Systemen abstrahiert und sich hauptsächlich mit der Rolle und der Funktion von Ressourcen beschäftigt, ist es grundsätzlich unabhängig von konkreten Implementierungen und der Syntax und Semantik von spezifischen Protokollen [19]. Nichtsdestotrotz ist fast jedes REST-basierte Webservice auf Grundlage von HTTP implementiert. Der Fokus auf HTTP erscheint logisch, da Fieldings Arbeit auch als Richtlinie für die Definition der Standards von HTTP 1.1 und URIs herangezogen wurde [29].

Seit seiner Einführung im Jahr 2000 hat REST zunehmend an Bedeutung gewonnen und ist nun eine der wichtigsten Architekturkonzepte für Webapplikationen. Ein Grund für die rasche Verbreitung ist laut Richardson et al. [50] auf den Umstand zurückzuführen, dass immer mehr Technologien bzw. Applikationen eine Serviceorientierung anstreben. Sie führen weiters an, dass viele heutige Programmiersprachen Frameworks zur Erstellung REST-basierter Webservices anbieten. Aufgrund der großen Verbreitung wird REST auch oft als Marketingbegriff zweckentfremdet. Insbesondere die Definition eines “RESTful” Webservices, also ein Webservice, welches alle Konzepte/Einschränkungen von REST umsetzt, wird in der Praxis oft missbräuchlich verwendet. Laut Richardson [50] sei den wenigsten Entwicklern bewusst, was Hypermedia wirklich bedeute. Folglich sind viele der verfügbaren “REStful” Webservices zwar an den meisten REST-Konzepten angelehnt, in Wirklichkeit aber eher eine Mischung aus REST und klassischem Remote Procedure Call (RPC).

6.2.1 Die Evolution von REST als Architekturstil

Der von Roy Fielding entworfene Architekturstil REST kann als hybrider Architekturstil angesehen werden, der von einer Vielzahl an bestehenden Architekturstilen bzw.-konzepten beeinflusst und abgeleitet worden ist. Fielding [18, 19] startete seine Überlegungen basierend auf einem “Null-Style”, also einem System ohne jegliche Einschränkungen zwischen den einzelnen Komponenten. Die erste Einschränkung, die er hinzufügte war die des Client-Server-Stils (CS), um eine Trennung von User Interface und Datenzugriff (seperation of concerns) und eine Co-Evolution der Komponenten zu ermöglichen. Er verfeinerte dies noch insofern, als er grundsätzlich nur zustandslose Kommunikation zwischen Client und Server (CSS) zuließ, um die Zuverlässigkeit, Skalierbarkeit und Sichtbarkeit (visibility) zu erhöhen. Die zustands-

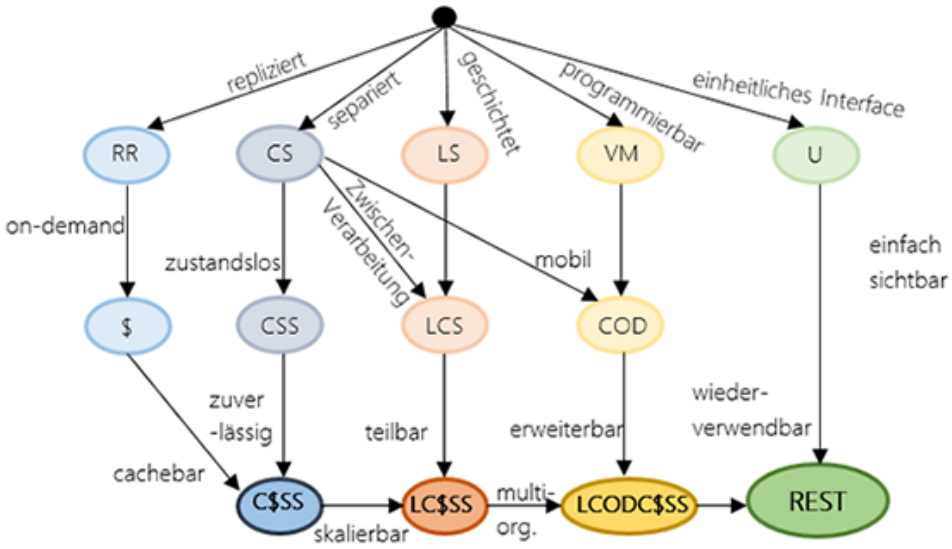

Abbildung 6.1: Die Evolution von REST als Architekturstil (angelehnt an [19])

lose Kommunikation erfordert, dass jeder Request alle erforderlichen Daten enthalten muss, sodass der Server diese Anfrage ohne zusätzliches Wissen erfolgreich verarbeiten kann. Dies führt zu einer Verschlechterung der Netzwerkperformance, da vermehrt repetetive Daten übertragen werden. Um die Effizienz des Netzverkehrs zu erhöhen, ist die nächste Einschränkung die Integration von Caching-Mechanismen, die zu einem zustandslosen Client-Cache-Server Stil (C$SS) führt.

Fielding [18, 19] erweiterte den C$SS-Stil um weitere Konzepte. Die wohl wichtigste weitere Einschränkung ist die einer einheitlichen Schnittstelle (Uniform Interface) zwischen den Komponenten, welche als primäre Unterscheidung zu anderen netzwerkbasierten Stilen angesehen werden kann. Um eine einheitliche Schnittstelle zu ermöglichen, sind laut Fielding folgende Einschränkungen notwendig: Die Identifikation von Ressourcen, die Manipulation von Ressourcen durch Repräsentationen, selbst-beschreibende Nachrichten und Hypermedia as the Engine of Application State (HATEOAS). Diese Einschränkungen werden im nächsten Abschnitt näher vorgestellt. Zusätzlich zu einer einheitlichen Schnittstelle hatte Fielding noch zwei weitere Einschränkungen für REST. Die Erste, Layered System (LS), erlaubt den Entwurf von hierarchischen Architekturen (z. B. durch den Einsatz von Proxies), wobei jeder Schicht nur der unmittelbare Kommunikationspartner bekannt ist. Die Zweite, Code-on-Demand (COD), ist optional und erlaubt es einem Client dynamisch Skripte vom Server zu laden und auszuführen. Zusammengefasst kann REST, wie in Abbildung 6.1 dargestellt, aus einer

Kombination von einem Layered System (LS), Code-On-Demand (COD), einer zustandslosen Client-Cache-Server (C$SS) Architektur und einer einheitlichen Schnittstelle (U) angesehen werden [18, 19].

6.2.2 REST-basierte Webservices

Aus Sicht eines Softwareentwicklers stellt sich nun die Frage, wie eine konkrete Implementierung von REST-basierten Webservices erfolgen kann. Die grundlegende Annahme in diesem Abschnitt ist, dass sowohl HTTP als Applikationsprotokoll als auch URLs für die eindeutige Adressierung eingesetzt werden. Laut Richardson et al. [50] setze REST per se keine Protokolle voraus, nahezu alle praktischen Implementierungen seien jedoch auf Basis von HTTP und URLs umgesetzt. Darüber hinaus habe REST die Weiterentwicklung der Spezifikationen von HTTP und URIs maßgeblich beeinflusst.

In diesem Abschnitt werden nun die wichtigsten Konzepte zur Implementierung einer webbasierten, "RESTful" API erläutert. Diese Konzepte sind grundsätzlich völlig unabhängig von der eingesetzten Softwareplattform bzw. Programmiersprache (z. B. C# / ASP.NET, Java, Node.js, etc.).

Eindeutige adressierbare Ressourcen Der zentrale Gedanke von REST ist, dass alle Komponenten einer Applikation in Form von Ressourcen abstrahiert werden. Eine Ressource wird in diesem Zusammenhang als "any component of an application that is worth being uniquely identified and linked to" [32] definiert. Laut Richardson et al. [50] könne eine Ressource also grundsätzlich alles sein: von einem menschlichen Wesen über die Farbe Schwarz bis hin zu einer Menge an Primzahlen. Sie weisen jedoch darauf hin, dass typischerweise (Informations-)Ressourcen aber jene Dinge sind, die auch in einem Informations- und Kommunikationssytem verarbeitet und persistiert werden.

> *The central idea of REST revolves around the notion of resource as any component of an application that needs to be used or addressed. Resources can include physical objects (e.g. temperature sensors) abstract concepts such as collections of objects, but also dynamic and transient concepts such as server-side state or transactions. [29]*

Während die Semantik einer Ressource statisch ist, verändert sich der eigentliche Inhalt dynamisch [35]. Für eine Ressource gibt es nur eine wichtige Einschränkung: Jede Ressource muss über eine URI eindeutig adressierbar sein. Im Falle von Webservices und dem Einsatz von HTTP(S) sollte jede Ressource einen eindeutigen Uniform Ressource Locator (URL) besitzen. Listing 6.1 veranschaulicht eine konkrete URL eines Things mit der ID 137. Die generische Struktur (*Protocol://ServiceName/ResourceType/ResourceID*)

Listing 6.1: Eindeutige Adressierung von Ressourcen

```
1 https://wotcloud.azurewebsites.net/things/137
```

wird in diesem Beispiel durch das Protokoll HTTPS, den Servicenamen "wotcloud.azurewebsites.net", den Ressourcentyp "things" und die Ressourcen-ID 137 konkretisiert. Es ist anzumerken, dass Ressourcen bzw. Ressourcentypen beliebig geschachtelt werden können.

URI vs. URL Die Begrifflichkeiten URI und URL führen häufig zu Verwirrungen. Während ein Uniform Ressource Identifier (URI) eine allgemeine Identifizierung eines Objektes spezifiziert (beispielsweise ein Buch über eine eindeutige ISBN-Nummer), ist ein Uniform Ressource Locator (URL) spezifischer. Beide Begriffe sind in RFC 3986 spezifiziert:

> *A URI can be further classified as a locator, a name, or both. The term "Uniform Resource Locator (URL)" refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location"). [4]*

Der wesentliche Unterschied liegt also darin, dass eine URL auch die Ressource dereferenzieren kann. Folglich ist für REST-basierte Webservices eine URL relevant, da nur für jene Ressourcen, die mit einer eindeutigen URL adressiert sind, Repräsentationen abgerufen werden können [50]. Ohne Repräsentationen gibt es auch keinen Representational State Transfer (REST). Aufgrund dessen, dass jede Ressource eine eindeutige Adresse erhält, wird auch ermöglicht, dass Ressourcen bzw. deren Repräsentationen miteinander verlinkt werden. Eindeutig adressierbare Ressourcen stellen somit eine wichtige Grundlage für weitere REST-Konzepte (Einheitliche Schnittstelle, HATEOAS) dar.

Repräsentationen REST Komponenten führen ihre Operationen nicht direkt auf Ressourcen sondern auf deren Repräsentationen durch. Eine Repräsentation ist ein Abbild des aktuellen oder gewünschten Zustands einer Ressource und enthält sowohl die eigentlichen Daten der Ressource als auch Metadaten der Ressource und der Repräsentation [18]. Ein Client kommt also nie mit der eigentlichen Ressource, sondern nur mit ihrer Repräsentation in Berührung. Der Server sendet Repräsentationen, um den Zustand einer Ressource zu beschreiben. Der Client sendet ebenfalls eine Repräsentation, um den gewünschten Zustand einer Ressource zu beschreiben. Diese Vorgehensweise hat einige Vorteile. Erstens wird durch diese Trennung ermöglicht, dass eine Ressource mehrere Repräsentationen in unterschiedlichen Datenformaten (z. B. JSONs, XMLs, CSVs etc.) erhält. Zweitens wird durch eine

Listing 6.2: Repräsentation einer Ressource in JSON (HTTP-Request & HTTP-Response)

```
GET http://wotcloud.azurewebsites.net/api/1/things/109 HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content_Type: application/json
{
    "id": 109,
    "name": "Sielaff FK 230 - Engel1",
    "description": "Aufgestellt bei Kunde: Engel GmbH in Schwertberg",
    "latitude": 48.30694,
    "longitude": 14.28583,
    ...
}
```

Listing 6.3: Repräsentation einer Ressource in XML (HTTP-Request & HTTP-Response)

```
GET http://wotcloud.azurewebsites.net/api/1/things/109 HTTP/1.1
Accept: application/xml
```

```
HTTP/1.1 200 OK
Content_Type: application/json
<?xml version="1.0" encoding="UTF-8" ?>
  <id>109</id>
  <name>Sielaff FK 230 - Engel1</name>
  <description>Aufgestellt bei Kunde: Engel GmbH in Schwertberg</
    description>
  <latitude>48.30694</latitude>
  <longitude>14.28583</longitude>
</xml>
```

Repräsentation ein Zustand einer Ressource beschrieben. Folglich wird durch die Modifikation einer Repräsentation am Client eine gewünschte Zustandsänderung einer Ressource am Server beschrieben.

Listing 6.2 veranschaulicht die Abfrage einer Repräsentation der Ressource “Thing” mit der ID 109 unter Angabe des gewünschten Datenformats - in diesem Fall JSON. Im Gegensatz dazu wird in Listing 6.3 eine Repräsentation derselben Ressource in XML geladen. Die Angabe des gewünschten Datenformats erfolgt im Accept-Teil des HTTP-Headers. Dieser Prozess wird auch als *content-negotiation* bezeichnet. Die zulässigen Werte und Details für diesen Header sind in RFC 2616 spezifiziert [22].

Zustandslosigkeit Das Konzept der Zustandslosigkeit der Client-Server Kommunikation erfordert, dass jeder Request vom Client zum Server alle notwendigen Informationen erhält, die der Server benötigt, um die Anfrage

zu verstehen und verarbeiten zu können. Folglich bleibt jeglicher Session-State ausschließlich am Client [19]. Laut Fielding [18] dient diese Einschränkung folgenden Zwecken:

- Es werden weniger phyische Ressourcen benötigt und die Skalierbarkeit wird erhöht, da kein Applikationszustand zwischen den Requests gespeichert werden muss
- Es wird ermöglicht, dass Interaktionen parallel ausgeführt werden können ohne die Interaktionssemantik überprüfen zu müssen
- Es erlaubt auch Intermediären (z. B. Proxy, Cache, etc.) einzelne Requests (isoliert) zu verarbeiten und gegebenenfalls weiterzuleiten
- Es erfordert alle Informationen in einem Request, die auch ein Cache für die Beantwortung benötigen würde

HTTP unterstützt dieses Konzept originär, da es keine HTTP-Sessions oder Transaktionen gibt. Guinard [29] weist darauf hin, dass aber trotzdem ein Interaktionszustand möglich ist, der entweder im Request eingebettet ist (Cookies) oder in der Form eines serverseitigen Zustands (z. B. für Authentifizierung und Authorisierung) existiert. Dennoch ist die Interaktion an sich vollständig unabhängig (es ist kein Kontext für eine Interpretation notwendig) und somit zustandslos.

Einheitliches Interface und selbstbeschreibende Nachrichten Wie schon in Abschnitt 6.2.1 erwähnt, versteht Fielding [19] unter einem einheitlichen Interface eigentlich die folgenden Einschränkungen: die Adressierbarkeit von Ressourcen und die Manipulation von Ressourcen über Repräsentationen, wie zuvor beschrieben. Darüber hinaus sind noch selbstbeschreibende Nachrichten und die Verlinkung von Ressourcen relevant. Für die konkrete Implementierung von Webservices sind im Sinne eines einheitlichen Interfaces vor allem die selbstbeschreibenden Nachrichten, also die wohldefinierte Protokollsemantik von HTTP als Applikationsprotokoll relevant. Der HTTP-Standard definiert acht unterschiedliche Arten von Nachrichten, von denen die wichtigsten sechs in Tabelle 6.1 beschrieben sind. In Entwicklerkreisen wird das Konzept des einheitlichen Interfaces vielfach auf die Verwendung dieser HTTP-Verben reduziert.

Die Tabelle veranschaulicht, dass es Unterschiede in der Qualität der HTTP-Methoden gibt. Jene Methoden, die als sicher gekennzeichnet sind, wie beispielsweise GET, HEAD und OPTIONS, verändern den Zustand einer Ressource nicht [50]. Idempotente Methoden führen zwar zu einer Zustandsänderung, können jedoch beliebig oft ohne Seiteneffekte wiederholt werden. So führt auch ein dreimaliger DELETE-Aufruf nur dazu, dass eine Ressource einmal gelöscht wird und folglich nicht mehr existent ist. Einen Sonderfall stellt die POST-Methode dar. Sie ist weder sicher noch idempotent. Wird ein POST-Request wiederholt ausgeführt, so werden mehrere Ressourcen auf Basis der übermittelten Repräsentation erzeugt. Der Zu-

Methode	Beschreibung	Qualität
GET	Ruft eine Repräsentation einer Ressource ab	Sicher
PUT	Ersetzt den Zustand einer Ressource mit den der übermittelten Repräsentation	Idempotent
POST	Erzeugt eine neue Ressource, die hierarchisch unter der aktuellen Ressource eingeordnet wird, basierend auf der übermittelten Repräsentation	Unsicher
DELETE	Löscht eine Ressource	Idempotent
OPTIONS	Listet die erlaubten HTTP-Methoden einer Ressource auf	Sicher
HEAD	Retourniert nur den HTTP-Header einer Repräsentation ohne den eigentlichen Inhalt	Sicher

Tabelle 6.1: HTTP-Methoden/-Verben

stand der erzeugten Ressourcen ist zwar gleich, sie haben aber alle eine unterschiedliche Adresse erhalten. Hintergrund ist die Tatsache, dass die URL der zu erzeugenden Ressource nicht bekannt ist, weshalb bei einem POST-Request auch jene Ressource im Request adressiert wird, an die die übermittelte Repräsentation anzuhängen ist (POST-to-append) [50]. Nun wird auch der Unterschied zwischen PUT und POST ersichtlich: Während ein POST-Request eine neue Ressource erzeugt, deren Adresse er zuvor nicht kennt, muss einem PUT-Request die Adresse schon bekannt sein. Nichtsdestotrotz kann theoretisch auch durch einen PUT-Request eine Ressource angelegt werden. Es ist jedoch nicht möglich, dieselbe Ressource mehrfach anzulegen, da diese Methode idempotent ist und die Adresse immer vollständig spezifiziert sein muss. Neben den HTTP-Methoden gibt es auch noch eine Fülle an HTTP-Statuscodes, die an dieser Stelle nicht näher beleuchtet werden, da sie für den Kern der Arbeit nicht relevant sind. Für eine tiefgehendere Betrachtung der Statuscodes sei an weiterführende Literatur [22, 50] verwiesen.

Mit selbstbeschreibenden Nachrichten ist Nachrichtenaustausch zwischen Client und Server gemeint, der kein über die Nachricht hinausgehendes, zusätzliches Wissen erfordert. Ob eine Nachricht an sich aber wirklich selbstbeschreibend ist, hängt stark von jenem Format (Medientyp) ab, auf das sich Client und Server im Zuge der gegenseitigen Kommunikation geeinigt haben (*content negotiation*). Das Web bietet unterschiedlichste Möglichkeiten

für Medientypen an, die mittels HTTP ohne individuelle Service-Kontrakte benützt werden können [29]. Die meisten maschinenorientierten Services nützen XML oder JSON als Medientyp, da diese Formate weit verbreitet sind und von vielen Plattformen umfangreich unterstützt werden. JSON ist gegenüber XML leichtgewichtiger und kann direkt in Javascript-Objekte geparst werden [29].

Caching Unter Caching versteht man das Zwischenspeichern von Ressourcen, um unnötigen Netzverkehr bzw. Serveranfragen zu vermeiden. Caching wird erst durch die Konzepte der Zustandslosigkeit und den selbstbeschreibenden Nachrichten ermöglicht und basiert auf einer speziellen Markierung von HTTP-Responses. Diese Markierungen kennzeichen, ob und wie es möglich ist gewisse Daten zu cachen [18]. Caching ist eines der komplexesten Themen von HTTP und wurde 2014 in RFC 7234 leicht abgeändert [21]. Die einfachste Möglichkeit ist die Benützung des Cache-Control-Headers unter Angabe eines Gültigkeitszeitraumes von Daten (z. B. max-age=3600 Sekunden) [50]. Neben Cache-Control gibt es noch einige weitere Felder im Caching-Header (z. B. Age, Last Modified, Expires, Pragma, Warning, etc.). Das Ziel von Caching ist die Erhöhung der Performance einer Applikation oder um in es in Fieldings Worten auszudrücken [19]: "the best application performance is obtained by not using the network."

HATEOAS Hypermedia as the Engine of Application State (HATEOAS) ist das zentrale aber auch umstrittenste und meist diskutierte Konzept von REST[4,5]. Die zentrale Idee von HATEOAS ist, dass der Applikationszustand des Clients nur von den dynamischen (Hypermedia)-Informationen des Servers getrieben wird. Dazu werden dem Client nicht nur die angeforderte Repräsentation, sondern auch Links zu weiteren relevanten Ressourcen geliefert [50]. Ein REST-Client würde somit kein vordefiniertes Wissen über die Ressourcen-Hierarchie benötigen. Laut Guinard [29] "erforsche" der Client ein Service ohne explizite, proprietäre Discovery-Formate zu kennen, indem er standardisierte Identifier und wohldefinierte Medientypen benütze [29] und so von einem Zustand zum Nächsten wechsle. Er weist jedoch darauf hin, dass die Verlinkung der Ressourcen über deren Repräsentationen in einer standardisierten Form erfolgen sollte. Das beste Beispiel für dieses Konzept ist das World Wide Web selbst. Jeder Browser ruft von Webservern Hypertext-Repräsentationen (HTML) von Ressourcen ab. Eine Zustandsänderung am Client wird durch den Aufruf von verlinkten Repräsentationen (Hyperlinks), beispielsweise in dem der Benutzer im Browser auf einen Link klickt, erreicht.

[4] http://jeffknupp.com/blog/2014/06/03/why-i-hate-hateoas/

[5] http://stackoverflow.com/questions/9204110/restful-api-runtime-discoverability-hateoas-client-design?rq=1

Listing 6.4: Verlinkung von Ressourcen über Repräsentationen (HTTP-Request & HTTP-Response)

```
GET http://wotcloud.azurewebsites.net/api/1/things/114 HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content_Type: application/json
{
  "id": 114,
    "name": "Sielaff FK 230 - Engel 2",
    "description": "Aufgestellt bei Kunde: Engel GmbH in Schwertberg",
    "latitude": 48.30694,
    "longitude": 14.28583,
    "links": [
        {
            "Rel": "self",
            "Href": "http://wotcloud.azurewebsites.net/api/1/things/114"
        },
        {
            "Rel": "sensors",
            "Href": "http://
    wotcloud.azurewebsites.net/api/1/things/114/sensors"
        },
        {
            "Rel": "actuators",
            "Href": "http://
    wotcloud.azurewebsites.net/api/1/things/114/actuators"
        },
        {
            "Rel": "rules",
            "Href": "http://wotcloud.azurewebsites.net/api/1/things/114/rules"
        }]
}
```

Listing 6.4 veranschaulicht das Prinzip HATEOAS am Beispiel der Verlinkung eines Things und seinen Subressourcen. Ein klassischer RPC-Aufruf (z. B. GetThing) würde in diesem Anwendungskontext höchstwahrscheinlich ein Thing-Objekt inklusive aller Detaildaten liefern. Diese RESTful-API liefert als Ergebnis ein Thing mit Links zu den jeweiligen Repräsentationen der Detaildaten (Sensoren, Aktuatoren, Regeln) zurück. Folglich lässt sich der Applikationszustand über diese Links überführen.

Obwohl gerade das Konzept von Hypermedia maßgeblich für dynamische, lose gekoppelte Systeme/Applikationen verantwortlich ist, ist es sehr oft falsch verstanden oder missinterpretiert worden. Aus diesem Grund schrieb Roy Fielding 2008 auch einen berühmten Blogartikel, in dem er explizit auf die Wichtigkeit von Hypertext hinweist:

In other words, if the engine of application state (and hence the

> *API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. [20]*

Viele Systeme, unabhängig davon ob sie behaupten RESTful zu sein, basieren auf einem impliziten Kontrollfluss, der dieselbe Charaktertistik wie traditionelle Webservices, die einem Remote Procedure Call gleichen (z. B. SOAP), aufweist [39].

6.2.3 REST vs. SOAP/WS-*

Wie schon vielfach angeschnitten, unterscheidet sich REST als Architekturstil wesentlich von den klassischen Webservices im RPC-Stil. Das Simple Object Access Protocol (SOAP), als XML-basiertes RPC-Protokoll, ist ein typisches Beispiel für klassische, schwergewichtige Webservices, deren Zweck die Abstraktion der Unterschiede eines lokalen und eines entfernten Prozeduraufrufes sind [39]. Typischerweise sind RPC-Protokolle stark funktionsorientiert und bieten über eine Interface Description Language (IDL) - WSDL im Fall von SOAP/WS-*[6] - die Möglichkeit, applikationsspezifische Details basierend auf standardisierten Kommunikationsprotokollen - in diesem Fall HTTP(S) - zu spezifizieren. Folglich ergibt sich eine starke Kopplung zwischen den Schnittstellen von Client und Server, die auch nur eine gleichzeitige Weiterentwicklung beider Komponenten erlaubt.

Tabelle 6.2 veranschaulicht eine zusammengefasste Gegenüberstellung der Grundausrichtung von REST und SOAP/WS-*. Während SOAP/WS-* als Middleware das Web nur als Transportmedium benützt und zu einem stark gekoppelten System führt, verfolgt REST mit seiner ressourcenorientierten Architektur eine direkte Integration in das Web und somit einen gegenteiligen Ansatz [61]. Der große Vorteil von SOAP/WS-* liegt in den unterschiedlichsten Möglichkeiten durch die vielfältigen Standards und Protokolle. Deshalb gibt es auch zahlreiche Möglichkeiten hinsichtlich Sicherheitsmaßnahmen. Die große Stärke von REST liegt in der Web-Integration und der Skalierbarkeit, die durch die REST-Konzepte (siehe 6.2.2) ermöglicht wird. Für eine detailliertere Betrachtung und einen umfassenden Vergleich der Konzepte und Technologien sei an dieser Stelle an ein Paper von Pautasso et al. [47] verwiesen.

6.2.4 REST-Maturity-Model

Viele der heute verfügbaren RESTful-Webservices verletzen ein oder mehrere REST-Konzepte. REST hat sich als Marketingbegriff etabliert, jede HTTP-basierte API wird oft fälschlicherweise als RESTful bezeichnet [50]. Einerseits ist dies auf die grundsätzlich relativ einfache Handhabung und die rasche Einarbeitung zurückzuführen. Andererseits ergibt sich aufgrund

[6] https://www.oasis-open.org/standards

Kriterium	REST	SOAP und WS-
Typ	Architekturstil	Protokoll(e)
Fokus	Ressourcen	Methoden
Ausrichtung	datenorientiert	funktionsorientiert
Kopplung	lose	stark
HTTP-Nutzung	Applikationsprotokoll	Transportprotokoll
Web-Integration	+++	+
Sicherheitsaspekte	+	+++
Skalierbarkeit	+++	+

Tabelle 6.2: REST vs. SOAP (angelehnt an [39, 47, 50])

dessen, dass keine Standards und Protokolle vordefiniert sind und der damit einhergehenden Freiheiten ein Wildwuchs an unterschiedlichen Webservices mit großem Abweichungspotential. Insbesondere das Konzept der Verlinkung von Ressourcen wurde von vielen Entwicklern nicht umgesetzt bzw. ignoriert. Aus diesem Grund hat Leonard Richardson [49] im Zuge eines Vortrages auf einer Konferenz (QCon) ein Reifegradmodell für REST vorgestellt. Abbildung 6.2 veranschaulicht die folgenden vier Stufen hin zu einer RESTful-Architektur [23, 49]:

1. *Stufe 0 - Plain Old XML* Der Ausgangspunkt ist im Grunde das Gegenteil von REST. SOAP basierte Webservices nutzen HTTP als Transportsystem für Interaktionen in verteilten Systemen ohne die eigentlichen Mechanismen des Webs auszunutzen. HTTP wird nur als Nachrichtentunnel für eigene Interaktionsmechanismen benutzt - ein klassischer RPC-Stil. Es gibt genau eine URI, die durch den SOAP-Endpunkt definiert und der über einen HTTP-POST Request aufgerufen wird. Hinter diesem Endpunkt verbergen sich eine Fülle an selbst implementierten Prozeduren. Der große Nachteil ist laut Richardson die Komplexität, die sich hinter diesem einen Endpunkt verbirgt.
2. *Stufe 1 - Ressourcen* Viele Webservices namhafter Hersteller (z. B. Amazon, Flickr, del.icio.us) waren im Jahr 2008 der Stufe 1 zuzuordnen, obwohl sie als RESTful bezeichnet wurden. Der Unterschied zu Stufe 0 liegt in der Einführung und Benutzung von Ressourcen. Folglich gibt es auf Stufe 1 für jede Ressource eine eigene URI. An-

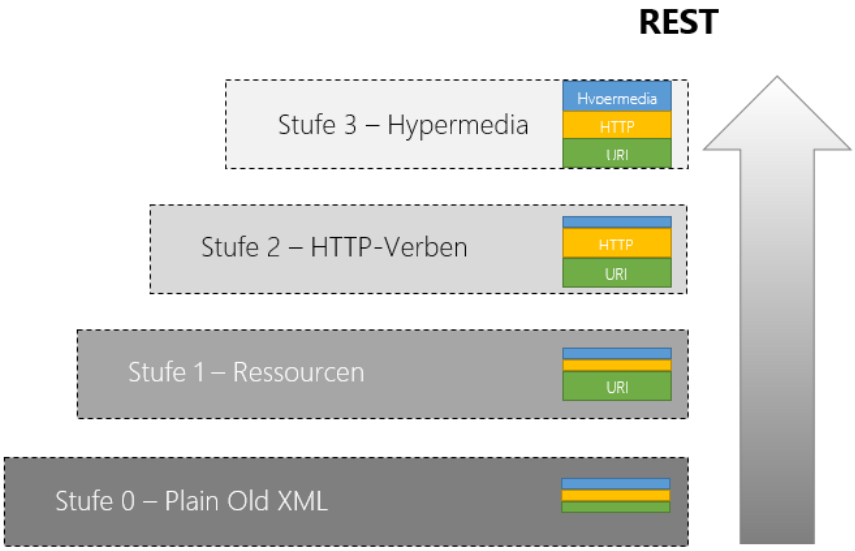

Abbildung 6.2: REST-Maturity-Model (angelehnt an [23])

statt alle Requests an einen einzelnen Endpunkt zu richten, werden Ressourcen individuell angesprochen. Dennoch wird auf Stufe 1 nur eine einzige HTTP-Methode verwendet. Obwohl Services der Stufe 1 meist sehr praktisch und beliebt sind (Richardson führt dies darauf zurück, dass URIs bei Entwicklern sehr beliebt sind), kam es aufgrund der Verwendung einer einzelnen HTTP-Methode für unterschiedliche Zwecke sehr oft vor, dass Operationen ausgeführt wurden, die nicht intendiert waren. So war es beispielsweise sehr leicht möglich, irrtümlich Dinge zu löschen.

3. *Stufe 2 - HTTP-Verben* Die zweite Stufe adressiert das Problem einer einzigen HTTP-Methode und nützt HTTP nicht nur als Transportsystem, sondern erstmals als Applikationsprotokoll. Es kann somit die Semantik der einzelnen HTTP-Methoden ausgenutzt und auf einem standardisierten Interface aufgebaut werden. Auf dieser Stufe wird klar zwischen GET, POST, PUT, DELETE, OPTIONS, etc. unterschieden. Eine Vielzahl an heute verfügbaren REST-basierten Webservices kann laut diesem Modell in dieser Stufe eingeordnet werden.
4. *Stufe 3 - HATEOAS* Die höchste Stufe nützt Hypertext um Clientapplikationen alle möglichen Zustandsübergänge anzubieten. Ressourcen auf Stufe 3 beschreiben ihre eigenen Fähigkeiten und bieten Verlinkungen zu anderen Ressourcen an, um einen Zustandswechsel zu ermöglichen. Diese Stufe ist am schwersten zu erreichen, da das Konzept von Hypermedia kompliziert und der Nutzen für einen Webservice-Entwickler nicht direkt ersichtlich ist.

6.3 REST im Web-of-Things-Umfeld

6.3.1 Relevanz von REST

Guinard & Trifa schlagen 2009 erstmals in ihrem Paper [31] eine konkrete Umsetzung einer Web-of-Things-Umgebung mittels einer REST-basierten Architektur vor. Sie begründen diese Entscheidung mit den vielen Vorteilen, die REST im Vergleich zu einer schwergewichtigen SOAP/WS-* Lösung mit sich bringt. WS-* Webservices basieren hauptsächlich auf zwei XML Formalismen: WSDL und SOAP. Darüber hinaus kommen noch einige weitere Standards zum Einsatz (WS-Adressing, WS-Security, WS-Discovery, etc.). Viele wissenschaftliche Projekte untersuchten und erweiterten laut Guinard [29] WS-* Szenarien, um eine Integration von Smart Things mit Applikationen zu ermöglichen. Obwohl dieser Ansatz eine Verbesserung im Vergleich zu den vor allem im Internet-of-Things üblichen proprietären Protokollen darstellte, kamen auch zahlreiche Nachteile ans Tageslicht. Erstens sind WS-* Standards sehr schwergewichtig und benötigen folglich viele CPU-Ressourcen, Hauptspeicher sowie Bandbreite. Dieser Umstand ist laut Guinard [29] vor allem bei ressourcenarmen Geräten, wie es Smart Things in der Regel sind, problematisch. Darüber hinaus benutzen WS-* Webservices HTTP lediglich als Transportprotokoll. Dies erschwert die Integration und Benutzung von Geräten mit dem World Wide Web (WWW) [31]. Weiters wird auch das ultimative Ziel von Web-of-Things, nämlich ein "loosely coupled ecosystem of services for smart things" [29] zu erschaffen, mit SOAP nicht erreicht. Wilde [61] argumentiert, dass die Eintrittsbarriere, um mit Ressourcen des Internet-of-Things zu kommunizieren sehr niedrig sein sollte. Dieser Umstand ist aufgrund des komplexen Technologiestacks bei einer RPC-orientierten SOAP/WSDL Architektur nicht gegeben.

Im Gegensatz zu WS-*/SOAP bietet REST die Möglichkeit Ressourcen direkt in das Web zu integrieren. Darüber hinaus bietet ein REST-basierter Ansatz die gewünschte lose Kopplung. Pautasso & Wilde [46] haben in ihrem Paper die Facetten von loser Kopplung systematisch untersucht und sind zum Schluss gekommen, dass ein RESTful-Ansatz im Vergleich zu einem klassischem RPC-Ansatz (über HTTP oder WS-* als Enterprise Service Bus) in jeder der zwölf Facetten eine losere oder zumindest gleichwertige Kopplung aufweist. Wie in Abbildung 6.3 dargestellt, können unterschiedlichste Applikationen je nach Einsatzszenario ohne jegliche Middleware direkt auf ein und dieselben Ressourcen zugreifen [61].

Guinard & Trifa [29, 31, 32, 58] integrierten Smart Things über REST ins Web, indem sie Things mit integrierten Webservern ausstatten. Folglich wird jedes Ding und seine inhärenten Ressourcen eindeutig adressiert und eine Möglichkeit geschaffen, die physische mit der virtuellen Welt zu vernetzen. Das Web-of-Things nützt HTTP als Applikationsprotokoll und schafft eine Entkopplung der Services von ihrer Präsentation (durch Repräsentation von

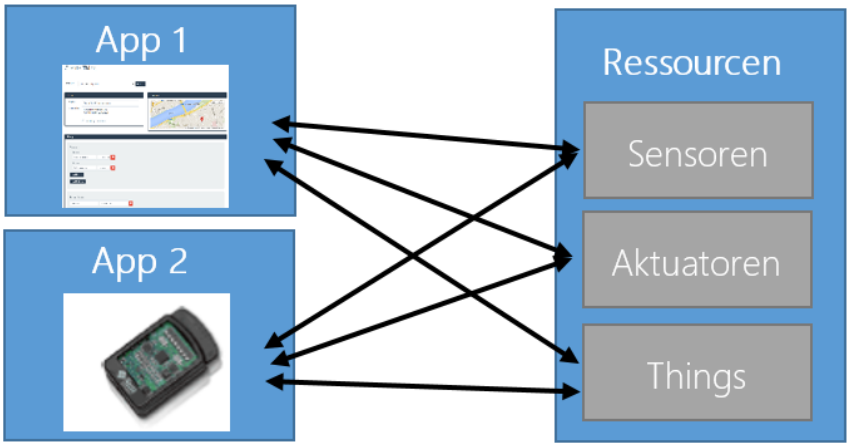

Abbildung 6.3: Zugriff auf Ressourcen in REST (angelehnt an [61])

Ressourcen) und bietet den Clients die Möglichkeit, das geeignetste Format als Medientyp auszuwählen. Darüber hinaus können auch Techniken zur Syndizierung (z. B. Atom) angewandt werden, um über das klassiche "Client-Pull" Modell von HTTP hinaus auch Push-Interaktionen mit Smart Things zu ermöglichen [29].

Zusammengefasst bietet REST aufgrund seiner Architektur folgende Vorteile [31, 32]:

- *Lose Kopplung:* Durch die lose Kopplung der Services ist eine einfache und effiziente Wiederverwendung sichergestellt
- *Repräsentationen:* Durch die Trennung von Ressourcen und deren Repräsentationen wird ermöglicht, dass der Client jenes Format der Repräsentation wählt, welches für die aktuelle Interaktion am geeignetsten erscheint
- *Verlinkung der Ressourcen:* Durch die eindeutige Adressierung jeder Ressource über eine URI lassen sich die Ressourcen auch einfach miteinander verlinken
- *Einheitliches Interface:* Durch die Fokussierung auf wenige HTTP-Methoden abstrahieren die Services von ihren jeweiligen applikationsspezifischen Semantiken und erlauben einen einfachen Zugriff auf Ressourcen.

6.3.2 RESTful Things: Eine ressourcenorientierte Architektur für smarte Dinge

Zur Realisierung des Web-of-Things müssen Smart Things in die Architektur des Webs integriert werden. Guinard [29] unterteilt diesen Prozess in die folgenden vier Schritte, die nachfolgend näher beschrieben werden: 1) Design

der Ressourcen, 2) Design der Repräsentationen, 3) Design des Interfaces und 4) Entwurf einer Implementierungsstrategie.

Design der Ressourcen

Der erste Prozessschritt beschäftigt sich mit der Modellierung der Funktionalität von Things als verlinkte Ressourcen. Grundsätzlich gibt es im Web-of-Things unterschiedliche Level von Ressourcen. Während manche Ressourcen einem physischen Objekt zugeordnet werden können, existieren andere Ressourcen nur virtuell. Die Modellierung der Ressourcen erfolgt in einer hierarchischen, verlinkten Struktur (Resource Oriented Architecture (ROA)), welche ein möglichst realistisches Modell der tatsächlichen physischen Struktur abbilden sollte.

Das Ressourcendesign beginnt mit der Identifizierung von Ressourcen. Es empfiehlt sich das Konzept der schrittweisen Verfeinerung anzuwenden, um ein Thing systematisch in kleinere Komponenten (=Ressourcen) zu zerlegen (Top-Down). Als Resultat dieser Zerlegung ergibt sich eine Struktur in Baumform. Jeder Ressource ist nun eine eindeutige URI zuzuweisen. Da es keine strikten Regeln für die Benennung von Ressourcen gibt, empfiehlt Guinard zumindest die folgenden Richtlinien:

- *Aussagekräftige Namen:* Nachdem der Ressourcenname in der URI aufscheint, sollte er möglichst kurz, prägnant und aussagekräftig sein.
- *Pluralisierung:* Aggregierte Ressourcen sollten immer im Plural angegeben werden. Falls ein Smart Thing beispielsweise mehrere Sensoren hat, so sollte es eine übergeordnete Ressource “Sensoren” geben, die eine Liste von Sensoren inklusive Hyperlinks zurückliefert.

Nach erfolgter Identifizierung und Strukturierung ist auch eine Verlinkung der Ressourcen erforderlich, um die REST-Konzepte (insbesondere HATEOAS - siehe Abschnitt 6.2.2) umzusetzen. Links sind in einer ressourcenorientierten Architektur sehr wichtig, da sie dem Client die Möglichkeit bieten, verwandte Ressourcen über Links in deren Repräsentationen dynamisch zu konsumieren.

Design der Repräsentationen

Ressourcen sind abstrake Entitäten, die nicht an eine spezifische Repräsentation gebunden sind. Folglich können verschiedenste Formate zur Repräsentation der Services eines Things verwendet werden. Während das Web selbst HTML als Medientyp (MIME-Type) verwendet, werden für Maschine-zu-Maschine (M2M) Kommunikationen andere wie beispielsweise XML oder JSON verwendet. Guinard empfiehlt die Verwendung von JSON-Repräsentationen, die einerseits direkt als Javascript-Objekte geparst werden können und andererseits eine ideale Möglichkeit für die Integration in Web-Mashups bieten.

Die Wahl des Medientyps hat auch auf die Verlinkung von Ressourcen große Auswirkung. Wenn ein Client eine HTML-Repräsentation lädt, ist die Verlinkung durch den Standardmechanismus von HTML zur Spezifizierung von Hyperlinks automatisch gegeben. Andere Formate, wie beispielsweise JSON, bieten aber keine Standards zur Spezifikation von Links.

Design des Interfaces

Das einheitliche Interface REST-basierter Webservices ist durch die Identifikation von Ressourcen und durch das HTTP-Protokoll definiert. Für jede Ressource sind die erlaubten Operationen (GET, PUT, POST, DELETE), die unterstützten Medientypen (content-negotiation) und die verwendeten HTTP-Statuscodes zu spezifizieren. Während GET vor allem für das Abrufen von Sensordaten verwendet wird, dient PUT der Ansteuerung von Aktuatoren.

Entwurf einer Implementierungsstrategie

Der letzte Schritt ist der Entwurf und die Umsetzung einer Strategie, um Smart Things in das Web zu integrieren. Die unterschiedlichen Strategien für die Implementierung und das Deployment der Things werden im nächsten Abschnitt detailliert diskutiert.

6.3.3 Web-of-Things Deployment Strategien

Grundsätzlich kann man zwischen zwei unterschiedlichen Deployment-Strategien im Web-of-Things unterscheiden: die native Integration von Things in das Web und die Konsolidierung über Web-of-Things-Hubs.

Native Integration verteilter, REST-basierter Things

Wie bereits in Abschnitt 2.2.2 erwähnt, basiert das Web-of-Things-Architekturmodell von Dominik Guinard [29] auf der Annahme, dass Smart Things trotz geringer Ressourcen in der Lage sind TCP/IP und HTTP 1.1 Protokolle zu implementieren. Aus diesem Grund können Smart Things selbst REST-basierte Webservices anbieten, die von verschiedenen Applikationen konsumiert werden können. Abbildung 6.4 verdeutlicht dieses Szenario. Neben der Device Accessiblity, die eine direkte Integration der Dinge in das Web erlaubt, muss bei diesem Ansatz aber auch eine geeignete Infrastruktur zum Suchen (Findability) der Things existieren, da die meisten Anwendungen die Daten vieler Dinge benötigen. Clients können (Sensor-)Daten entweder in Form eines Pull-basierten Ansatzes abfragen oder, falls unterstützt, über einen Publish-Subscribe Mechanismus über Datenänderungen informiert werden.

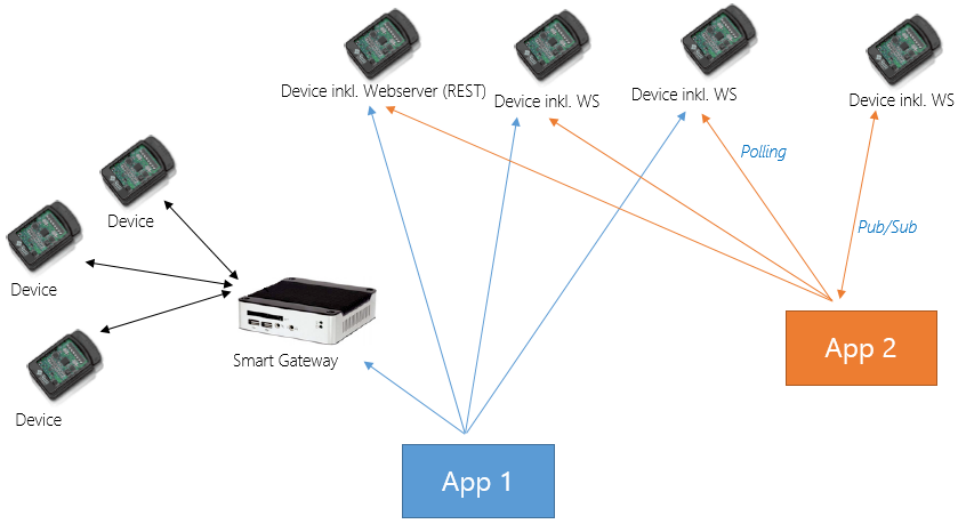

Abbildung 6.4: Direkter Zugriff auf die native REST-API von Things

Die native Integration von Smart Things ist mit einigen Nachteilen bzw. Einschränkungen behaftet. Obwohl viele "embedded devices" in Puncto Performance einen (zumindest abgespeckten) Webserver hosten können, ist die Anzahl an Clients, die gleichzeitig bedient werden können, im Vergleich zu einem herkömmlichen Webserver stark eingeschränkt. Folglich ist diese Strategie nur für jene Szenarien zu empfehlen, die nur eine überschaubare Anzahl von Clients und Zugriffen erfordern. Eine weitere Herausforderung stellt die Suche nach geeigneten Geräten dar. Auch wenn Guinard [29] ein Metadatenmodell zur Beschreibung von Things vorschlägt, welches für eine semantische Suche nach Smart Things benützt werden kann, bedarf gerade dieser Aspekt aufgrund fehlender Standards noch intensiver Forschungsarbeit. Darüber hinaus ist gerade in Enterprise-Szenarien oft nicht gewünscht, dass jegliche Daten öffentlich frei verfügbar sind. Sicherheitsstandards und -algorithmen, wie beispielsweise asymmetrische Verschlüsselungsverfahren (z. B. RSA), haben aber vielfach höhere Ressourcenanforderungen, die von Smart Things nicht oder nur teilweise erfüllt werden können.

Die Rolle von Smart Gateways

Eine Möglichkeit um die Limitierungen eines nativen Deployments zu umgehen, ist der Einsatz von sogenannten Smart Gateways. Smart Gateways funktionieren ähnlich wie Reverse-Proxys, indem sie Requests aus dem Internet entgegennehmen, welche anschließend in einem internen Netzwerk unter Benützung von geeigneten Low-Level-Protokollen verarbeitet werden [32]. Smart Gateways abstrahieren also proprietäre Kommunikationsprotokolle und proprietäre APIs von eingebetteten Systemen und bieten deren

Funktionalität über eine REST-basierte Schnittstelle an [31]. Der Vorteil liegt darin, dass über einen Smart Gateway somit auch jene Things ins Web integriert werden können, die keine Unterstützung für IP und HTTP(S) anbieten können. Ein Smart Gateway kann zudem bereit Daten mehrerer Dinge orchestrieren und zusätzliche Funktionalitäten (Mashups, etc.) anbieten [31]. Wie in Abbildung 6.4 ersichtlich, ist auch eine Kombination von Smart Gateways und nativem Deployment möglich.

Web-of-Things Hub

Einen völlig anderen Ansatz im Vergleich zur nativen Integration von Dingen stellen Web-of-Things Hubs dar. Im Zentrum dieser Strategie steht nicht die direkte Verknüpfung von Things und Applikationen, sondern eine Web-of-Things-Plattform als zentraler Datenhub [6]. Abbildung 6.5 illustriert diese Architektur: Smart Things liefern entweder direkt oder über Smart Gateways ihre Sensordaten in Echtzeit an die Plattform. Der Hub persistiert die Daten und stellt sie über REST-basierte Services zur Verfügung. Diese Plattformen dienen also nicht nur als zentraler Datenstorage für Dinge, sondern auch als zentraler Zugriffspunkt für unterschiedliche Anwendungen [6]. Der große Vorteil liegt in den umfassenden Skalierungsmöglichkeiten der Plattform und in der Tatsache, dass Daten entsprechend aufbereitet werden können. Es bietet sich zudem die Möglichkeit auf historische Daten zuzugreifen.

Die Anlieferung der Daten kann über verschiedenste Möglichkeiten erfolgen. Entweder liefern Things direkt oder über einen Smart Gateway proaktiv Daten (Push). In diesem Fall benötigen die Dinge keine eigenen Webserver und folglich auch keine eigene REST-API, sondern es genügt eine Möglichkeit zur Erzeugung von HTTP-Requests. Der Nachteil dieser Variante ist die Kopplung zwischen Thing und Hub, da die Adresse und Struktur der Aufrufe (URI) vordefiniert sein muss. Eine weitere Variante wäre, dass die Plattform die Daten von den Things abholt (Pull) oder abonniert (Publish-Subscribe). Im Unterschied zum nativen Deployment kümmert sich in diesem Szenario aber die Plattform und nicht die Applikationen selbst um die Aggregation der Daten.

6.4 WoTCloud: REST-basierte Webservices

Basierend auf der umfassenden Einführung von REST und den Integrationsmöglichkeiten von Smart Things in das Web-of-Things fokussiert dieser Abschnitt die konkrete Implementierung und das Deployment der Webservices von WoTCloud. Ausgangspunkt für die Implementierung der Plattform sind die Anforderungen aus Abschnitt 5.2 und das konzeptuelle Modell aus Abschnitt 5.3.

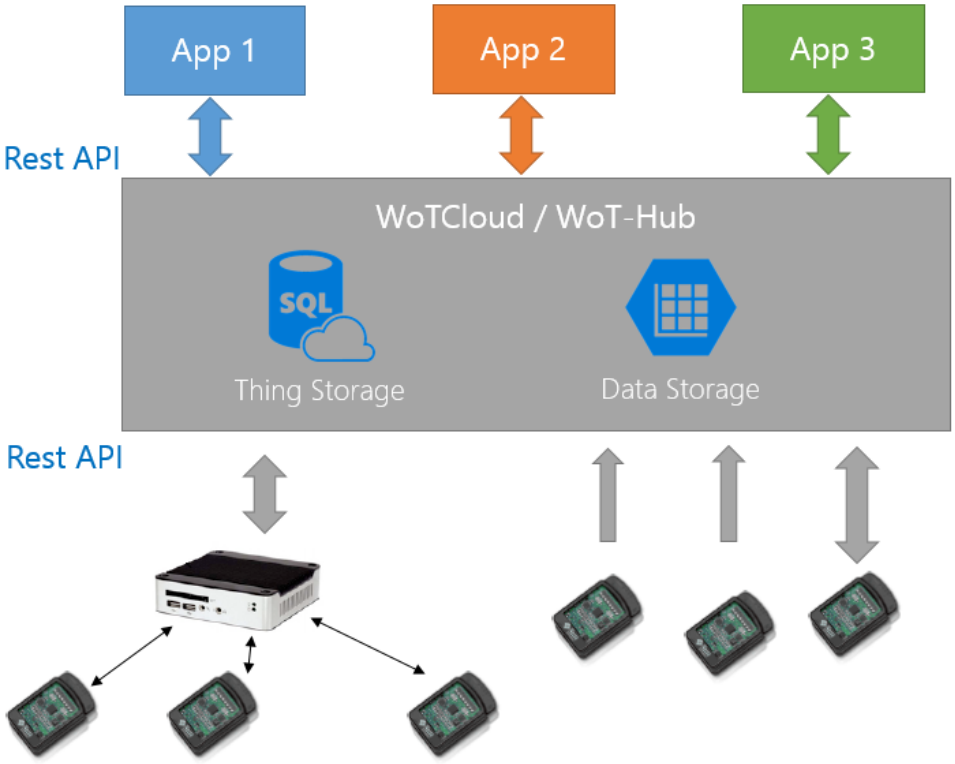

Abbildung 6.5: Web-of-Things Hub (angelehnt an [6])

6.4.1 Entwurf einer ressourcenorientierten Architektur

1. Design der Ressourcen Der erste Prozessschritt dient der Identifikation und Strukturierung der verlinkten Ressourcen. Die Autoren identifizierten basierend auf den Anforderungen für die Plattform folgende primäre Ressourcen: Tenants, Things, Templates, Sensors, Actuators und Rules. Aufgrund des Multi-Tenancy-Ansatzes stellt der Tenant den Wurzelknoten in der hierarchischen URI-Struktur dar. Wie in Abbildung 6.6 ersichtlich, verfügt jeder *Tenant* über die Subressourcen *Things* und *Templates*. Things und Templates sind sehr ähnlich und bestehen wiederum aus *Sensors*, *Actuators* und *Rules*. Der einzige Unterschied liegt in der Tatsache, dass Templates nur als Schablone dienen und keine Sensordaten enthalten können.

Grundsätzlich sind alle Ressourcen genau einem Tenant zugeordnet. Dieser Ansatz dient der strikten Trennung der Kundendaten und wird auch zur Autorisierung verwendet. Um die Isolation der Kundendaten auf Datenbankebene zu erreichen, gibt es verschiedenste Ansätze (eine Datenbank, eine Datenbank pro Kunde, Sharding, etc.). Eine ausführliche Erläuterung unterschiedlicher Multi-Tenancy-Ansätze und deren Umsetzung wird in der begleitenden Arbeit von Andreas Neuhauser durchgeführt [44]. Der Tenant

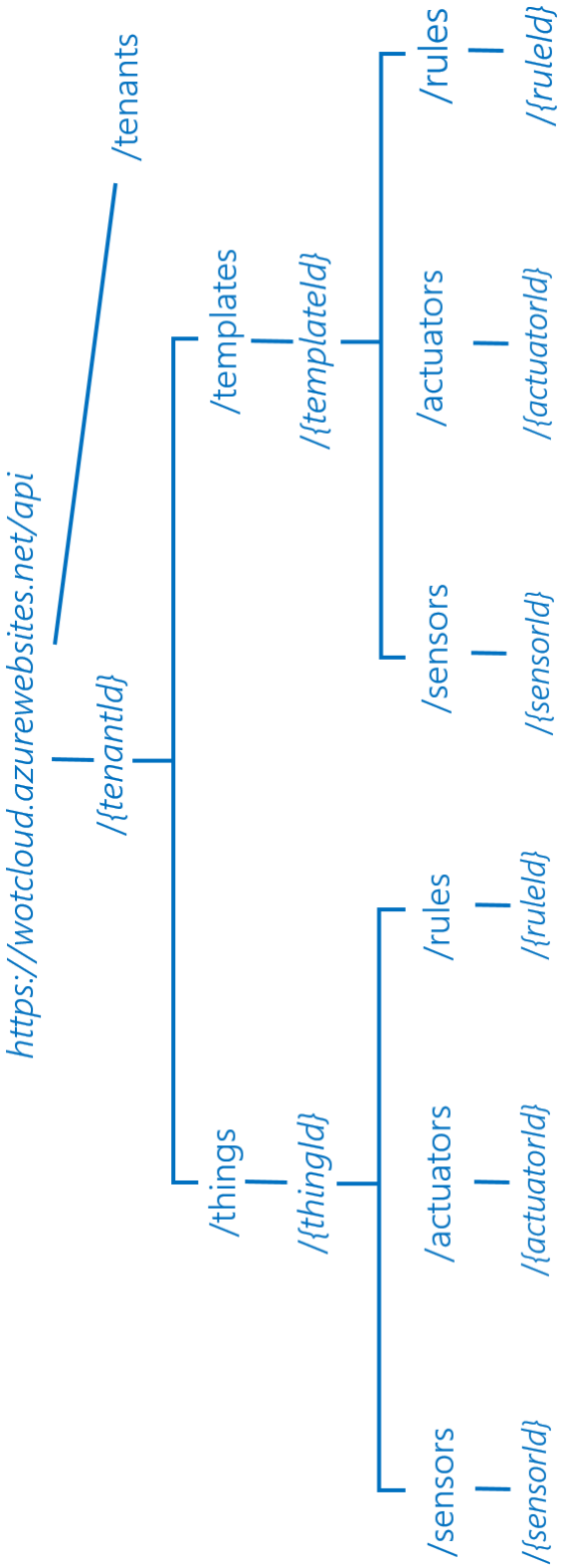

Abbildung 6.6: Design der Ressource-Oriented-Architecture von WoT-Cloud.

ist in der URI-Struktur die einzige Ausnahme hinsichtlich der Pluralisierung von Ressourcen. Es gibt also kein Präfix “tenants/” vor der jeweiligen Tenant-ID. Der Grund dafür ist die Tatsache, dass jeder Client normalerweise nur die Daten eines einzelnen Tenants konsumieren darf.

2. Design der Repräsentationen Beim Design der Repräsentationen geht es vor allem um die Entscheidung, welcher oder welche Medientypen vom Webservice unterstützt werden. Die Autoren entschieden sich aufgrund der weiten Verbreitung und Akzeptanz und dem im Vergleich zu XML geringeren Overhead für JSON. Der große Vorteil von JSON liegt darin, dass der auf HTML5 und AngularJS basierte Webclient der Plattform die JSON-Daten des Webservices nativ parsen und direkt im Objektmodell verwenden und an die UI-Komponenten binden kann.

Simple und komplexe Things Im konzeptuellen Modell (siehe Abschnitt 5.3) wird zwischen SimpleThings und ComplexThings unterschieden, wobei ComplexThings wiederum aus einer Menge von SimpleThings bestehen können. Aufgrund der hierarchischen Struktur von URIs ist in der vorgestellten ROA für WoTCloud keine Unterscheidung von Simple- und ComplexThings notwendig. Die Ressource */things/{thingId}* ist selbst inhärent hierarchisch. Ein Aufruf eines Things, egal ob simpel oder komplex, erfolgt immer über dessen eindeutige ID. Der Identifier von Things ist innerhalb eines Tenants eindeutig. Listing 6.5 verdeutlicht, dass die Unterscheidung von simplen und komplexen Dingen beim Aufruf nur über die ID gesteuert wird. Darüber hinaus wird auch die Verlinkung von Ressourcen über deren Repräsentation in JSON ersichtlich.

Verlinkung von Ressourcen Die Verlinkung der Ressourcen über Repräsentationen erfordert eine gewisse Standardisierung, sodass alle beteiligten Parteien (Client, Server, externe) dieses Konzept richtig verstehen und anwenden können. Pures JSON bietet keine Möglichkeit zur Definition einer entsprechenden Syntax und deren Semantik. Aus diesem Grund haben sich in den letzten Jahren einige Forschungsprojekte mit diesem Problem beschäftigt. Ein Ansatz, *JSON-LD*, der auch schon in einem W3C Standard resultiert, versucht sowohl JSON-basierte Webservices als auch verlinkte Daten zu integrieren [39]. Laut Lanthaler [39], der selbst als Experte an der Entwicklung des Standards beteiligt war, versucht JSON-LD neben der Integration von verlinkten Daten auch bestehende JSON-Dokumente so anzureichern, dass sie selbstbeschreibend sind. Darüber hinaus hat er in seiner Dissertation Hydra entwickelt. Die Grundidee von Hydra liegt in der Bereitstellung eines leichtgewichtigen Vokabulars, mit dem der Server in der Lage ist, dem Client gültige Zustandsübergänge anzuzeigen [39]. Der wesentliche Unterschied von Hydra im Vergleich zu Ressource Description Framework

Listing 6.5: Aufruf von simplen und komplexen Dingen (HTTP-Requests & HTTP-Responses)

```
GET http://wotcloud.azurewebsites.net/api/1/things/1 HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content_Type: application/json
{
    "id": 109,
    "name": "Sielaff FK 230 - Engel1",
    "description": "Aufgestellt bei Kunde: Engel GmbH in Schwertberg",
    "latitude": 48.30694,
    "longitude": 14.28583,
    "links": [
        {
            "Rel": "self",
            "Href": "http://wotcloud.azurewebsites.net/api/1/things/1"
        },
        {
            "Rel": "child",
            "Href": "http://wotcloud.azurewebsites.net/api/1/things/2"
        },
        {
            "Rel": "child",
            "Href": "http://wotcloud.azurewebsites.net/api/1/things/3"
        },
        {
            "Rel": "child",
            "Href": "http://wotcloud.azurewebsites.net/api/1/things/4"
        }, ... ]
}
```

```
GET http://wotcloud.azurewebsites.net/api/1/things/2 HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content_Type: application/json
{
    "id": 2,
    "name": "Schacht 1",
    "description": "Cola",
    "latitude": null,
    "longitude": null,
    "links": [
        {
            "Rel": "self",
            "Href": "http://wotcloud.azurewebsites.net/api/1/things/2"
        },
        {
            "Rel": "sensors",
            "Href": "http://wotcloud.azurewebsites.net/api/1/things/2/sensors"
        }, ... ]
}
```

Listing 6.6: Design eines Webservices zur Anlage eines Things

```
1 /// <summary>
2 /// Creates a new thing
3 /// </summary>
4 /// <param name=tenantId>id of the tenant</param>
5 /// <param name=thingModel>thing to create</param>
6 /// <returns>Thing</returns>
7 [Route("api/{tenantId:int}/things")]
8 [HttpPost]
9 [ValidateModel]
10 public HttpResponseMessage CreateThing(int tenantId, [FromBody]
       ThingModel thingModel)
11 {
12     ...
13     return Request.CreateResponse(HttpStatusCode.OK, thingId);
14 }
```

Schema (RDFS) liegt in der zusätzlichen Unterstützung von Hypermedia.

Leider existiert zu diesem Zeitpunkt keine Implementierung von JSON-LD bzw. Hydra für Microsoft's ASP.NET Web API. Aus diesem Grund wird für den Prototypen von WoTCloud vorerst auf eine standardisierte Verlinkung und Beschreibung von Ressourcen verzichtet, da eine eigenständige Implementierung von JSON-LD für .NET den Rahmen dieser Arbeit um ein Vielfaches übersteigen würde. Dennoch wird auf eine eigene, konsistente Beschreibung der Links gesetzt. Jeder Link in WoTCloud besteht aus einer Beschreibung der Beziehung *"Rel"* mit den möglichen Werten "self", "child", "next" für Elemente des gleichen Typs beziehungsweise "sensors", "actuators" und "rules" für den Verweis auf andere Ressourcentypen. Zusätzlich enthält jeder Link eine URI ("Href") zur entsprechenden Ressource (siehe Listings 6.4 und 6.5).

3. Design des Interfaces Die Service-Interfaces der ASP.NET Web API des Prototypen werden durch eine Mischung von Annotationen, Konventionen und Methodendeklarationen definiert (siehe Listing 6.6). Durch das Konzept des Attribute Routing werden Routen zu Action-Methoden definiert (siehe Zeile 7); genauso wie der Typ der HTTP-Methode (GET, PUT, POST, DELETE) - in diesem Fall [HttpPost] über Annotationen definiert wird (siehe Zeile 8). Geschwungene Klammern in der Route deuten auf einen Parameter hin, der auch zwingend in der Methodendeklaration existieren muss. Die Annotation vor dem Parameter thingModel in Zeile 10 des Listings 6.6 deklariert, dass dieser Parameter aus dem Body des HTTP-Requests ausgelesen wird. Ohne diese Annotation werden Parameter aus der URL des HTTP-Requests transferiert.

In dieser Art und Weise können beliebig viele Methoden definiert und zu sinnvollen Klassen (Controller) subsumiert werden. Die Autoren haben

für jede primäre Ressource einen eigenen Controller definiert, der auch die Möglichkeit zur Definition von Präfixen für Routen bietet. Abbildung 6.7 illustriert eine kurze Dokumentation aller Webservices von WoTCloud.

4. Entwurf einer Implementierungsstrategie Basierend auf dem in Kapitel 3 vorgestellten Szenario und den in Kapitel 5 abgeleiteten Anforderungen wurde schnell klar, dass ein Smart Vending Operator eine zentrale, geschlossene Plattform für die Verwaltung und den Betrieb seiner Automaten benötigt. Trotzdem bietet eine Web-of-Things-Plattform als Software-as-a-Service (SAAS)Lösung in der Cloud die Möglichkeit für den Betrieb für eine Vielzahl an Operatoren hochskaliert zu werden.

In unserem konkreten Szenario liefern Getränkeautomaten entweder direkt oder über einen Smart Gateway ihre Sensordaten über die REST-basierten Services an die Plattform. Die Voraussetzung dafür ist, dass der Automat zuvor in WoTCloud angelegt wurde und der Automat die URI-Struktur der Plattform kennt. Die Plattform persistiert alle Daten und stellt den zentralen Bezugspunkt sämtlicher Applikationen dar. Nebem dem Webclient von WoTCloud, der ebenfalls die REST-basierte API konsumiert, können weitere zukünftige Applikationen, wie beispielsweise eine Smartphone App für Kunden zur Lokation von Getränkeautomaten oder auch das Warenwirtschaftssystem des Operators, die Daten der Plattform nutzen.

6.4.2 WoTCloud im Kontext des REST-Maturity-Models

Der Reifegrad der Webservices von WoTCloud ist hinsichtlich des vorgestellten REST-Maturity-Modells aus Abschnitt 6.2.4 zwischen Stufe 2 und 3 einzuordnen. Die ressourcenorientierte Architektur bietet eine eindeutige Adressierung über URLs (Stufe 1) und je nach Anwendungszweck (Anlegen, Abfragen, Ändern, Löschen) entsprechende HTTP-Methoden für jede Ressource (Stufe 2). Darüber hinaus wird auch der Verlinkung von Ressourcen Rechnung getragen. Aufgrund der fehlenden Standardisierung (JSON-LD o.ä.) sind die Autoren der Meinung HATEOAS nicht vollständig umgesetzt zu haben. Folglich können diese Services korrekterweise zwar als REST-basiert (oder “RESTified”), nicht jedoch als RESTful bezeichnet werden.

Der Webclient nutzt darüber hinaus noch einige zusätzliche Ressourcen, die speziell dafür geschaffen wurden, um die Anzahl der Transaktionen über das Internet und den damit verbundenen Overhead zu reduzieren und die Performance zu steigern. So gibt es beispielsweise die Möglichkeit zu Things oder Templates eine Detailressource abzufragen, die alle relevanten Subressourcen (Sub-Things, Sensoren, Aktuatoren, Regeln) gruppiert aufbereitet zurückliefert. Würde es diese Ressource nicht geben, müssten vom Client anstatt eines einzigen Requests bei einem Automaten mit 10 Schächten ungefähr 45 einzelne Requests abgesetzt werden. Aus diesem Grund haben sich die Autoren für eine Mischung aus Services, die zu 100 Prozent den

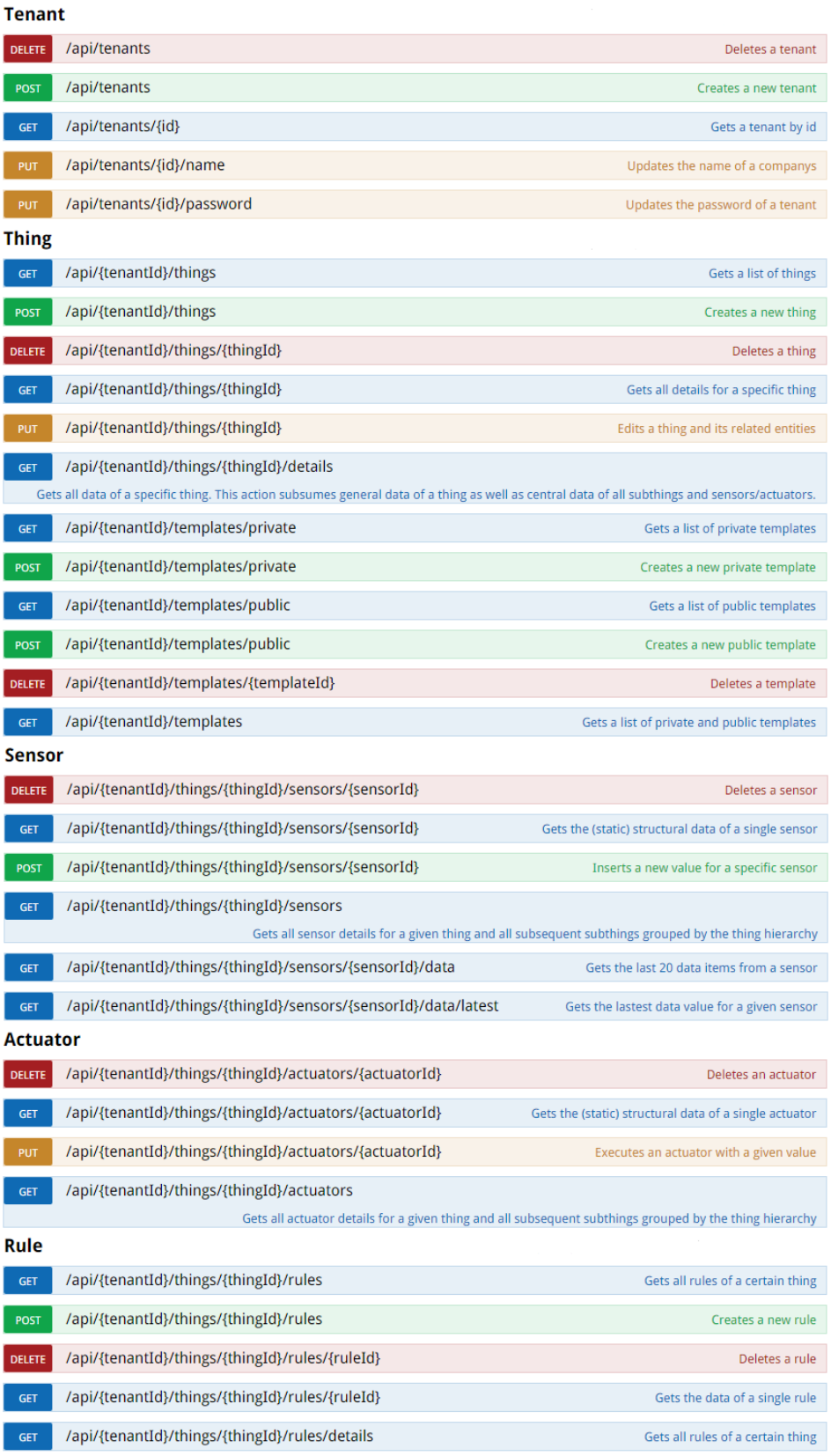

Abbildung 6.7: Dokumentation der WoTCloud-Webservices.

Listing 6.7: Erzeugung eines OAuth2 Bearer Access-Token in WoTCloud (HTTP-Request & HTTP-Response)

```
POST http://wotcloud.azurewebsites.net/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Accept: application/json
{
    grant_type=password&username=wagner&password=test
}
```

```
HTTP/1.1 200 OK
Content_Type: application/json
{
    "access_token": "FvFO_UbTc_uXHbL4T1zFSchNeJD41cCZRWET0n9nQjUHbuqzaS
    ehtJ_mh6B42rRMs_uo6tDJscSSTJlXwgkfP0hfdyABj-1BeXiCNnABsLXs8K5T8s2OT
    YpO4L9F0vnK1aXYShXLvq66zbrD6E7KhuPaxfjHcpbEq4TSB33TvY33mo9eeD40c5S4
    PRJS-kq5o41HxT2lQLGtgvJ6yVS1FKSgKp8jIiQZIRj9VuhYlo",
    "token_type": "bearer",
    "expires_in": 1799,
    "tenant": "1",
    ".issued": "Sun, 12 Jul 2015 15:28:06 GMT",
    ".expires": "Sun, 12 Jul 2015 15:58:06 GMT"
}
```

REST-Konzepten entsprechen und einigen zusätzlichen Services, die zwar auch Stufe 2 des REST-Maturiy Modells entsprechen, aber trotzdem eher einer RPC-Semantik gleichen, entschieden. Der Nachteil dieses Ansatzes ist, dass die Kopplung zwischen Weblclient und dem Webserver der Plattform dadurch erhöht wird. Sämtliche Drittanwendungen können jedoch auch jetzt schon alle primären Ressourcen des Prototypen nutzen ohne diese zusätzlichen Services zu benötigen.

6.4.3 Sicherheit

Ein wesentlicher und bisher nicht beachteter Aspekt ist die Absicherung der Web-of-Things-Plattform. Für diesen Zweck wird neben HTTPS auch OAuth2[7] eingesetzt. Bevor eine Applikation Webservices konsumieren kann, muss sie sich einen gültigen Access-Token vom OAuth-Endpunkt (z. B. "https://wotcloud.azurewebsites.net/token") besorgen. In unserem Fall ist der Authorisierungsserver auch gleichzeitig der Webserver der Plattform. Um diesen Access-Token zu erlangen, sind zwingend ein gültiger Benutzername und ein gültiges Passwort eines Tenants anzugeben. Der ausgestellte Access-Token ist nur für den jeweiligen Tenant und in einem definierten Zeitinvertall (z. B. 30 Minuten) gültig. Dazu hält jeder Token, wie in Listing 6.7 dargestellt, alle benötigten Informationen.

[7] http://tools.ietf.org/html/rfc6749

Eine token-basierte Authentifizierung bietet im Vergleich zu Cookies folgende Vorteile [38]:

1. *Skalierbarkeit des Servers:* Jeder Token enthält alle Informationen, die der Server zur Bearbeitung benötigt. Folglich ist serverseitig keine Speicherung von Sessiondaten notwendig.
2. *Lose Kopplung:* Der Client ist an keine spezifischen Authentifizierungsmechanismen gekoppelt, da der Token vom Server generiert und validiert wird. Die Aufgabe des Clients ist nur die transiente Speicherung des Tokens sowie jedem Request den Token anzufügen.
3. *Unterstützung mobiler Geräte:* Während Cookies in browserbasierten Ansätzen kein Problem darstellen, ist die Unterstützung von Cookies in nativen Apps (Android, iOS, Windows Phone) eine schwierige Aufgabe. Der token-basierte Ansatz ist im Gegensatz dazu weit einfacher umzusetzen.

Im Zuge der prototypischen Implementierung ist derzeit auch für Things die Anforderung eines Access-Tokens Voraussetzung für die Kommunikation mit der Webplattform. Zukünftig sollen aber noch zusätzliche Mechanismen implementiert werden, die keinen Benutzernamen und kein Passwort erfordern. Denkbar wäre beispielsweise die (einmalige) Generierung von statischen Access-Tokens pro Tenant oder Thing. Problematisch wird es, falls ein Getränkeautomat aufgrund mangelnder Ressourcen weder HTTPS noch token-basierte Zugriffskontrolle unterstützt. Hinsichtlich dieser Szenarien ist, wie auch generell im Internet-of-Things-Bereich, noch großes Potential für zukünftige Forschungsarbeit vorhanden[8] [58].

6.5 Simulator

Das grundlegende Ziel dieser Arbeit ist die Entwicklung und Implementierung eines Konzeptes zur Unterstützung von komplexen Dingen in einer Web-of-Things-Plattform. Auf Basis eines Smart Vending Szenarios wurde die Architektur und Umsetzung von WoTCloud vorgestellt. Da die tatsächliche Anbindung eines physischen Getränkeautomaten aufgrund der komplexen, hardwarenahen Protokolle und mangelnder Standardisierung in dieser Branche nie das Ziel war, die Autoren aber dennoch ein realistisches Set-Up erstellen wollten, haben sie zusätzlich einen Simulator entwickelt. Der Simulator soll alle grundlegenden Funktionalitäten eines Automaten bieten. Neben dem Verkaufsvorgang in einem spezifischen Schacht soll auch einem Nachfüllvorgang und einer Preisänderung Rechnung getragen werden.

Abbildung 6.8 illustriert das User Interface des Simulators, der als Webanwendung mittels ASP.NET MVC entwickelt wurde. Nach der Festlegung

[8] http://www.networkworld.com/article/2921004/internet-of-things/beware-the-ticking-internet-of-things-security-time-bomb.html

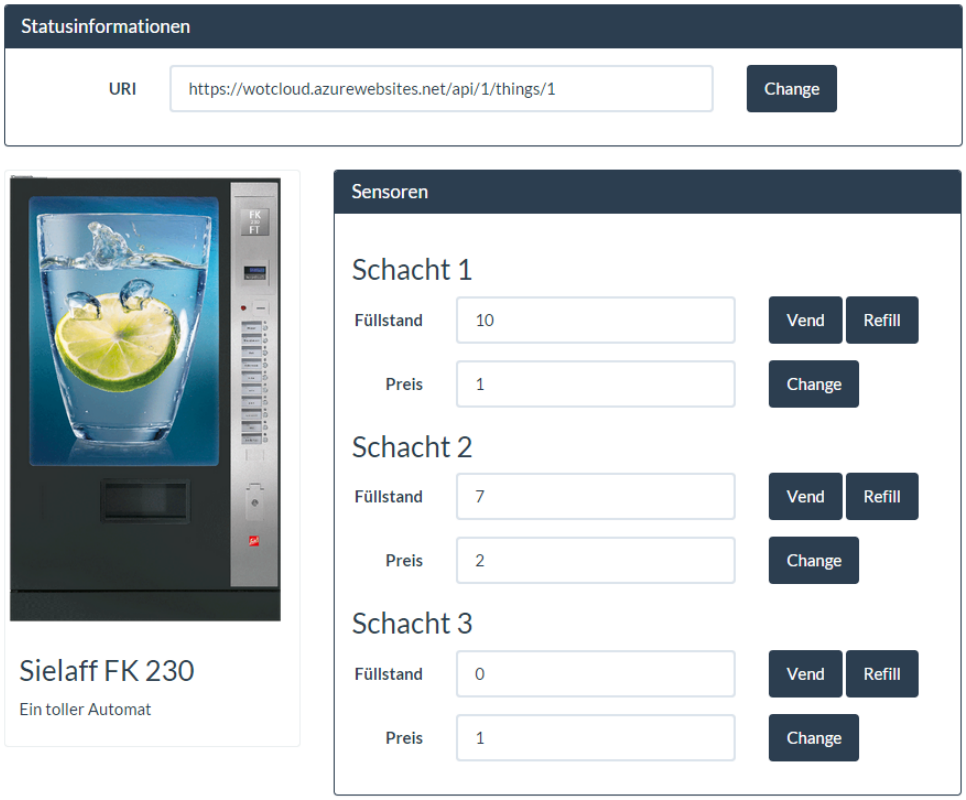

Abbildung 6.8: Simulation der Kommunikation eines Getränkeautomaten mit WoTCloud.

eines Getränkeautomaten durch eine eindeutig URL der Plattform (z. B. *https://wotcloud.azurewebsites.net/api/1/things/1*) werden dynamisch alle untergeordneten Schächte des Automaten geladen und die Füllstands- und Preissensoren ausgelesen. Wird nun ein Vorgang (Vend, Refill, Change) vom Benutzer ausgelöst, so sendet der Simulator einen entsprechenden HTTP-Request mit dem neuen Sensorwert an die Plattform. Umgekehrt bietet sich über Business Rules (siehe Kapitel 7) die Möglichkeit, den Aufruf von Aktuatoren zu simulieren. Beispielsweise kann über Regeln der Preis eines Produktes (über den Schacht) via Aktuatoren angepasst werden. Dazu ist im Aktuator, der von der Geschäftsregel aufgerufen wird die URL des Simulators festzulegen. Der Simulator wurde deshalb als Webanwendung realisiert, um auch den Aufruf von Aktuatoren (durch eine eindeutige URL) simulieren zu können.

6.6 Zusammenfassung

REST-basierte Webservices können als architektonische Grundlage des Web-of-Things angesehen werden. Aus diesem Grund wurden in diesem Kapitel die Entwicklung von REST als Architekturstil sowie die essentiellen Konzepte vorgestellt. Ein RESTful Webservice hat sich insbesondere an folgende Einschränkungen zu halten: 1) zustandslose Interaktionen, 2) einheitliches Interface, 3) Identifikation von Ressourcen, 4) Manipulation von Ressourcen durch Repräsentationen, 5) selbstbeschreibende Nachrichten und 6), hypermedia-as-the-engine-of-application-state (HATEOAS). Darüber hinaus wurden mit dem Richards Maturity Model vier Entwicklungsstufen hin zu einer RESTful-Architektur vorgestellt. Es wurde REST mit WS-*/SOAP verglichen und erläutert, warum gerade REST für das Web-of-Things von Bedeutung ist.

Basierend auf diesen Grundlagen wurde ein von Dominik Guinard vorgestelltes Modell zum Entwurf einer ressourcenorientierten Architektur für smarte Dinge vorgestellt und in einem konkreten Smart Vending Szenario angewandt. Es wurden sowohl die Ressourcen, Repräsentationen, Service-Interfaces als auch die Deployment-Strategie von WoTCloud systematisch vorgestellt und begründet. Anschließend wurde WoTCloud im REST Maturity Model eingeordnet und erste Security-Mechanismen des Prototypen diskutiert. Abschließend wurde ein webbasierter Getränkeautomaten-Simulator vorgestellt.

Kapitel 7

Business Rules für WoTCloud

Dieses Kapitel beschäftigt sich mit der Integration von Geschäftsregeln in die Web-of-Things-Plattform. Nach einer allgemeinen theoretischen Einführung zum Thema Geschäftsregeln und regelbasierter Systeme wird der konkrete Anwendungsfall der Plattform vorgestellt. Aufbauend darauf werden verschiedene Rule-Engines verglichen und evaluiert. Schlussendlich wird die konkrete Umsetzung und die Integration von Geschäftsregeln in WoTCloud vorgestellt.

7.1 Einführung

Der Einsatz von Regeln ist ein wichtiger Teil vieler IT-Systeme der heutigen Zeit. Viele Unternehmen nutzen deklarative Regeln um möglichst flexbile Anwendungen auf einem hohen Abstraktionslevel zu gestalten. Diese Regeln dienen unter anderem der Modellierung von Sicherheitsrichtlinien, der Steuerung von Geschäftslogik oder zur Schlussfolgerung im Semantic Web [9].

Allgemein betrachtet können Regeln als selbstständige Wissenseinheiten betracht werden, die abhängig von ihrem Einsatzzweck auch verschiedenste Formen der Schlussfolgerung involvieren. Laut Boley [9] ergeben sich vor allem die folgenden drei Einsatzgebiete:

- Sicherstellung der statischen und dynamischen Integrität durch Constraints
- Ableitungen/Schlussfolgerungen
- Reaktionen/reaktives Verhalten

Reaktives Verhalten im Web, also die Fähigkeit Ereignisse zu erkennen und in angemessener Zeit darauf zu reagieren, ist ein wichtiger Baustein zur Verbindung von passivem und dynamischem Web. Berstel et al.

[5] verstehen in diesem Zusammenhang einfache Datenquellen als passives Web und jene Datenquellen, welche mit reaktivem Verhalten angereichert sind, als dynamisches Web. Weiters ergäbe sich aus der allgemeinen Definition von reaktivem Verhalten ein vielfältiges Anwendungsgebiet: Neben Webanwendungen, die auf Benutzereingaben reagieren (z. B. E-Commerce-Plattformen), sind vor allem auch (verteilte) Web-Informationssysteme gemeint, die auf Updates in anderen Systemen oder anderen Lokationen im Web reagieren. Eine Web-of-Things-Plattform, welche auf Sensordaten von vielen verteilten Smart Things reagiert, wäre ein typischer Anwendungsfall für solch ein webbasiertes Informations- und Kommunikationssystem. Reaktives Verhalten kann vor allem durch reaktive Geschäftsregeln realisiert werden. Im Gegensatz zu einer reinen, hartverdrahteten Implementierung im Programmcode ergibt sich durch Geschäftsregeln eine höhere Abstraktion, eine bessere Anpassbarkeit durch deklarative Regeln und eine einfachere Wartbarkeit [5].

7.2 Theoretische Grundlagen

7.2.1 Definition

Die OMG definiert Geschäftsregeln (Business Rules) sehr allgemein als "any rule under business jurisdiction" [28]. Aus Anforderungssicht sollten diese Business Rules in einer deklarativen Form, wie beispielsweise in der Semantics of Business Vocabulary and Business Rules (SBVR)-Spezifikation, die zur Dokumentation von Geschäftsvokabular (Definitionen), Fakten (Verbindung) und Regeln dient, spezifiziert werden [15, 52]. Aus dieser grobgranularen Definition lassen sich eine Vielzahl von unterschiedlichen Einsatzszenarien für Geschäftsregeln ableiten. So können Regeln unter anderem als Einschränkung dienen (z. B. *nur jene Automaten mit mindestens 8 Schächten*), implizite Daten definieren (z. B. *Top-Kunden sind jene Kunden, die mindestens 20 Automaten gekauft haben*), die Integrität von Daten(-banken) sicherstellen oder dynamisches Verhalten spezifizieren (z. B. *erhöhe den Preis, wenn die Außentemperatur über 30 Grad liegt*) [9].

Taveter & Wagner definieren Business Rules wie folgt:

> *statements that express (certain parts of) a business policy, such as defining business terms, defining deontic assignments (of powers, rights and duties), and defining or constraining the operations of an enterprise, in a declarative manner. [57]*

Diese Definition berücksichtigt, dass Geschäftsregeln immer aufgrund (strategischer) Ziele abgeleitet werden und sowohl definieren als auch einschränken. So können Geschäftsregeln sowohl von außen vorgegeben werden (z. B. durch gesetzliche Vorgaben), als auch innerhalb einer Organisation

zur Erreichung bestimmter Ziele. Geschäftsregeln sind deklarativ: Sie beschreiben, *was* erreicht werden soll bzw. welcher Zustand gehalten werden soll, nicht jedoch das *wie*. Im Gegensatz zu imperativen bzw. objektorientierten Programmiersprachen und deren Bibliotheken bringt regelbasiertes Programmieren als deklarative Sprache eine fein-granularere Modularität und höhere Abstraktion mit sich [5]. Oft werden diese als Einschränkungen in der Form - *Wenn <Bedingung> Dann <Aktion>* - definiert [15]. Diese Form der natürlichen Sprache erleichtert es auch technisch unversierteren Benutzern Regeln in solchen Applikationen zu definieren, verstehen und zu warten [5]. Laut Charfi & Mezini [15] sind Geschäftsregeln vor allem in jenen Branchen nützlich, welche einen hohen Grad an Entscheidungen und Richtlinien aufweisen (z. B. Finanz- und Versicherungsbranche). Sie ermöglichen, dass gewisse Teile vom Domänen-Wissen unabhängig vom Rest der Anwendung spezifiziert, verwendet und aktualisiert werden können.

7.2.2 Vorteile

Die Verwendung von Business Rules in Anwendungen bringen folgende Vorteile mit sich [27]:

- *Flexibilität durch Regeln:* Die Art und Weise wie die Applikation arbeitet, kann einfach verändert und kontrolliert werden.
- *Verbesserung der Wartbarkeit*
- *Transparenz schaffen durch Unternehmensvokabular:* Regelbasierte Systeme erfordern die Definition eines klar verständlichen und eindeutigen Unternehmensvokabulars, welches auch in vielen anderen Bereichen wiederverwendet werden kann. Dies erhöht unter anderem auch das Verständnis zwischen Fachbereich und IT-Abteilung.
- *Effizienz durch Automation:* Eine Regel impliziert eine Automation gewisser Abläufe.
- *Reduktion der Eintrittsbarrieren:* Regeln sind für jeden (mit Dömänen-Wissen) leicht verständlich, da sie eine große Nähe zum menschlichen Denken aufweisen. Folglich kann auch der Fachbereich ohne IT-Kenntnisse Regeln definieren und warten.

7.2.3 Kategorisierung

In der Literatur gibt es viele unterschiedliche Ansätze zur Klassifizierung von Business Rules [34, 57]: Schrefl et al. [52] merken an, dass wissenschaftliche Publikationen grundsätzlichen zwischen statischen Geschäftsregeln, die ein Ausdruck invarianter Zustände zwischen Fakten und deren Beziehungen sind, und dynamischen Geschäftsregeln, welche auch Zustandsänderungen ausdrücken, unterscheiden. Bubenko et al. [12] unterscheiden zwischen “constraint rules”, “event-action rules” und “derivation rules”. Martin &

Odell [45] klassifizieren in zwei Klassen: “constraint rules” und “derivation rules”. Interessant ist dabei, dass sie reaktive Regeln (Stimulus-Response Rules) in der Klasse der “constraint rules” subsumieren [57]. Herbst [53] unterscheidet zwischen Integritätsregeln, welche entweder statisch oder dynamisch sein können, und Automatisierungsregeln.

Eine weitere, sehr umfassende Klassifizierungsmöglichkeit, welche in Folge näher beschrieben wird, schlagen die Autoren Boley, Kiefer, Patrânjan und Polleres vor [9]. Laut den genannten Autoren können grundsätzlich folgende Typen von Regeln unterschieden werden [9]:

- *Deduktive Regeln:* Das Ziel von deduktiven Regeln ist es aus bereits bestehendem Wissen durch logische Schlussfolgerung und mathematischen Kalkulationen weiteres Wissen abzuleiten. Oft werden diese auch als abgeleitete Regeln, konstruktive Regeln oder Views bezeichnet [11, 57]. Das Anwendungsgebiet von deduktiven/konstruktiven Regeln ist groß. So fallen laut Bry & Marchiori [11] neben Datenbanksichten (Views) auch Datalog/Prolog-Klauseln, Cascading Style Sheets (CSS)-Selektoren, Extensible Stylesheet Language Transformation (XSLT) Templates oder semantische RDFS-Regeln. Deduktive Regeln beschreiben statische Abhängigkeiten zwischen Entitäten, welche zur Ableitung von zusätzlichem, implizitem Wissen verwendet werden. Folglich werden diese oft als Implikationen in der Form *Konklusion <= Bedingung* (Dann-Wenn) formuliert.
- *Normative Regeln:* Jene Daten, welche normativen Regeln unterworfen sind, müssen spezifizierte (Integritäts-)Bedingungen in all ihren Zuständen gewährleisten. Anstatt neues Wissen abzuleiten, stehen hier bestehende Datenbestände im Fokus. Normative Regeln werden vor allem zur Sicherstellung der Konsistenz von Daten und der Geschäftslogik von Unternehmen eingesetzt. Oft werden diese Regeln auch als Integritätsbedingungen oder strukturelle Regeln bezeichnet [11]. Teilweise können normative Regeln auch durch deduktive oder reaktive Regeln realisiert werden. Normative Regeln können in der Form einer Ablehnung - *falsch/Fehler <= Bedingung* - definiert werden. Trifft eine Anforderung nicht zu, so soll eine Aktion nicht ausgeführt werden können.
- *Reaktive Regeln:* Basierend auf dem aktuellen Zustand eines Systems oder aufgrund von Ereignissen wird über reaktive Regeln das reaktive Verhalten eines Systems in Form von Aktionen spezifiziert. Es werden also automatisch bestimmte Aktionen ausgeführt, wenn gewisse Bedingungen bzw. Ereignisse eintreten. Reaktive Regeln werden oft auch als aktive Regeln oder dynamische Regeln bezeichnet. Im Gegensatz zu deduktiven Regeln, basieren reaktive Regeln auf Zustandsänderungen und deren gewünschten Implikationen. Grundsätzlich können zwei unterschiedliche Typen von reaktiven Regeln unterschieden werden:

- *Produktionsregeln:* Immer dann, wenn eine Änderung auftritt, wird bei Produktionsregeln eine Bedingung überprüft und eine Aktion ausgeführt, wenn die Bedingung erfüllt wurde. Normalerweise beziehen sich diese Aktionen auf die zugrunde liegenden Daten, können aber auch Zuweisungen und Schleifen enthalten. Produktionsregeln werden in der Form *Wenn <Bedingung> Dann <Aktion>* spezifiziert.
- *ECA-Regeln:* Im Gegensatz zu Produktionsregeln werden ECA-Regeln nur dann evaluiert, wenn ein spezifiziertes Ereignis eintritt. Ein Ereignis ist in diesem Zusammenhang als jegliche Änderung eines Zustandes in der Welt definiert. Folglich werden Aktionen nur dann ausgeführt, wenn ein bestimmtes Ereignis eintritt und eine Bedingung erfüllt wird. ECA-Regeln werden in der Form *Bei <Event> Wenn <Bedingung> Dann <Aktion>* formuliert. Abhängig von der jeweiligen Beschreibungssprache der Regel können sowohl einzelne (single/atomic events) als auch zusammenhängende (composite events) Ereignisse definiert werden. Ein zusammenhängendes Ereignis könnte beispielsweise eine Kombination von temporalen, atomaren Ereignissen sein.

Neben den drei genannten Ansätzen - deduktive, normative und reaktive Regeln - existiert laut Taveter & Wagner [56] auch noch ein vierter Ansatz, *deontische Zuweisungen*, der aber nur teilweise anerkannt und nur marginal diskutiert wurde. Die deontische Logik behandelt normative Begriffe wie Verpflichtung oder Erlaubnisse. Eine deontische Zuweisung weist einem internen Agenten Macht, Rechte und Pflichten zu. Durch diese Zuweisungen wird die deontische Struktur einer Organisation definiert und somit die Handlungen der internen Agenten gesteuert und eingeschränkt [56]. Ein Beispiel für eine deontische Zuweisungsregel wäre es, nur dem Verkaufsleiter die Adaption von Preisen eines Verkaufsautomaten zu gestatten.

Aus Sicht einer Web-of-Things-Plattform sind in erster Linie reaktive Regeln, im Speziellen ECA-Regeln, relevant. Deshalb wird dieser Typ von Regeln im nächsten Unterabschnitt genauer vorgestellt und diskutiert.

7.2.4 Event-Condition-Action (ECA) Regeln

ECA-Regeln stellen die bekannteste Form von aktiven Regeln dar. Sie haben ihren historischen Hintergrund in aktiven Datenbanksystemen und wurden das erste Mal im HiPAC-Projekt verwendet [3, 14]. Grundsätzlich sind ECA Regeln in drei Teile gegliedert: Der *Ereignisteil* spezifiziert die Ereignisse, die für den Aufruf der Regel auftreten müssen, der *Bedingungsteil* drückt eine Vorbedingung für das Ausführen der Regel aus, und der *Aktionsteil* enthält die eigentlich auszuführende Aktion. Wenn ihre Ereignisse aufgetreten sind, heißt eine Regel ausgelöst, wenn die Bedingung ausgewertet wurde, be-

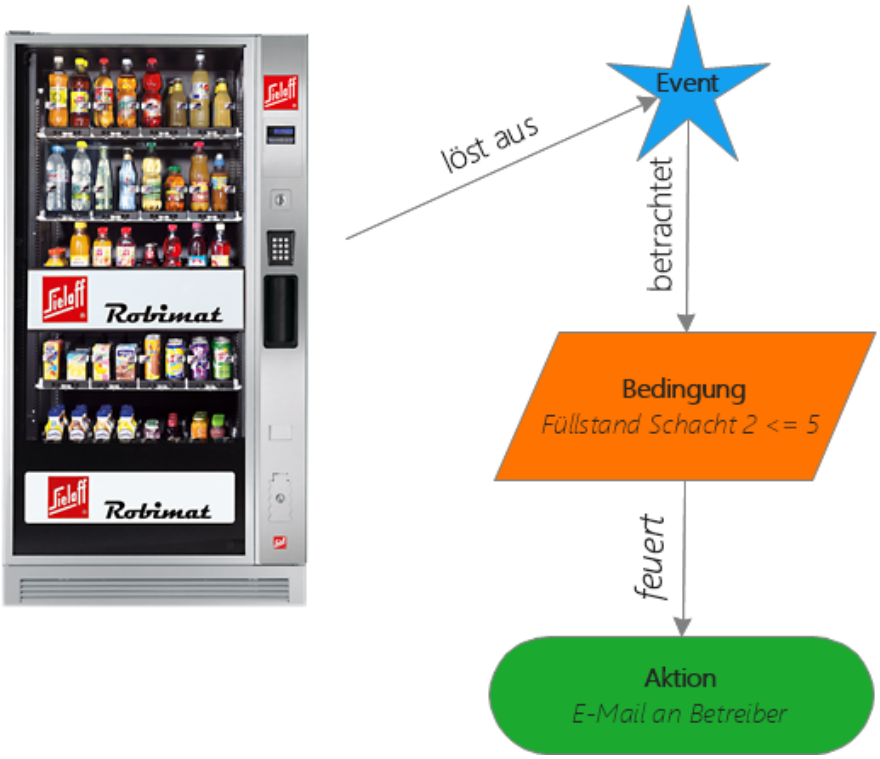

Abbildung 7.1: Eine beispielhafte Darstellung einer ECA-Regel für Smart Vending

zeichnet man die Regel als betrachtet, und wenn die Betrachtung erfolgreich war, wird eine Regel ausgeführt oder auch gefeuert [5]. Die Aufspaltung einer ECA Regel in 3 Teile hat laut Berndtsson & Mellin eine Reihe von Gründen [3]:

1. Ereignisse, Bedingungen und Aktionen haben unterschiedliche Rollen. Während Ereignisse den Zeitpunkt zur Auslösung von Regeln darstellen und Bedingungen festlegen, was es zu überprüfen gilt, spezifiziert die Aktion was ausgeführt wird. Folglich ergibt sich dadurch eine klare, eindeutige Trennung der Semantik und eine Vermeidung einer subjektiven Vermischung dieser Teile.
2. Diese Trennung erlaubt flexible Ausführungssemantiken. So können beispielsweise Ereignisse und/oder Bedingungen gekoppelt werden.
3. Unterschiedliche Rollen optimieren die Performance. Im Gegensatz zu Produktionsregeln werden ECA-Regeln nur dann ausgelöst, wenn ein bestimmtes Ereignis auftritt. Dieser Umstand führt vor allem bei aktiven Datenbanken zu einer Verbesserung der Performance.
4. Die Rollentrennung erleichtert die konzeptuelle Modellierung.

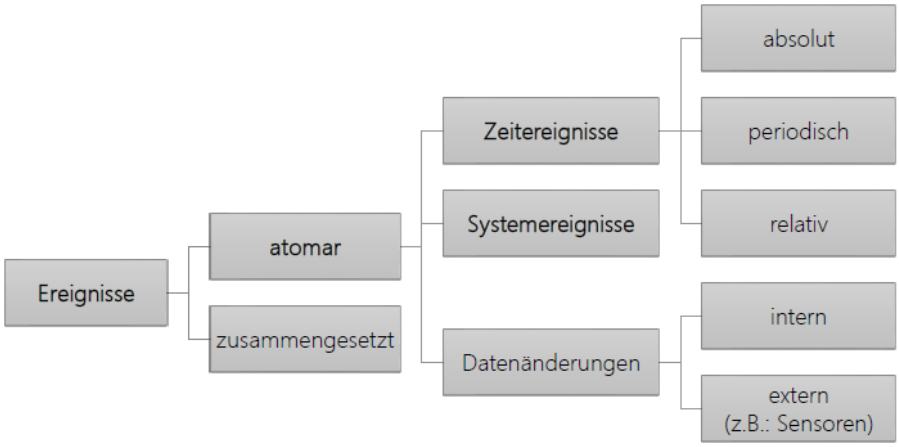

Abbildung 7.2: Unterschiedliche Event-Arten (angelehnt an [5])

Ereignis

Ereignisse sind der treibende Faktor für die Ausführung einer Regel. Meist wird unter Ereignissen ein signalisierter, relevanter Zustandswechsel (intern oder extern) subsumiert [5, 16]. Es gibt jedoch keine klare, eindeutige Definition. Laut Berstel et al. [5] habe ein Ereignis zwei Hauptaufgaben: Einerseits determiniere es den Zeitpunkt der Reaktion und andererseits extrahiere es Daten aus dem Event, welche in der Bedingung oder der Aktion verwendet werden können. Weiters habe ein Ereignis zum Zwecke der Verarbeitung eine Repräsentation in Form eines Objekts oder einer Nachricht. Dabei könne es verschiedenste Quellen bzw. Repräsentationen von Ereignissen geben, die in 7.2 dargestellt sind. Wichtig ist die Unterscheidung zwischen atomaren und zusammengesetzten Ereignissen. Atomare Ereignisse können in Systemereignisse (z. B. CPU-Last, Netzwerkauslastung, etc.), Zeitereignisse und Datenereignisse (z. B. Sensoren, Web-Services, Webformulare, etc.) unterteilt werden. Zeitereignisse können wiederum in drei Untertypen gegliedert werden: Es wird zwischen periodischen (z. B. alle 5 Stunden), absoluten (z. B. 10.08.2015 - 11:00 Uhr) oder relativen Zeitereignissen (z. B. jede Stunde nach Verkauf von mindestens 2 Produkten) unterschieden. Zusammengesetzte Ereignisse stellen eine Auflistung mehrerer atomarer Ereignisse dar, die ein gemeinsames Muster aufweisen. Dieses Muster wird auch als “composite event query” bezeichnet [5].

Bedingung

Bedingungen einer ECA Regel werden üblicherweise in Form einer Abfrage an eine persistente Datenquelle formuliert. Analog zu Ereignissen haben

auch Bedingungen zwei Hauptaufgaben. Einerseits die Überprüfung der Bedingung um eine Regel abzufeuern (die Aktion auszuführen), andererseits um Daten in der Form von (System-)Variablen an die Aktion zu übergeben [5]. Grundsätzlich ist eine Abfrage von (persistenten) Daten aus einer Datenquelle (Datenbank, Subprogramm, etc.) erforderlich, um eine Bedingung überprüfen zu können.

Aktion

Im Aktionsteil wird definiert, wie auf eine bestimmte Situation reagiert werden soll. Während Ereignis und Bedingung nur den aktuellen Zustand eines Systems abfragen, wird dieser durch die Aktion auch modifiziert [5]. Aktionen werden im Zuge einer Transaktion ausgeführt. Eines der Hauptprobleme im Aktionsteil ist, dass Aktionen zeitgleich ausgeführt und sie sich somit beinflussen könnten. Darüber hinaus ist es problematisch, falls Aktionen weitere Regeln anstoßen, indem sie Ereignisse eintreten lassen. Dieser Umstand kann zu einem kaskadierenden Regelanstoß führen [3].

7.3 Business Rules und Smart Vending

Laut den Ergebnissen der Befragung in Abschnitt 3.3 und dem beschriebenen Szenario in Abschnitt 3.4 ergibt sich die Forderung nach der Abbildung von einfachen, regelbasierten Aktionen im Zuge der prototypischen Implementierung der Web-of-Things-Plattform. Hintergrund ist der Wunsch, basierend auf den durch die Plattform konsolidierten Sensordaten, gewisse Geschäftsprozesse zu automatisieren bzw. überhaupt erst zu ermöglichen. Durch die Anbindung von Getränkeautomaten an eine Web-of-Things-Plattform erhält ein Automatenbetreiber erstmals die Möglichkeit Daten in Echtzeit zu erhalten und zu verarbeiten. Folglich ergibt sich die Möglichkeit auf gewisse (geschäfts-)kritische Ereignisse (z. B. Kühltemperatur zu hoch, Automatentür offen, Füllstand niedrig) zu reagieren. Um die genannten Beispiele realisieren zu können, ist aber ein entsprechender Benachrichtigungsmechanismus nötig, welcher durch reaktive Regeln umgesetzt werden kann. Darüber hinaus bietet sich die Chance über Aktuatoren Prozesse zu automatisieren. So kann beispielsweise der Preis von Gütern in Abhängigkeit von der Außentemperatur (Hot/Cold Day Pricing) via Aktuatoren und Regeln automatisiert gesteuert werden.

Ziele

Aus diesen Überlegungen ergeben sich folgende Ziele für die Implementierung von Geschäftsregeln:

- Einfache ECA-Regeln: Die Plattform soll die Erstellung und Ausführung von einfachen Event-Condition-Action (ECA)-Regeln ermögli-

chen. Es sollen atomare Ereignisse, sowie einfache Bedingungen und Aktionen definierbar sein.

- Geschäftsregeln auf Thing-Ebene: Pro komplexem Ding soll es möglich sein einfache Geschäftsregeln zu definieren.
- Reaktion auf Sensordaten: Es soll möglich sein, Geschäftsregeln basierend auf einer Veränderung von Daten eines spezifischen Sensors (Sensor liefert neues Datum) zu definieren (Ereignis).
- Temporale Ereignisse: Neben Datenänderungen sollen auch periodische Zeitpunkte (täglich) als Ereignis definierbar sein.
- E-Mail Benachrichtigungen: Definition von einfachen ECA-Regeln, die eine Benachrichtigung per E-Mail ermöglichen. Dazu sollen Sensordaten in Echtzeit ausgewertet werden.
- Aktuatoren: Neben E-Mail Benachrichtigungen sollen auch Aktuatoren als Aktion einer Regel angesteuert werden.

7.4 Rule-Engines

Um eine effiziente Ausführung von Geschäftsregeln zu ermöglichen, werden vielfach Rule-Engines als Softwarekomponenten eingesetzt. In diesem Abschnitt werden bereits bestehende .NET Rule-Engines hinsichtlich ihrer Eignung für das beschriebene Szenario bzw. die definierten Ziele evaluiert. Um eine Evaluierung durchführen zu können, werden zuerst Evaluierungskriterien definiert. Anschließend werden 4 Rule-Engines ausgewählt und untersucht. Im .NET-Bereich gibt es zwar mehr als ein Dutzend solcher Softwarekomponenten, viele davon wurden aber kaum gewartet und weiterentwickelt. Folglich werden nur jene Rule-Engines in Betracht gezogen, die auch noch in den letzen beiden Jahren weiterentwickelt wurden.

7.4.1 Kriterien

Nachfolgend werden die Kriterien aus dem in Abschnitt 7.3 entwickelten Szenario und der Zielsetzung abgeleitet. Ein zusätzliches KO-Kriterium ist der Umstand, dass ausschließlich kostenfreie Softwarebibliotheken zum Einsatz kommen können.

1. Kostenfreie Lizenzierung: Die Rule-Engine muss kostenlos und Open Source sein.
2. Kompatibilität: Die Softwarebibliothek muss mit .NET 4.5.1 kompatibel sein und sich in die Serverkomponente ASP.NET Web API integrieren lassen. Darüber hinaus muss eine Persistierung der Regeln in Microsoft Azure möglich sein.
3. ECA-Regeln: Es muss eine Unterstützung für Event-Condition-Action (ECA)-Regeln vorhanden sein. Die Erstellung der Regeln muss auch

	InRule[1]	*NRules*[2]	*Flexrule*[3]	*NxBre*[4]
Kostenfrei	Nein	Ja	Nein	Ja
Kompatiblität	Ja	Ja	Ja	Nein
ECA-Regeln	Ja	Ja	Ja	Ja
Temporale Regeln	k.A.	Nein	Nein	Nein
Bedingung	Ja	Ja	Nein	Ja
Eigene Aktionen	Nein	Ja	Nein	Nein
Datenformat	XML	.Net-Objekte	XML	XML

Tabelle 7.1: Ergebnis der Evaluierung der Rule-Engines

programmatisch möglich sein, da der Endbenutzer nur mit der Web-of-Things-Plattform interagiert und dort seine Regeln definiert.

4. Temporale Regeln: Neben datenbasierten Regeln sollen auch zeitbasierte (periodische) Regeln unterstützt werden.
5. Bedingung: Der Bedingungsteil muss eine dynamische Definition der zu vergleichenden Objekte erlauben.
6. Eigene Aktionen: Im Aktionsteil sollten frei definierbare Aktionen (Aktuator, E-Mail) definiert werden können.
7. Datenformat: Die Serverkomponente von WoTCloud muss das entsprechende Datenformat der Regeln verarbeiten können.

7.4.2 Ergebnis

Die Tabelle 7.1 fasst das Ergebnis der Evaluierung in Form eines Vergleichs anhand der von den Autoren identifizierten Kriterien zusammen. Generell ist der Hauptanwendungszweck vieler Rule-Engines einen Inferenzmechanismus für deduktive Regeln anzubieten und nicht die Unterstützung reaktiver Regeln. InRule und FlexRule sind aufgrund ihres kommerziellen Charakters und der fehlenden Unterstützung für temporale Regeln und eigene Aktionen nicht einsetzbar. NxBre unterstützt auch in der aktuellsten Version laut Dokumentation nur .NET 2.0 und scheidet aufgrund des mangelnden Supports für .NET 4.5 aus. Der letzte Kandidat, NRules, ist ein noch sehr junges Framework, welches von vier Entwicklern auf GitHub vorangetrieben wird. Es basiert auf dem Rete-Algorithmus und nützt Reactive Ex-

tensions[5], eine Bibliothek für asynchrone und event-basierte Erweiterungen von C#-Collections. NRules unterstützt die Spezifikation von Regeln nur in .NET-Objekten selbst und ermöglicht durch Reactive Extensions sowohl ECA-Regeln als auch eigene Bedingungen und Aktionen. Es gibt jedoch keine Unterstützung für temporale Ereignisse.

7.4.3 Schlussfolgerung

Der Autor ist nach Durchführung der Evaluierung der Auffassung, dass das entwickelte Szenario aus Abschnitt 7.3 durch die evaluierten Rule-Engines nicht ausreichend unterstützt wird, da keine der Rule-Engines alle definierten Kriterien unterstützt. Obwohl NRules für datengetriebene Regeln anwendbar wäre, lässt die ohnehin notwendige Implementierung der .NET-Objekte und deren Persistierung sowie die Notwendigkeit für eine Eigenimplementierung für temporale Regeln eine vollständige Eigenimplementierung der definierten Anforderungen sinnvoller erscheinen.

7.5 Web-of-Things Rules

Der Prototyp der implementierten Web-of-Things-Plattform (WoTCloud) unterstützt grundsätzlich zwei Ansätze reaktiver Event-Condition-Action (ECA) Regeln: Jene, deren Ereignis durch Daten (Sensoren) getriggert werden und jene, die aufgrund eines definierten, täglichen Zeitpunktes ausgelöst werden. Die Auswahl des jeweiligen Typs erfolgt bereits bei der Anlage der Regel. Regeln können sowohl bei der Neuanlage von Things als auch bei der Anlage von Templates erstellt werden. Wird ein Thing basierend auf einer Vorlage erstellt, so werden per Default alle Regeln des Templates übernommen. Der Benutzer hat jedoch vor dem Speichern die Möglichkeit diese Regeln abzuändern, neue Regeln hinzuzufügen bzw. übernommene Regeln zu löschen.

7.5.1 Überblick

Grundsätzlich ergeben sich somit mehrere Varianten von Geschäftsregeln in WoTCloud, die in Tabelle 7.2 dargestellt sind. Der Benutzer kann zwischen Datenereignissen, die auf jeden neuen Datenwert eines Sensors reagieren und Zeitereignissen, die eine Spezifikation von Zeitpunkten, welche täglich zur Auslösung der Regeln führen, entscheiden. Die Bedingung erlaubt es aus 6 verschiedenen Operatoren - kleiner, kleiner gleich, größer, größer gleich, gleich und ungleich - auszuwählen und einen bestimmten Sensor eines Things mit einem spezifizierten Wert zu vergleichen. Bei Regeln mit Datenereignissen wird immer der aktuelleste Sensorwert, also jener, der

[5] https://msdn.microsoft.com/en-us/data/gg577609.aspx

Ereignis	Datenregel - Sensordaten		
	Zeitbasierte Regel - periodische Zeitpunkte		
Bedingung	Neuer/Letzer Sensorwert	< <= > >= == !=	Bedingungswert
Aktion	Aufruf Aktuator		
	E-Mail Benachrichtigung		

Tabelle 7.2: Mögliche Ausprägungen von Geschäftsregeln in WoTCloud

gerade neu hinzugefügt wird, mit der Bedingung verglichen. Bei Zeitereignissen wird der letzte persistierte Sensorwert mit dem in der Bedingung spezifizierten Wert verglichen. Im Aktionsteil stehen sowohl Aktuatoren als auch E-Mail-Benachrichtigungen zur Auswahl.

In den weiteren Unterabschnitten wird die Implementierung von Datenregeln und Zeitregeln detaillierter vorgestellt.

7.5.2 Data Rules

Das Ereignis wird bei datengetriebenen Regeln über Daten getrieben, d. h. dass jedes Datum, welches von einem Sensor übermittelt wird, zu einer Überprüfung der Bedingung der Regeln führt. Dieser Typ von Regeln wird also von den Sensoren der Things getrieben. Abbildung 7.3 veranschaulicht die Anlage solcher Regeln in WoTCloud. Die erste Regel *Hot Cold Day Pricing* wird durch die Übermittlung eines neuen Wertes des Außentemperatursensors ausgelöst und bei einem Wert größer oder gleich 30 Grad Celsius wird die Regel gefeuert. Die Aktion besteht in einem Aktuator-Aufruf. Der Aktuator wurde durch eine URI-spezifiziert, welcher dann mit dem Wert “3” im Body des HTTP-Requests (POST) aufgerufen wird. Die zweite Regel wird durch den Füllstandssensor des ersten Schachts des Automaten ausgelöst. Ist nur mehr ein Produkt im Schacht vorhanden (Füllstand kleiner 2), so wird der Operator per E-Mail mit der Meldung “Refill Dispenser 4711

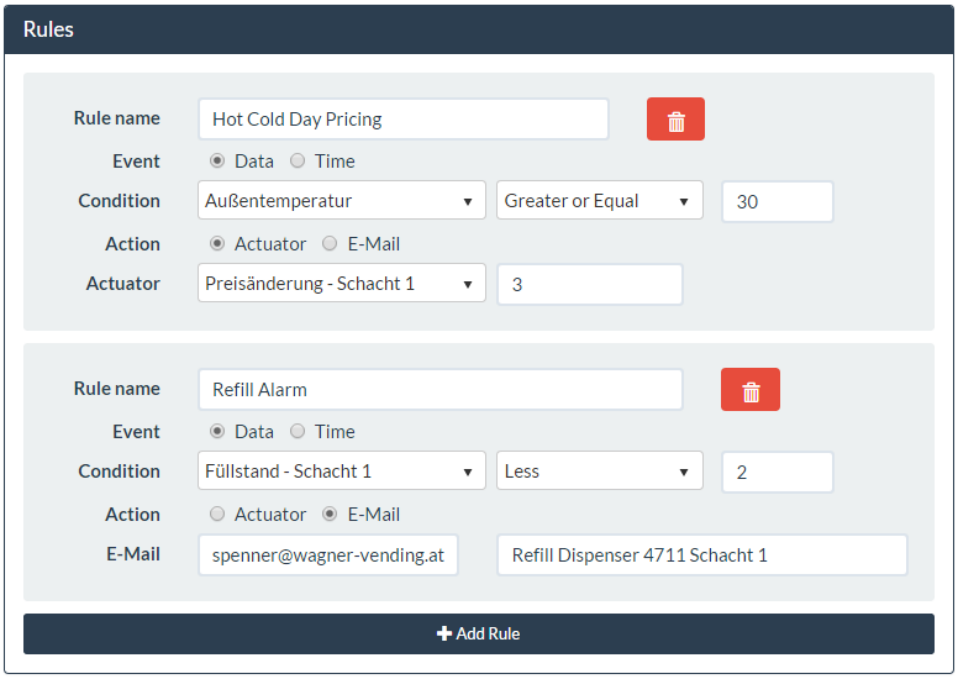

Abbildung 7.3: Anlage datengetriebener Regeln in WoTCloud

Listing 7.1: HTTP-Post für das Einfügen von Sensordaten

```
1 POST http://wotcloud.azurewebsites.net/api/1/things/1/sensors/2 HTTP/1.1
2 Accept: application/json, text/plain, */*
3 Content-Type: application/json
4 Authorization: Bearer j1jzWkmP3BfOmaiSHSsqsAkvGq...
5
6 15
```

Schacht 1" benachrichtigt.

Da die zuvor beschriebenen Rule Engines die gewünschte Funktionalität nicht oder nur teilweise unterstützen, wurde die Unterstützung für datengetriebene Regeln selbst implementiert. Diese Entscheidung wurde zusätzlich durch den Umstand bestätigt, dass für eine Web-of-Things-Plattform hauptsächlich reaktive Regeln, die keinen Inferenzmechanismus benötigen, relevant sind. Der Aufruf zur Übermittlung von Daten eines Sensor erfolgt, wie in Listing 7.1 dargestellt, über einen HTTP-Post mit dem entsprechenden Wert (15) im Body des Requests. Die Serverkomponente verarbeitet den Aufruf, löst die HTTP-Route auf, validiert die Daten und übergibt sie schließlich der zur Route passenden Action-Methode (siehe Listing 7.2).

Die Methode *InsertSensorValue* überprüft zuerst ob eine gültige Authentifizierung vorliegt und ob der im HTTP-Request angegebene Token für

Listing 7.2: Action-Methode zum Einfügen des Sensorwerts und Anstoß zur ansynchronen Regelprüfung

```
/// <summary>
/// Inserts a new value for a specific sensor
/// </summary>
/// <param name=tenantId>id of the tenant</param>
/// <param name=thingId>id of the thing</param>
/// <param name=sensorId>id of the sensor</param>
/// <param name=value>sensor value</param>
/// <returns>HttpStatusCode 200 if sucessful</returns>
[HttpPost]
[Route("{sensorId:int}")]
[ValidateModel]
public HttpResponseMessage InsertSensorValue(int tenantId, int thingId,
    int sensorId, [FromBody]string value)
{
    ////Authentication & Authorization
    if (!AuthorizationHelper.VerifyTenantClaim(this.User, tenantId.
    ToString(CultureInfo.InvariantCulture)))
    {
        return Request.CreateErrorResponse(HttpStatusCode.Forbidden,
    ErrorMessageHelper.NotAuthorized);
    }

    ////Stores the new value in Azure Table-Storage
    var storageManager = new StorageManager(tenantId.ToString(
    CultureInfo.InvariantCulture));
    storageManager.InsertSensorValue(sensorId.ToString(CultureInfo.
    InvariantCulture), value);

    Task.Run(() => this.TriggerDataRules(thingId, sensorId, value));

    return Request.CreateResponse(HttpStatusCode.OK);
}
```

diesen Tenant berechtigt ist. Anschließend wird über die Klasse StorageManager, die den Datenzugriff aller Action-Methoden auf den Azure Table Storage kapselt, der neue Wert persistiert. Zu guter Letzt wird mit dem Aufruf der Methode *TriggerDataRules* eine Überprüfung der datengetriebenen Regeln angestoßen. Um weder Server noch Client durch diese Überprüfung zu blockieren, erfolgt dieser Aufruf im Hintergrund. Zu diesem Zweck wird das seit C# 4.0 verfügbare Konzept der Tasks verwendet. Laut Albahari & Albahari [1] wurden diese Tasks geschaffen, um gewisse Nachteile von Threads zu eliminieren. So stelle ein Task eine Abstraktion von Threads dar, indem es lediglich eine Operation repräsentiere, die gleichzeitig ausgeführt werden solle. Ob ein Task von einem Thread erledigt wird oder nicht, sei nun nicht mehr relevant, es stelle lediglich ein Versprechen dar, dass der Action-Delegate ausgeführt, und falls vorhanden, der spezifizierte

Listing 7.3: Eigentliches Auslösen, Betrachten und Feuern der Regeln eines Sensors

```
1 /// <summary>
2 /// Verifies if data-bound rules for a given sensor exist
3 /// and triggers them
4 /// </summary>
5 /// <param name=thingId>id of the thing</param>
6 /// <param name=sensorId>id of the sensor</param>
7 /// <param name=value>new value</param>
8 private void TriggerDataRules(int thingId, int sensorId, string value)
9 {
10     using (var context = new EfUnitOfWork())
11     {
12         var rules = context.RuleRepository.Find(t => t.ThingId ==
       thingId && t.SensorId == sensorId).ToList();
13         foreach (var r in rules.Where(r => RuleHelper.EvaluateCondition(
       r.ConditionValue, r.ConditionOperator, value)))
14         {
15             if (r.ActionType == RuleActionType.Actuator && r.ActuatorId.
       HasValue)
16             {
17                 var act = context.ActuatorRepository.SingleOrDefault(t
       => t.Id == r.ActuatorId.Value);
18                 if (act != null)
19                 {
20                   RuleHelper.ExecuteActuator(act.ActuatorUri, r.
       ActionValue);
21                 }
22             }
23             else if(!string.IsNullOrWhiteSpace(r.ActionEmailRecipient)
       && !string.IsNullOrWhiteSpace(r.ActionValue))
24             {
25                 RuleHelper.SendEMail(r.ActionEmailRecipient, r.
       ActionValue);
26             }
27         }
28     }
29 }
```

Rückgabetyp retourniert werde [1]. Folglich bietet ein Task viele Vorteile. Neben der Möglichkeit Tasks zu verketten, zu stoppen und auf Exceptions zu reagieren, dienen Tasks auch als generischer Rückgabetyp von asynchronen Methoden. In unserem Beispiel genügt jedoch der einfache Aufruf von *Task.Run()* kombiniert mit einem Action-Delegate (die Methode TriggerDataRules) als Parameter.

Das Auslösen der Regeln eines Sensors findet nun wie zuvor erwähnt durch die Methode *TriggerDataRules* statt. Listing 7.3 veranschaulicht die Implementierung. Zuerst wird ein Datenkontext instantiiert, der für die Abfrage der Daten mit Hilfe der Datenzugriffsschicht (RuleRepository) dient.

Es werden nur jene Regeln geladen, die zu einem spezifischen Thing und einem spezifischen Sensor gehören. Diese Bedingung ist im Predicate der Erweiterungsmethode "Find" definiert. Anschließend wird mit der Foreach-Schleife über all jene Regeln iteriert, deren Bedingung einen positiven Rückgabewert retourniert hat. Die Erweiterungsmethode "Where" gehört zu einem Sprachkonstrukt von C# namens Language Integrated Query (LINQ). In diesem sogenannten Lambda-Ausdruck (Predicate) wird die Methode EvaluateCondition aufgerufen, die den neuen Sensorwort unter Berücksichtigung des Operators mit dem in der Regel spezifizierten Bedingungswert vergleicht. Ist der Vergleich positiv, so wird die Regel gefeuert. In unserem Fall wird hierbei noch zwischen dem Aufruf eines Aktuators und einer E-Mail Benachrichtigung unterschieden. Der Aktuatoraufruf resultiert in einem simplen HTTP-Post Request an die spezifizierte URI und dem spezifizierten Wert im Body des Requests.

7.5.3 Time Rules

Temporale Regeln werden durch die Defintion von verschiedenen Zeitpunkten bzw. Zeitintervallen definiert. WoTCloud unterstützt nur periodisch wiederkehrende Zeitpunkte. Konkret gibt es die Möglichkeit einen täglichen Zeitpunkt (z. B. 16:30) zu definieren, der schlussendlich zur Auslösung der Regel führt. Es ist also nicht möglich relative bzw. absolute Zeitpunkte zu definieren. Während datengetriebene Regeln einen Dateninput eines Sensors benötigen, werden temporale Regeln nur durch Zeit determiniert. Abbildung 7.4 veranschaulicht die Anlage temporaler Regeln in WoTCloud. Die erste Regel *DemandPricing* wird täglich um 15:00 ausgelöst und ruft den Aktuator *Preis Schacht 2* zur Veränderung des Schachtpreises auf, falls der aktuelleste Wert des Sensors *Füllstand Schacht 2* kleiner als 5 ist. Die zweite Regel erhöht den Preis nochmals ab 20:00 Uhr, wiederum nur falls der Füllstand kleiner als 5 ist.

Die Implementierung dieser temporalen Regeln erfolgt analog zu den datengetriebenen Regeln ohne Einbindung einer bestehenden Rule-Engine. Um die Auslösung der zeitabhängigen Regeln sicherzustellen, wird auf einen Scheduler zurückgegriffen, der von Microsoft's Cloud Plattform Azure angeboten wird. Mithilfe sogenannter WebJobs bietet Microsoft die Möglichkeit innerhalb einer Webapplikation eigene Programme bzw. Skripte laufen zu lassen. Es werden drei Ausführungsmechanismen unterstützt: bei Bedarf, kontinuierlich oder basierend auf einem Zeitplan. WebJobs unterstützen Programme und Skripte mit den unterschiedlichsten Dateitypen (z. B..cmd, .bat, .exe (Windows cmd), .ps1 (Powershell), .sh (Bash), .php (Php), .py (Python), .js (Node.js), .jar (java), etc.) [42].

Der Autor hat die Implementierung mittels einer kleinen Konsolen-Applikation realisiert. Diese Applikation nützt die Datenzugriffsschicht der Serverkomponente und sucht in der Datenbank nach aktiven, temporalen Re-

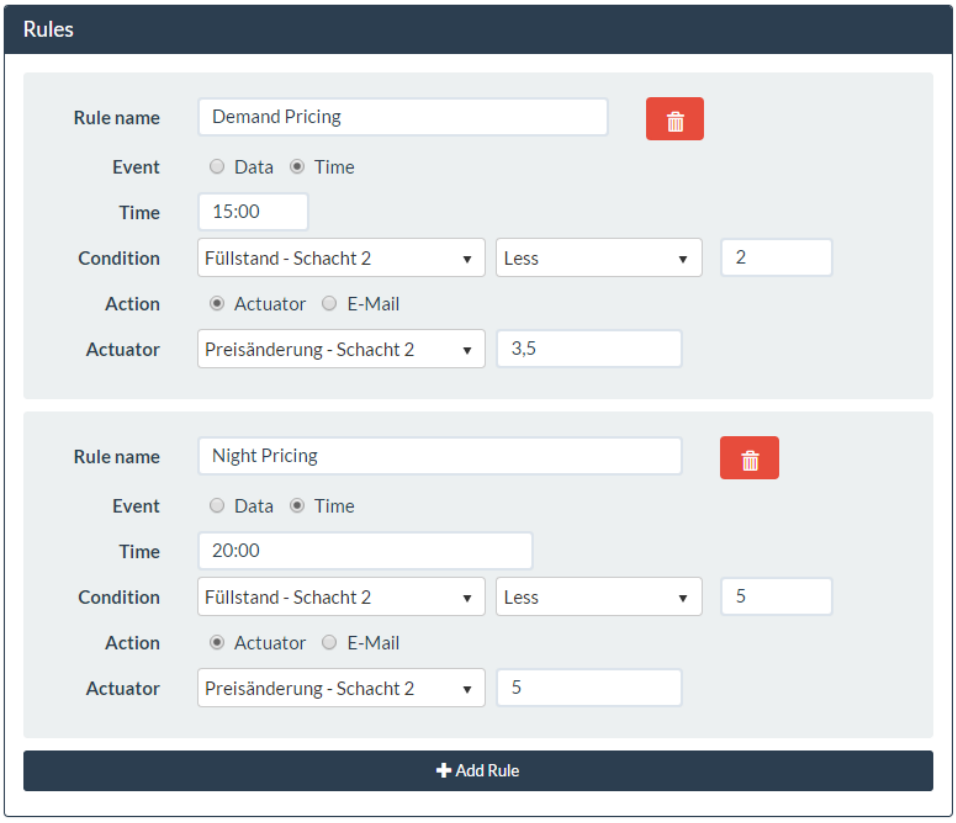

Abbildung 7.4: Anlage temporaler Regeln in WoTCloud

geln mit der aktuellen Tageszeit. Falls solche Regeln existieren, werden jeweils die aktuellsten Sensorenwerte zu diesen Regeln geladen. Anschließend werden die temporalen Regeln ähnlich wie die zuvor beschriebenen datengetriebenen Regeln (siehe Listing 7.3) betrachtet und im Falle eines positiven Vergleichs des Sensorwertes mit der spezifizierten Bedingung abgefeuert. Die Konsolenanwendung wird von Visual Studio in eine .exe-Datei kompiliert. Darüber hinaus kümmert sich die Entwicklungsumgebung auch noch um das automatische Deployment zu einem Azure WebJob. Dazu ist die Definition eines Zeitplanes notwendig, welcher entweder über einen Dialog oder direkt als JSON-File angelegt werden kann. Listing 7.4 veranschaulicht den Zeitplan des WebJobs *WoTCloudTemporalRule*, der zwischen Juni und August 2015 täglich zu Beginn jeder Minute ausgeführt wird.

Der große Vorteil von Azure WebJobs sind die einfache Verwaltung und die Skalierungsmöglichkeiten. So könnten bei größerem Lastaufkommen einfach mehrere WebJobs erstellt werden, die sich abwechselnd (z. B. durch Round-Robin) um die temporalen Regeln kümmern. Neben dem einfachen Deployment bietet Azure auch eine übersichtliche Historie der Aufträge an.

Listing 7.4: Definition des Zeitplanes eines Azure WebJobs in VisualStudio

```
{
    "$schema": "http://schemastore.org/schemas/json/webjob-publish
    -settings.json",
    "webJobName": "WoTCloudTemporalRule",
    "startTime": "2015-06-01T00:00:00+01:00",
    "endTime": "2015-09-01T00:00:00+01:00",
    "jobRecurrenceFrequency": "Minute",
    "interval": 1,
    "runMode": "Scheduled"
}
```

Es sind die letzten gestarteten Jobs inklusive Status, sowie in einer weiteren Detailansicht sogar detaillierte Logs (inkl. Konsolenausgaben der implementierten Anwendung) ersichtlich. Abbildung 7.5 illustriert das Zusammenspiel der unterschiedlichen Komponenten. Ein Scheduler kümmert sich um die im Zeitplan spezifizierten Aufrufe der WebJobs, die innerhalb einer Azure Webanwendung deployed sind. Der Webjob, in unsererm Fall die Konsolenanwendung, kümmert sich anschließend um die Ausführung der temporalen Regeln. Die Tatsache, dass ein Webjob innerhalb einer Webanwendung existiert und sogar dieselbe Datenzugriffsschicht (gleiche .DLL) benutzt, erlaubt ihm auch den Zugriff auf die SQL-Datenbank sowie den nicht-relationalen Table-Storage.

7.6 Zusammenfassung

Dieses Kapitel diente der Beschreibung von Geschäftsregeln und der Unterstützung von reaktiven Regeln in WoTCloud. Zuerst wurden Geschäftsregeln theoretisch fundiert und deren praktische Relevanz motiviert. Es wurde eine allgemeine Klassifizierung von deduktiven, normativen und reaktiven Regeln vorgenommen. Darauf aufbauend wurden Event-Condition-Action (ECA) Regeln detaillierter vorgestellt. Anschließend wurden die Anforderungen für die Abbildung von einfachen, regelbasierten Aktionen im Zuge der prototypischen Implementierung der Web-of-Things-Plattform beschrieben. Diese Anforderungen wurden als Grundlage für die Evaluierung von vier verschiedenen Rule-Engines verwendet. Die Ergebnisse der Evaluierung ließen den Autor zum Schluss kommen, dass eine eigenständige Implementierung sinnvoller wäre. Schlussendlich wurden darauf eingegangen, wie die Unterstützung für reaktive Regeln im Detail implementiert wurde. Insbesondere wurden Azure WebJobs für die Umsetzung von temporalen Regeln vorgestellt.

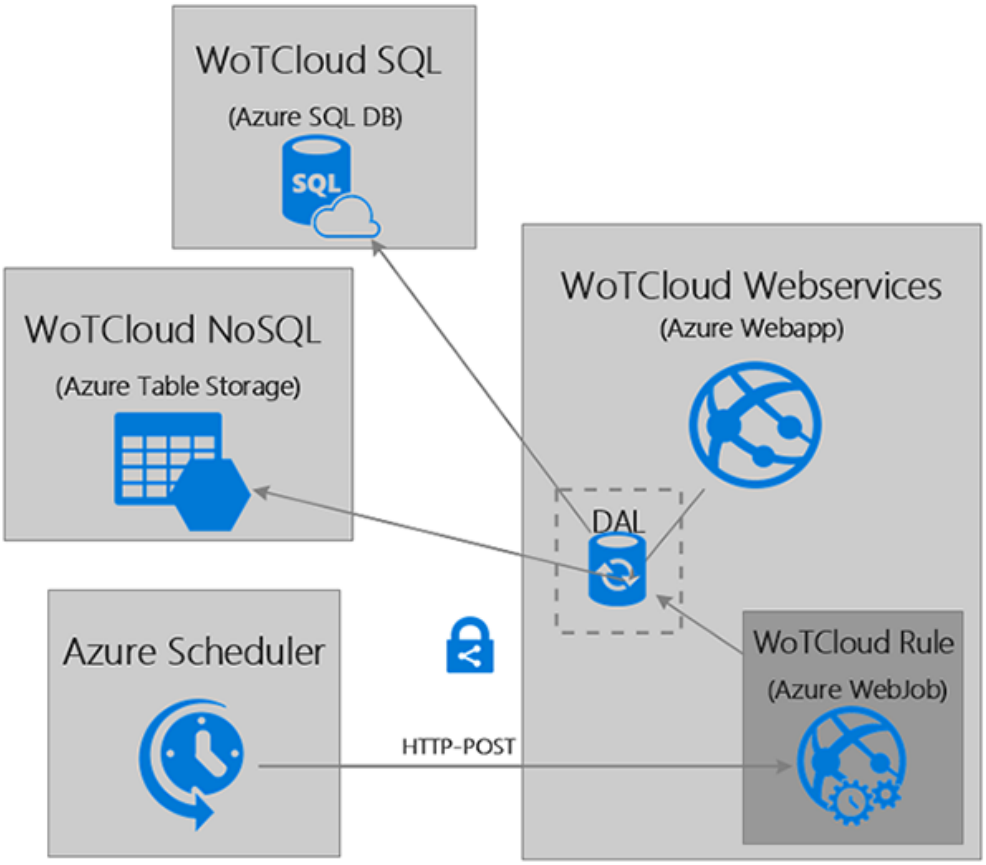

Abbildung 7.5: Anstoß temporaler Regeln durch Azure WebJobs

Kapitel 8

Fazit und Ausblick

8.1 Zusammenfassung

Die zunehmende Vernetzung von Geräten über das Internet führte dazu, dass das World Wide Web und assoziierte Technologien nun auch als technische Basis für Smart Things dienen. Das sogenannte Web-of-Things nutzt neben den erprobten Webstandards auch einen REST-basierten Architekturstil, um eine Integration und Kommunikation von und mit smarten Dingen in standardisierter Art und Weise zu ermöglichen. Im Rahmen dieser Arbeit wurde ein Ansatz entwickelt, der auch die Integration komplexer Dinge in das Web-of-Things ermöglicht. Zu diesem Zweck wurde ein umfangreiches Szenario im Bereich des Smart Vendings vorgestellt. Die Anforderungen für dieses Szenario wurden durch eine Befragung von Marktteilnehmern dieser Branche abgeleitet. Anschließend wurden bestehende Web-of-Things-Plattformen systematisch untersucht und hinsichtlich ihrer Eignung zur Abbildung von Automaten evaluiert.

Aufgrund mangelnder Unterstützung für komplexe Dinge entschlossen sich die Autoren dazu eine eigene Web-of-Things-Plattform namens WoT-Cloud prototypisch zu entwickeln. Es wurde ein konzeptuelles Modell erstellt, welches sowohl eine Unterscheidung zwischen simplen und komplexen Dingen als auch einen Multi-Tenancy-Ansatz und die Unterstützung von Geschäftsregeln und Templates ermöglicht. Aufbauend darauf wurde eine cloudbasierte Software-as-a-Service Lösung mit REST-basierten Webservices und einem HTML5/AngularJS basierten Webclient entwickelt. Die Webplattform erlaubt eine generische Anlage und Verwaltung beliebiger Automaten sowie die Definition von Geschäftsregeln.

Der besondere Fokus dieser Masterarbeit lag in der REST-basierten Architektur der Web-of-Things-Plattform. Neben den wichtigen Konzepten von REST wurde insbesondere die Relevanz von REST für das Web-of-Things motiviert, sowie eine systematische Integration von smarten Dingen in die Web-of-Things-Plattform vorgestellt. Es wurden sowohl die Ressourcen, Re-

präsentationen und Service-Interfaces als auch die Deployment-Strategie von WoTCloud systematisch vorgestellt und begründet. Darüber hinaus wurde die Unterstützung und Relevanz von reaktiven Regeln in WoTCloud diskutiert. Neben datengetriebenen Regeln wurde auch die Implementierung von temporalen Regeln via Azure WebJobs im Detail veranschaulicht.

8.2 Ausblick

Dieser Abschnitt diskutiert zukünftige Forschungsfelder, die im Zuge dieser Arbeit identifiziert wurden. Weiters werden Einschränkungen bzw. derzeitige Schwächen des Prototypen adressiert und mögliche Lösungswege aufgezeigt. Die folgenden Aspekte bieten Möglichkeiten für zukünftige Verbesserungen der Plattform.

HATEOAS Wie schon in Abschnitt 6.4.2 erwähnt, bietet der Prototyp keine standardisierte Verlinkung der Ressourcen, da derzeit keine Implementierung solch eines Standards (beispielsweise JSON-LD) für ASP.NET existiert. Für zukünftige Versionen der Plattform sollte aber eine Standardisierung und somit ein weiterer Schritt in Richtung RESTful-Webservices getätigt werden.

Security Ein großes Forschungsthema der Zukunft ist die Implementierung von Sicherheitsmaßnahmen für smarte Dinge. Trotz der enorm raschen Verbreitung vernetzter Geräte gibt es noch unzulängliche Möglichkeiten, Things mit beschränkten technischen Ressourcen adäquat abzusichern.

Komplexität WoTCloud unterstützt derzeit smarte Dinge in zwei unterschiedlichen Komplexitätsstufen: einfache und komplexe Dinge. Dennoch ist das Design der Webservices und insbesondere der URI-Struktur so ausgelegt, dass in einem weiteren Schritt beliebig tiefe dynamische Produkthierarchien definiert werden können. Dazu müsste das konzeptuelle Modell aus Abschnitt 5.3 insofern adaptiert werden, als komplexe Dinge sich - anstatt aus einer Menge von simplen - aus einer Menge an abstrakten Dingen zusammensetzen müssten. Die größte Herausforderung in diesem Kontext würde die Adaptierung des Webclients darstellen.

Einbindung Kunde Der bisherige Ansatz von WoTCloud unterstützt vor allem die Betreiber von Automaten. Aus diesem Grund wäre auch die Einbindung der Kunden (z. B. via Smartphone-App) wünschenswert. Die bestehende Plattform stellt eine solide Basis für die Umsetzung erweiterter Anwendungsszenarien für den Bereich Smart Vending dar.

Alternative Anwendungsszenarien Obwohl WoTCloud insbesondere zur Unterstützung komplexer Dinge im Bereich Smart Vending entwickelt wurde, sind mittels der implementierten Plattform auch andere Anwendungsszenarien umsetzbar. Vor allem aufgrund der generischen Ressourcen (Tenant, ComplexThing, SimpleThing, Sensor, Actuator, Rule) sind vielfältige Szenarien denkbar.

Teil III

Anhang

Anhang A

Inhalt der CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 WoTCloud.Client

In diesem Ordner befinden sich sämtliche Quellcodedateien und Bibliotheken, die für die Erstellung des Webclients verwendet wurden.

A.1.1 Deployment

Der Webclient ist vollständig in HTML & Javascript bzw. AngularJS implementiert und läuft somit auf allen gängigen Webservern (z. B. Microsoft Internet Information Service (IIS), Apache Tomcat, etc.). Das Deployment kann in diesem Fall durch einen simplen Kopiervorgang der Dateien (beispielsweise über FTP) erfolgen.

A.2 WoTCloud.Service

In diesem Ordner befinden sich sämtliche Quellcodedateien des Service- und Data Layers von WoTCloud. Die REST-basierten Webservices wurden mithilfe von Microsoft's ASP.NET Web API erstellt. Zusätzlich finden sich im Unterordner *Scripts* alle SQL-Scripts der benötigten Entitäten. Der Data Layer basiert auf Microsoft's Entity Framework und nützt den *Code-First* Ansatz. Aus diesem Grund ist es nicht notwendig, die SQL-Skripts auf dem Datenbankserver auszuführen, da die Tabellen und Constraints beim ersten Start der Applikation automatisch erzeugt werden.

A.2.1 Deployment

Der Service Layer erfordert Microsoft IIS als Webserver. Alternativ dazu ist auch ein Deployment als Web App auf Windows Azure möglich. Das

Deployment kann direkt aus der Entwicklungsumgebung Visual Studio von Microsoft erfolgen (Publish to Azure bzw. Publish Web Site). Der Data Layer erfordert einen MS-SQL Server bzw. eine (cloud-basierte) SQL Azure Database von Microsoft Azure. Vor dem Start der Applikation ist der entsprechende Connection-String zur Datenbank in der Datei *web.config* anzupassen.

A.3 WoTCloud.Simulator

In diesem Ordner befinden sich sämtliche Quellcodedateien des erstellten Simulators für Getränkeautomaten. Der Simulator basiert auf Microsoft's ASP.NET MVC.

A.3.1 Deployment

Der Simulator erfordert Microsoft IIS als Webserver. Das Deployment kann direkt aus der Entwicklungsumgebung Visual Studio von Microsoft erfolgen (Publish to Azure bzw. Publish Web Site).

Quellenverzeichnis

[1] Joseph Albahari und Ben Albahari. *C# 5.0 in a Nutshell: The Definitive Reference*. Fifth Edit. Sebastopol, CA, USA: O'Reilly Media, Inc., 2012, S. 1044 (siehe S. 89, 90).

[2] Luigi Atzori, Antonio Iera und Giacomo Morabito. „The Internet of Things: A Survey". In: *Comput. Netw.* 54.15 (2010), S. 2787–2805 (siehe S. 2, 7).

[3] Mikael Berndtsson und Jonas Mellin. *ECA Rules*. 2009 (siehe S. 80, 81, 83).

[4] Tim Berners-Lee, R Fielding und Larry Masinter. *RFC 3986*. Techn. Ber. 2005 (siehe S. 50).

[5] Bruno Berstel u. a. „Reactive Rules on the Web". English. In: *Reasoning Web: Third International Summer School 2007*. Hrsg. von Grigoris Antoniou u. a. Bd. 4636. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 183 (siehe S. 77, 78, 81–83).

[6] Michael Blackstock und Rodger Lea. „IoT interoperability: a hub-based approach". In: *Internet of Things (IOT), 2014 International Conference on the* (2014), S. 79–84 (siehe S. 14, 64, 65).

[7] Michael Blackstock und Rodger Lea. „Toward interoperability in a web of things". In: *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication - UbiComp '13 Adjunct*. ACM. 2013, S. 1565–1574 (siehe S. 14, 15).

[8] Michael Blackstock und Rodger Lea. „WoTKit: A Lightweight Toolkit for the Web of Things". In: *Proceedings of the Third International Workshop on the Web of Things*. ACM, 2012, S. 3 (siehe S. 2, 15).

[9] Harold Boley und Michael Kifer. „Rule Interchange on the Web". English. In: *New York*. Hrsg. von Grigoris Antoniou u. a. Bd. 4636. Lecture Notes in Computer Science September. Springer Berlin Heidelberg, 2007, S. 269–309 (siehe S. 76, 77, 79).

[10] David L. Brock. *The electronic product code (EPC): a naming scheme for physical objects (White paper)*. Cambridge, MA, USA: Auto-ID Center, 2001, S. 21 (siehe S. 6).

[11] François Bry und Massimo Marchiori. „Ten theses on logic languages for the Semantic Web“. English. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Hrsg. von François Fages und Sylvain Soliman. Bd. 3703 LNCS. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 42–49 (siehe S. 79).

[12] Janis a Bubenko Jr. und J Stirna. *EKD User Guide*. Stockholm, Sweden, 1998. URL: http://people.dsv.su.se/~js/ekd%5C_user%5C_guide.html (besucht am 10. 06. 2015) (siehe S. 78).

[13] Bundesverband der Deutschen Vending-Automatenwirtschaft e.V. *Vending - Der Verkauf von Waren durch Automaten*. 2015. URL: http://www.bdv-vending.de/branche/vending/ (besucht am 22. 04. 2015) (siehe S. 16).

[14] M. J. Carey, M. Livny und R. Jauhari. „The HiPAC project: combining active databases and timing constraints“. In: *ACM SIGMOD Record* 17.1 (1988), S. 51–70 (siehe S. 80).

[15] Anis Charfi und Mira Mezini. „Hybrid web service composition“. In: *Proceedings of the 2nd international conference on Service oriented computing - ICSOC '04*. ICSOC '04. New York, NY, USA: ACM, 2004, S. 30 (siehe S. 77, 78).

[16] Klaus R Dittrich und Stella Gatziu. „Aktive Datenbanksysteme“. In: *Konzepte und Mechanismen, second ed., dpunkt. verlag, Heidelberg, Germany* (2000) (siehe S. 82).

[17] European Vending Association. *Telling the good story of coffee service and vending*. 2015. URL: http://www.vending-europe.eu/eva/an%5C_introduction%5C_to%5C_vending-update-november-2014.pdf (besucht am 22. 04. 2015) (siehe S. 16).

[18] Roy T Fielding und Richard N Taylor. „Principled Design of the Modern Web Architecture“. In: *ACM Trans. Internet Technol.* 2.2 (2002), S. 115–150 (siehe S. 47–50, 52, 54).

[19] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. Diss. 2000, S. 162 (siehe S. 11, 36, 40, 45–49, 52, 54).

[20] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. 2008. URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (besucht am 25. 06. 2015) (siehe S. 56).

[21] Roy Thomas Fielding, Mark Nottingham und Julian F Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. Techn. Ber. RFC 7234, June, 2014 (siehe S. 54).

[22] Roy Thomas Fielding u. a. *RFC 2616: Hypertext Transfer Protocol–HTTP/1.1*. Techn. Ber. 1999 (siehe S. 51, 53).

[23] Martin Fowler. *Richardson Maturity Model: steps toward the glory of REST*. 2010. URL: http://martinfowler.com/articles/richardsonMaturityModel.html (besucht am 01. 07. 2015) (siehe S. 57, 58).

[24] Gartner Inc. *Gartner Identifies the Top 10 Strategic Technology Trends for 2013 Newsroom Gartner Identifies the Top 10 Strategic Technology Trends for*. 2013. URL: http://www.gartner.com/newsroom/id/2867917 (besucht am 26. 04. 2015) (siehe S. 1).

[25] Gartner Inc. *Gartner Says 4.9 Billion Connected "Things"Will Be in Use in 2015*. 2014. URL: http://www.gartner.com/newsroom/id/2905717 (besucht am 26. 04. 2015) (siehe S. 2).

[26] Neil Gershenfeld. *Wenn die Dinge denken lernen*. Econ, 1999, S. 236 (siehe S. 7).

[27] Patrick Grässle und Markus Schacher. *Agile Unternehmen durch Business Rules: Der Business Rules Ansatz*. Springer-Verlag, 2006, S. 340 (siehe S. 78).

[28] Object Method Group. *Semantics of Business Vocabulary and Business Rules*. 2008. URL: http://www.omg.org/spec/SBVR/1.0/ (besucht am 24. 05. 2015) (siehe S. 77).

[29] Dominique Guinard. „A Web of things application architecture". Ph.D. ETH Zurich, 2011, S. 220 (siehe S. 11–13, 15, 47, 49, 52, 54, 59, 60, 62, 63).

[30] Dominique Guinard und Vlad Trifa. *Building the Web of Things*. Manning Publications Co., 2016, S. 375 (siehe S. 12).

[31] Dominique Guinard und Vlad Trifa. „Towards the Web of Things : Web Mashups for Embedded Devices". In: *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*. Madrid, Spain, Apr. 2009, S. 1–8 (siehe S. 10, 11, 46, 59, 60, 64).

[32] Dominique Guinard u. a. „5 From the Internet of Things to the Web of Things : Resource Oriented Architecture and Best Practices". In: *Architecting the Internet of Things*. Hrsg. von Dieter Uckelmann, Mark Harrison und Florian Michahelles. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 97–129 (siehe S. 10, 11, 46, 49, 59, 60, 63).

[33] Dominique Guinard u. a. „Interacting with the SOA-based internet of things: Discovery, query, selection, and on-demand provisioning of web services". In: *IEEE Transactions on Services Computing* 3.3 (2010), S. 223–235 (siehe S. 11).

[34] Barbara Von Halle. *Business Rules Applied: Building Better Systems Using the Business Rules Approach*. New York, NY, USA: John Wiley & Sons, Inc., 2001, S. 592 (siehe S. 78).

[35] Michael Jakl. *Representational State Transfer*. Techn. Ber. Vienna: TU Wien, 2005, S. 24 (siehe S. 45, 46, 49).

[36] Tim Kindberg u. a. „People , Places , Things : Web Presence for the Real World People , Places , Things : Web Presence for the Real World". In: *Mobile Networks and Applications* 7.5 (2001), S. 365–376 (siehe S. 10).

[37] Sriram Krishnan. *Programming Windows Azure*. O'Reilly Media, 2010, S. 345 (siehe S. 39).

[38] Badrinarayanan Lakshmiraghavan. *Pro Asp. Net Web API Security: Securing ASP. NET Web API*. Apress, 2013, S. 416 (siehe S. 73).

[39] Markus Lanthaler. „Third Generation Web APIs - Bridging the Gap between REST and Linked Data". Diss. TU Graz, 2014, S. 204 (siehe S. 56, 57, 67).

[40] Friedemann Mattern und Christian Flörkemeier. „Vom Internet der Computer zum Internet der Dinge". German. In: *Informatik-Spektrum* 33.2 (2010), S. 107–121 (siehe S. 6–8).

[41] Ingo Melzer. *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis*. 4. Aufl. Springer Spektrum, 2010, S. 382 (siehe S. 36).

[42] Microsoft. *Azure Web Jobs*. 2015. URL: https://azure.microsoft.com/en-us/documentation/articles/web-sites-create-web-jobs/ (besucht am 05. 06. 2015) (siehe S. 91).

[43] Valery Mizonov und Seth Manheim. *Azure Table Storage and Windows Azure SQL Database - Compared and Contrasted*. 2014. URL: https://msdn.microsoft.com/en-us/library/azure/jj553018.aspx (besucht am 25. 05. 2015) (siehe S. 39).

[44] Andreas Neuhauser. „Eine generische Web-of-Things Plattform für Smart Vending unter besonderer Betrachtung von Multi-Tenancy und Templates". Master Thesis. Johannes Kepler Universität Linz, 2015 (siehe S. 4, 65).

[45] J. Odell und J. Martin. *Object-Oriented Methods: A Foundation: UML Edition*. Prentice-Hall, 1998 (siehe S. 79).

[46] Cesare Pautasso und Erik Wilde. „Why is the web loosely coupled?" In: *Proceedings of the 18th international conference on World wide web - WWW '09*. Madrid, Spain, Apr. 2009, S. 911 (siehe S. 59).

[47] Cesare Pautasso, Olaf Zimmermann und Frank Leymann. „Restful Web Services vs. "Big“ Web Services: Making the Right Architectural Decision“. In: *Proceedings of the 17th International Conference on World Wide Web*. WWW '08. New York, NY, USA: ACM, 2008, S. 805–814 (siehe S. 56, 57).

[48] Klaus Pohl und Chris Rupp. *Basiswissen Requirements Engineering Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level*. ISQL-Reihe. dpunkt-Verlag, 2011, S. 176 (siehe S. 31).

[49] Leonard Richardson. *Richardson Maturity Model*. 2008. URL: http://www.crummy.com/writing/speaking/2008-QCon/act3.html (besucht am 01. 07. 2015) (siehe S. 57).

[50] Leonard Richardson, Mike Amundsen und Sam Ruby. *RESTful Web APIs*. O'Reilly Media, Inc., 2013, S. 379 (siehe S. 47, 49, 50, 52–54, 56, 57).

[51] Gerald Santucci. *The Internet of Things : Between the Revolution of the Internet and the Metamorphosis of Objects*. 2010. URL: http://cordis.europa.eu/fp7/ict/enet/publications%5C_en.html (besucht am 03. 04. 2015) (siehe S. 7).

[52] Michael Schrefl, Bernd Neumayr und Markus Stumptner. „The Decision-scope Approach to Specialization of Business Rules: Application in Business Process Modeling and Data Warehousing“. In: *Proceedings of the Ninth Asia-Pacific Conference on Conceptual Modelling - Volume 143*. Bd. Adelaide, APCCM '13. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2013, S. 3–18 (siehe S. 77, 78).

[53] S. Spreeuwenberg. „Business rules“. English. In: *Informatie*. Contributions to Management Science Oktober. Physica-Verlag HD, 2005, S. 51–88 (siehe S. 79).

[54] Manfed Steyer und Holger Schwichtenberg. *Moderne Webanwendungen mit ASP.NET MVC und JavaScript: ASP.NET MVC im Zusammenspiel mit Web APIs und JavaScript-Frameworks*. 2. Auflage. Köln: O'Reilly Media, 2014, S. 560 (siehe S. 41).

[55] Tops vending systems. *Sielaff Robimat*. 2015. URL: http://www.topsvending.be/en/products/sielaff-robimat (besucht am 28. 04. 2015) (siehe S. 16, 17).

[56] Kuldar Taveter und G. Wagner. „Agent-oriented business rules: Deontic assignments“. In: *Proc. of Int. Workshop on Open Enterprise Solutions: Systems, Experiences, and Organizations (OES-SEO2001), Rome, Italy*. 2001 (siehe S. 80).

[57] Kuldar Taveter und Gerd Wagner. „Agent-Oriented Enterprise Modeling Based on Business Rules“. In: *Conceptual Modeling — ER 2001 SE - 39*. Hrsg. von Hideko S.Kunii, Sushil Jajodia und Arne Sølvberg. Bd. 2224. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, S. 527–540 (siehe S. 77–79).

[58] Mihai Vlad Trifa. „Building Blocks for a Participatory Web of Things: Devices, Infrastructures, and Programming Frameworks“. Diss. ETH Zurich, 2011, S. 190 (siehe S. 59, 73).

[59] Dieter Uckelmann, Mark Harrison und Florian Michahelles. „An architeture approach towards the future internet of things“. In: *Architetureing the internet of things*. Hrsg. von Dieter Uckelmann, Mark Harrison und Florian Michahelles. Springer Berlin Heidelberg, 2011, S. 1–24 (siehe S. 2, 6–9).

[60] Mike (Microsoft) Wasson. *Single-Page Applications: Build Modern, REsponsive Web Apps with ASP.NET*. 2013. URL: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx (besucht am 28. 04. 2015) (siehe S. 42).

[61] Erik Wilde. „Putting Things to REST“. In: *Transport* 15.November (2007), S. 1–13 (siehe S. 11, 46, 56, 59, 60).