HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

WLAN PARTNER

Fog Computing

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühlingssemester 2019

Autoren:	Valentina Merturi, Stefano Kals
Betreuer:	Prof. Beat Stettler, Ins HSR
Auftraggeber:	Basile Bluntschli, WLAN-Partner.com AG
Experte:	Michael Schneider, Cloud Guard
Gegenleser:	Manuel Bauer, HSR

Abstract

In unserer Bachelorarbeit geht es hauptsächlich um den Bereich Fog-Computing. Dieser Begriff ist sehr gross und befasst sich mit der Datenverarbeitung, bevor die interessanten Daten in die Cloud gesendet werden.

Die Firma WLAN Partner aus Zürich arbeitet zusammen mit dem öffentlichen Verkehr an einem Projekt mit Postautos. Jedes Postauto soll mit einem Router ausgestattet werden und soll Echtzeit-Informationen zur Verfügung stellen. Da an diesem Router diverse Sensoren wie GPS, Odometer und Tür-Sensor angeschlossen sind, senden diese Sensoren ununterbrochen und willkürlich Daten an den Router. Unsere Aufgabe basierte deshalb darauf, genau diese relevanten Daten abzufangen. Zudem sollte es für den Benutzer (WLAN Partner) möglich sein, eigene Regeln zu definieren. Eine Regel könnte zum Beispiel sein, dass wenn die Position (via GPS-Signal) des Postautos sich in einem spezifischen Umfeld befindet, eine Aktion ausgelöst wird. Dies könnte in Form einer Werbung auf dem Smartphone sein.

Für die Realisierung des intelligenten Gateways stellt uns WLAN Partner einen netModule NB2800 Router zur Verfügung. Die Use Cases haben wir selbst und in Zusammenarbeit mit dem öffentlichen Verkehr erstellt, um unsere Arbeit darauf aufzubauen. Abgesehen davon, dass es sich um eine Linux-basierte Anwendung handelt, überliess uns der Kunde für die Implementierung die komplette Wahl der einzusetzenden Technologien.

Ziel dieser Arbeit ist es, dass der Benutzer die Daten jeder Zeit mitverfolgen und spezifische Regeln erstellen kann, um Geräte und deren Handhabung mit Daten dynamisch konfigurieren zu können. Der Benutzer soll die Möglichkeit haben jederzeit mit dem System des Postautos zu interagieren, beispielsweise bei einem überhitzten Motor. Mit dem Erstellen solcher Regeln und der Einsicht von Echtzeitdaten entstehen unzählige Anwendungsmöglichkeiten, welche wir als grossen Nutzen erachten.

Management Summary

Ausgangslage

Die Firma WLAN Partner aus Zürich arbeitet mit dem öffentlichen Verkehr zusammen an einem Projekt mit Postautos. Diese Postautos werden jeweils mit einem Router ausgestattet, an dem diverse Sensoren angeschlossen sind. Mit unserer Arbeit soll es möglich sein, diese Daten in Echtzeit auszuwerten und Regeln zu definieren, die gewünschte Aktionen auszulösen. Dies könnten Werbungen, Alarme oder auch Lautsprecher-Ansagen sein. Da die Sensoren ihre Daten einfach willkürlich an den Router senden und noch keine Benutzerschnittstelle existiert, soll ein intelligenter Gateway von uns entwickelt werden.

Vorgehen und Technologien

Wir haben für jeden Entwickler eine Entwicklungsumgebung eigenständig mittels Linux VM, Docker, Docker-Compose sowie den benötigten Abhängigkeiten aufgesetzt. Die ganze Implementierung wurde in Python3 umgesetzt. Zudem verwendeten wir Mosquitto als MQTT Broker und Redis als Datenbank. Das Zusammenspiel von Mosquitto und Redis war für die Umsetzung der eigenen Regeln von grosser Bedeutung. Ausserdem setzten wir die neuen yoyowallet Business-Rules ein. Damit war es für uns sogar möglich mehrere Sensor-Variablen miteinander zu vergleichen und in eine Bedingung zu setzen.

Ergebnisse

Zu Beginn benötigten wir relativ viel Einarbeitungszeit. Doch danach konnten wir viele Use Cases abdecken und die Regeln genauso implementieren, wie unser Kunde (WLAN Partner) sie einsetzen wollte. Zudem war diese Arbeit in Bezug auf den Überbegriff *Fog-Computing* ausserordentlich lehrreich, denn keiner von uns hatte vorher in einem solchen Bereich gearbeitet.

Ausblick

Der von uns entwickelte intelligente Gateway ist nun in einem Zustand, den man gut testen und weiterentwickeln kann. Die wichtigsten Funktionen konnten wir mittels Sensor-Simulationen testen. Nun ist aber der nächste Schritt, echte Sensoren anzuschliessen sowie die gewünschten und noch nicht umgesetzten Anwendungsfälle zu realisiert.

Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

Die Firma WLAN-Partner.com AG ist spezialisiert für sichere, performante und hochverfügbare Netzwerklösungen. Aktuell arbeitet sie mit dem öffentlichen Verkehr an einem gemeinsamen Projekt. Jedes Postauto soll mit einem netModule Router ausgestattet werden, um dem Benutzer (momentan unserem Kunden WLAN-Partner.com AG) Echtzeitdaten zur Verfügung zu stellen.

1.2 Zweck und Ziel

An diesem Router sind diverse Sensoren (wie GPS, Odometer, Thermometer, etc.) angeschlossen. Da diese Sensoren ununterbrochen Daten senden, die nicht immer von Bedeutung sind, müssen diese Daten für die Weiterverarbeitung intelligent aussortiert werden. Der Benutzer soll somit die Gelegenheit haben, die Echtzeitdaten auszuwerten oder anhand dieser gewünschte Aktionen auszuführen.

1.3 Bereitgestellte Hardware

Der Kunde hat uns einen netModule NB2800 Router zur Verfügung gestellt. Dieser enthielt eine Standard-Konfiguration, jedoch keine Lizenz für die Virtualisierung.

Abbildung 1.1: netModule NB2800 Router

1.4 Lieferumfang

Der Kunde erhält den netModule NB2800 Router zurück. Zudem wird unsere Entwicklung der Rule Engine in diesem Dokument detailliert festgehalten, damit der Kunde ohne Probleme weiterentwickeln kann. Ausserdem gibt es für die Entwicklungsumgebung, das Deployment sowie für die Anwendung der Rule Engine jeweils Anleitungen als separate Dokumente.

1.5 Annahmen und Einschränkungen

Da wir bei der Wahl der einzusetzenden Technologien auf uns selbst gestellt waren und der Router Linux-basiert war, gab es für uns nur eine einzige Einschränkung. Die ausgewählten Technologien mussten Linux-kompatibel sein.

1.6 Umfang

Gemäss den Anforderungen des Kunden gab es bei der Rule Engine nicht nur Regeln zu erstellen, sondern auch entsprechende Filter und diverse Funktionen zu realisieren. Da wir aber in Bezug auf die Implementierungszeit eingeschränkt waren, mussten wir den Umfang etwas eingrenzen. In erster Linie galt es in dieser Arbeit ein Konzept auszuarbeiten, wie eine solche Rule Engine funktionieren soll. Der Kunde erwartete kein fertiges Produkt, welches er am Ende direkt einsetzen kann, sondern ein Konzept. Dieses Konzept soll möglichst so ausgearbeitet sein, dass die gewünschten Daten ausgelesen und einige kundenspezifische Regeln funktionieren. Zudem muss alles genaustens Dokumentiert werden, damit einer Weiterentwicklung nichts im Weg steht.

2 Aufgabenstellung

Die ausgestellte Aufgabenstellung war sehr generell und breit gestreut. Daraufhin mussten wir uns ein paar Mal mit dem Kunden treffen, um den definitiven Projektumfang festzulegen. Die folgende Aufgabenstellung beschreibt den ausgearbeiteten Umfang und Fokus unseres Projekts.

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

Aufgabenstellung Bachelorarbeit
Fog-Computing

1 Betreuer und Experte

Diese Arbeit wird mit dem Praxispartner WLAN -Partner.com AG durchgeführt und von Prof. Beat Stettler betreut.

Industriepartner

WLAN-Partner.com AG
Stauffacherstrasse 16
8004 Zürich

Ansprechperson

Basile Blunschli
Engineering / Betrieb

2 Studierende

Diese Arbeit wird als Bachelorarbeit in der Abteilung Informatik durchgeführt von

- Valentina Merturi
- Stefano Kals

3 Ziel der Arbeit / Aufgabenstellung

Im IoT Umfeld sammeln immer mehr Sensoren unendlich viele Messwerte, die verarbeitet werden müssen. Dies führt bei den zentralen Analyse Systemen in der Cloud zu immensen Datenansammlungen - davon meist ein grosser Teil uninteressanter Datenfriedhof - und dadurch zu langen Verzögerungen bei der Analyse und Auswertung. In dieser Arbeit soll deshalb die Analyse der Sensordaten weg von der Cloud, näher zu

1

den Sensoren gebracht werden, so dass schlussendlich nur noch "interessante" Daten den Weg in die Cloud finden. Dazu werden Funktionen zur Voranalyse auf einem intelligenten Gateway (zwischen Cloud und Sensoren - daher Fog) verschoben.

Die Möglichkeiten der Vorverarbeitung sind vielfältig: Angefangen bei der Überprüfung von Schwellwerten, über Histogramme bis zur Korrelation von Datenströmen verschiedener Sensoren. Wird z.B. ein Schwellwert überschritten oder bewegt sich der Sensor in eine Geofence Zone, so muss eine Aktion ausgeführt werden können. Eine solche Aktion könnte beispielsweise eine Einspielung von Werbung sein.

Nach der Einarbeitung in das Thema Fog Computing sollen existierende Lösungen/Frameworks gesucht und evaluiert werden. Anschliessend wird ein Konzept mit einem Architekturentwurf zur Umsetzung erarbeitet und ein Prototyp entwickelt, mit welchem erste Tests durchgeführt werden können. Als Hauptkomponente wird eine intelligente Rule Engine entwickelt. Mit dieser Rule Engine ist es möglich, verschiedene Sensoren anzusprechen und deren Daten mittels dynamisch einspielbarer Konfiguration oder benutzerspezifischen Regeln zu verarbeiten, zu filtern, vorzuverarbeiten oder auch zu aggregieren. Optional soll es ebenfalls möglich sein, verschiedene echte Sensoren, die an das Gateway angeschlossen werden, anzusprechen.

Als Hardware für die kundenseitige Umsetzung dient ein netModule NB2800 MultiVehicle+ Router mit Embedded-Hardware auf ARM Basis und einem Linux-Betriebssystem. Für den Prototyp kann aber auch z.B. ein Rasperry Pi verwendet werden. Durch die bedingte Leistungsfähigkeit dieser Hardware Plattformen müssen die Performance-Anforderungen des Prototypen möglichst klein gehalten werden.

Neben den üblichen Software Entwicklungsdokumentationen soll eine (für nicht Entwickler verständliche) Anleitung der Entwicklungsumgebung und der verwendeten Komponenten, sowie ein Benutzerhandbuch zur Verwendung der Engine erstellt werden. Voraussetzung für die Arbeit ist gutes Know-How im Umgang mit Linux sowie Python Basiskenntnisse.

4 Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die zu erstellenden Dokumente bzw. Berichtsteile sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form festhalten.

5 Termine

Siehe Terminplan auf dem Skripte-Server

6 Beurteilung

Die Beurteilung erfolgt gemäss den vorgegebenen Richtlinien. Für die Beurteilung ist der verantwortliche Dozent zuständig.

Im Übrigen gelten die Bestimmungen der Abteilung Informatik für Bachelorarbeiten.

Ort, Datum: Rapperswil, 7.6.2019

Prof. Beat Stettler

3 Hintergrund

3.1 Internet of Things

Das IoT ist ein derzeit erfolgreicher Technologie-Trend, der überall auf der Welt diskutiert wird. Der Begriff *Internet of Things* wurde erstmals in einer Präsentation im Juni 1999 eingesetzt. In dieser Präsentation kombinierte Kevin Ashton die Radiofrequenz-Identifikation (RFID) mit dem zu diesem Zeitpunkt aufkommenden Internet - woraus der Begriff *Internet of Things* entstand. Die Vision des IoTs ist es, grosse Mengen an intelligenten Objekten mit dem Internet zu verbinden. Diese Objekte oder Gegenstände können von Sensoren, Aktoren, Warables, Mobiltelefone über Einplatinencomputer bis hin zu Mikro-DCs reichen. Sie werden auch IoT-Geräte genannt. Um diese Geräte mit dem Internet zu verbinden, werden Technologien wie WiFi, Bluetooth, RFID und Nahfeldkommunikation (NFC) eingesetzt. RFID und NFC werden zum Identifizieren, Verfolgen und Authentifizieren von Geräten genutzt.

Rückblickend zeigt sich, dass die RFID Technologie die erste war, die die Aufmerksamkeit auf die IoT-Welt gelenkt hat. Auch heute ist diese Technologie eine der stärksten Triebkräfte diesbezüglich. Mittels sogenannten RFID-Tags hat man die Möglichkeit Verbindungen zu den Geräten herzustellen. Diese sind einfach anwendbar, klein, eindeutig identifizierbar und beinhalten programmierbare Mikroprozessoren, die mit einer Antenne verbunden sind. Zusätzliche Komponenten wie Batterien oder GPS-Sensoren können je nach Anwendungsspezifikation hinzugefügt werden. Solche Tags können auf allen erdenklichen Gegenständen (Lebensmittelverpackungen, versandfähige Produkte, Tiere oder menschliche Tiere) platziert werden. Gängige Beispiele für solche getaggten Anwendungen sind Bibliotheks-Systeme, Logistik, Produktverfolgung (Post) und Zugangskontrollsysteme. Die Nachfrage nach solchen Tags ist sehr hoch, da man das Ziel verfolgt, möglichst viele Alltagsgegenstände zu erreichen. Jedoch wird aufgrund der riesigen Anzahl von Tags auch der Preis zu einem entscheidenden Interessenpunkt.

Neben RFID ist der Ansatz von Wireless Sensor Networks (WSNs) eine interessante verwandte Technologie bezüglich dem zukünftigen Trend des IoT. Obwohl WSN und das IoTs konkurrierende Technologien zu sein scheinen, gibt es viele Sensornetzwerke in WSN, die aus drahtlosen, miteinander verbundenen und kommunikationsfähigen Sensoren bestehen. Sensoren müssen andere Sensoren in unmittelbarer Nähe entdecken und ein Netzwerk aufbauen, um in einem hochgradig unproblematischen Umfeld miteinander zu kommunizieren. Gänginge Beispiele für solche WSNs sind Wärme- und Rauchmelder, die entweder eine Meldung als Check auslösen oder die direkt bei einem Notruf bei der Feuerwehr

Alarm auslösen.
Ein weiteres Technologiekonzept, welches derzeit in fast jedem Unternehmen diskutiert wird, ist die Industrie 4.0. Das IoT ist mit diesem Konzept eng verbunden, denn die Industrie 4.0 verfolgt das Ziel der Digitalisierung und Automatisierung von Herstellungsprozessen - beginnend beim ersten Entwurf eines Produkts bis hin zur kompletten Fertigung. Dazu gehören die Bereiche Big Data Analytik, Digitalisierung, Fertigung, Selbstanpassung sowie die künstliche Intelligenz und vieles mehr.

Aktuell besteht das IoT nicht nur aus Sensoren und Aktoren, sondern auch aus vielen heterogenen Geräten, die mit dem Internet verbunden werden. Dies unterstreicht die Notwendigkeit einer standardisierten Kommunikationstechnik zwischen den IoT-Geräten, welche man kontrollieren möchte. Zusätzlich müssen jedoch die Energieeffizienz und Rechenressourcen berücksichtigt werden. Diese Anforderungen und Anwendungsmöglichkeiten ziehen Akademien und Industrien an, die in die zukünftigen Forschungen des IoT investieren. Diverse Unternehmen versuchen damit die eigenen Ziele zu realisieren. Die Vorteile dieser IoT-Geräte sind die geringeren Kosten für Verarbeitungs- und Speicherleistungen die Verbreitung von IoT-Geräten, die elastische Big Data-Analyse in der Cloud und die Konvergenz der Daten. Darüber hinaus gewinnt man unfassbar viele verschiedene Einsatzmöglichkeiten wie zum Beispiel Smart Cities, Smart House, Smart Cars, Smart Grids, Smart Health Care, Smart Logistigs, autonome Autos und Roboter. In Bezug auf Smart Homes kommt noch dazu, dass man das Haus komplett über ein Smartphone, Tablet oder ähnliche Geräte steuern kann. So ist es möglich ein Fenster zu schliessen oder die Termperatur zu ändern ohne im Haus anwesend zu sein.

Man hat das Ausmass der Verbreitung von solchen IoT-Geräte prognostiziert und kam auf das Ergebnis, dass es bis zum Jahr 2020 ungefähr 50 Milliarden Smart Objects haben wird und dass dieser Sektor sich ausserordentlich stark entwickeln wird.
Im Allgemeinen ist klar, dass IoT eine vielversprechende Technologie ist, die viel Potential aufweist und das Interesse vieler an sich zieht. Die verschiedenen Einsatzmöglichkeiten verschaffen der Menschheit viele Vorteile und stärkt die Technologie gewaltig. Somit werden nicht nur einzelne Produkte oder Industrien beeinflusst, sondern die komplette Sichtweise von Systemen und Anwendungen wird verändert.

3.2 Cloud Computing

Cloud Computing ist ein umfassend erforschtes, zentralisiertes Computing-Paradigma, mit dem Schwerpunkt auf dynamische Bereitstellung von Rechen- und Speicherressourcen. Diese Ressourcen befinden sich in zentralen DCs. Die Auswahl des DC-Standortes ist an mehreren Faktoren gebunden wie zum Beispiel die Umgebungskosten, Temperaturen und Bodenpreise.

Zu den von Cloud-Anbietern bereitgestellten Ressourcen gehören Software-Services, Brow-

ser sowie Entwicklerplattformen zur Erstellung von Cloud-Anwendungen und zur Fertigung von Serverinfrastrukturen. Diese Servicemodelle der Bereitstellung von Cloud-Ressourcen werden als Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) und Infrastructure-as-a-Service(IaaS) bezeichnet. Aus diesen drei unterschiedlichen Servicemodellen sind folgende vier verschiedene Cloud-Bereitstellungsmodelle entstanden: Private Cloud, Community Cloud, Public Cloud und Hybrid Cloud.

Um grosse Mengen an On-Demand-Ressourcen auf Pay-per-user-Basis Cloud-Anbietern bereitzustellen, wird diese Virtualisierungstechnologie genutzt.

3.2.1 Eigenschaften

Die wesentlichen Merkmale von Cloud-Computing wurden bei der Definition von NIST eingeführt und decken immer noch die wichtigsten Aspekte ab.

On-Demand Self-Service Cloud-Dienste können ohne Personaleinsatz on-demand bereigestellt werden.

Broad Network Access Da die Cloud Computing DCs überall auf der Welt verteilt sind, kann jedes mit dem Internet verbundene Gerät mit der Cloud interagieren. Heutzutage sind viele Smart Geräte über Cloud-Services mit dem Internet verbunden, da viele Service-Provider bereits zentralisierte Cloud-Ressourcen nutzen (Beispiel: Netflix, Airbnb, Slack, etc.).

3.2.2 Servicemodelle

Die Servicemodelle beschreiben, wie in der Cloud-Umgebung befindliche Cloud-Ressourcen erfasst und genutzt werden können. Der Standort und das Eigentum an Ressourcen werden bei diesen Differenzierungen nicht berücksichtigt.

SaaS Dieses Modell (Software-as-a-Service) ist das erste Modell und die am stärksten eingeschränkte Möglichkeit. SaaS-Kunden greifen entweder über eine Weboberfläche mit einem Webbrowser oder einer Programmierschnittstelle auf den Service zu. Die physischen Cloud-Ressourcen des Dienstes können vom Kunden nicht genutzt oder speziell konfiguriert werden. Beispiele dafür sind Microsoft Office 365 und Google Apps.

PaaS Ein flexibleres Modell, welches sich auf Entwickler spezialisiert hat, ist Platform-as-a-Service. Hiermit wird ein Entwicklungsrahmen bereitgestellt, der es Entwicklern ermöglicht, Anwendungen in der Cloud-Umgebung zu entwickeln, zu testen und bereitzustellen. Mit Blick auf eine flexible Softwareumgebung befasst sich die Cloud mit der Verwaltung und Kontrolle der benötigten Ressourcen. Neben der allgemei-

nen Verwaltung und Kontrolle der Ressourcen, übernimmt die Cloud die dynamische Skalierung, um sicherzustellen, dass die vereinbarten SLA's zu keinem Zeitpunkt verletzt werden. Zu solchen Modellen gehören Google App Engine4 und Windows Azure5.

IaaS Dieses Modell (Infrastructure-as-a-Service) ist das flexibelste Modell. Der Kunde hat hier die volle Kontrolle über die VM, die auf physischen Cloud-Ressourcen läuft. Diese kann mit den erforderlichen Spezifikationen als RAM-, CPU-, Speicher- und Netzwerkfunktionen konfiguriert werden. Auf dieser VM ist der Benutzer in der Lage, jede Art von Anwendungen auf einem ausgewählten Betriebssystem bereitzustellen. Dennoch hat der Benutzer nicht die volle Kontrolle über die physischen Ressourcen, sondern nur über die erworbene VM. Bekanntermassen zählen Amazon Web Services und Open Stack zu diesen Modellen.

3.2.3 Deployment Modelle

Die soeben beschriebenen Servicemodelle werden in verschiedene Depolyment Modoelle ausgerichtet.

Private Cloud Die private Cloud ist ein Ansatz, bei dem ein Unternehmen eine Cloud-Umgebung verwendet, die explizit für genau dieses Unternehmen zugeschnitten ist. Der Besitzer, die Verwaltung und der Betrieb von Cloud-Ressourcen können vom Unternehmen selbst, von einem Drittunternehmen oder von einer Kombination aus beiden übernommen werden. Der Vorteil dieses Modells besteht darin, dass die an die private Cloud gesendeten Daten nur in der Umgebung gespeichert werden, die ausschliesslich für diese spezifische Organisation verwendet wird. Sensible Daten werden nicht mit einer anderen Organisation geteilt, die mit der Umgebung verbunden ist. Wenn diese spezifische private Cloud ausfällt, können Daten in diesem Single Point of Failure gesperrt werden.

Community Cloud Die Community-Cloud basiert auf der gleichen Idee wie die der privaten Cloud. Die Cloud-Umgebung ist jedoch für die Nutzung innerhalb einer bestimmten Community vorgesehen. Auch das Eigentum, die Verwaltung und der Betrieb können von einer oder mehreren Organisationen durchgeführt werden - die die Gemeinschaft bilden.

Public Cloud In einer öffentlichen Cloud ist die Cloud-Umgebung nicht für eine bestimmte Organisation oder Community bestimmt. Somit kann es von der Allgemeinheit genutzt werden. Das Eigentum, die Verwaltung und der Betrieb können entweder von der Regierung, akademischen Organisationen, jeder anderen Organisation oder einer Kombination all die-

ser Möglichkeiten übernommen werden. Der Vorteil der öffentlichen Cloud ist die offene Zugänglichkeit für alle, ohne Einschränkungen für bestimmte Unternehmen. Die negativen Auswirkungen der Open Accessibility sind daher die Daten- und Servicesicherheit.

Hybrid Cloud Die hybride Cloud ist eine Kombination aus einer privaten und einer öffentlichen Cloud. Daher können einige Cloud-Ressourcen in einer privaten Cloud gehalten, verwaltet und betrieben werden, während andere eine offen zugängliche Cloud nutzen können. So könnten beispielsweise sensible Daten in einer privaten und allgemein unkritische Daten in der öffentlichen Cloud gespeichert werden.

3.2.4 Virtualisierung

Die Virtualisierung beschreibt die Abstraktion der physischen Hardware-Ressourcen eines Computersystems. Das Konzept der Virtualisierung ist eine der wichtigsten Antriebskräfte von Cloud Computing aus folgenden zwei Gründen.

1. Die physische Hardware kann auf isolierte Weise für mehrere Kunden gemeinsam genutzt werden. So sind die Vorteile, dass ein einziges physisches Hardwaresystem mehrere Kunden als Grundlage für ihre Verarbeitungs- und Speicheranforderungen bedienen kann, wobei die meisten Hardware-Ressourcen der Maschine genutzt werden. Dies ermöglicht eine effiziente Nutzung der vorhandenen Ressourcen und eliminiert die Sperrung einer kompletten physischen Maschine für einen einzelnen Kunden.

2. Können heterogene Hardwarekomponenten zu einem homogenen Ressourcenpool zusammengeführt werden. Dieser Ressourcenpool-Mechanismus wird verwendet, um verschiedene VM-Einstellungen anzubieten (CPU, RAM und Speicherkapazitäten).

Es gibt verschiedene Virtualisierungskonzepte, aber wir beschränken uns hier auf die vollständige Virtualisierung und die Betriebssystem-Virtualisierung.

Vollständige Virtualisierung

Die Abstraktion einer vollständigen Virtualisierung führt zu einer virtuellen Kopie der physischen Ressourcen, die in einer VM verpackt sind. Von der Bereitstellung einer VM muss der Benutzer die Anzahl der CPU's, des RAM's und andere Funktionen angeben, welche die VM besitzen soll. Die Anzahl der auf einer Maschine erstellten VM's ist durch ihre physischen Ressourcen begrenzt. Da die auf den physischen Ressourcen laufenden VM's über eine koordinierende Middleware (Virtual Machine Monitor oder VMM) vollständig isoliert sind, bleiben die Systeme vollständig getrennt.

Die vollständige Virtualisierung von Hardware-Ressourcen hat das Ziel, eine isolierte Umgebung zu erreichen. Die Hardware wird von dem erwähnten VMM verwaltet. Das VMM ist für die Überwachung, Verwaltung und Bereitstellung von darüber liegenden VM's verantwortlich und verhindert den direkten Zugriff auf die physische Hardware. Jede VM besteht aus einem separaten Betriebssystem (einem Gastbetriebssystem und beliebigen Anwendungen), die darauf laufen. Die Vorteile dieser Art der Virtualisierung der physischen Hardware-Ressourcen sind die Flexibilität bei der Bereitstellung, einfache Bereitstellung anpassungsfähiger Ressourcen und die Möglichkeit, heterogene Hardware zusammenzustellen. Die negativen Aspekte dieses spezifischen Virtualisierungskonzepts sind die langen Startzeiten und der grosse Speicherplatzbedarf. Beide negativen Aspekte werden durch die Notwendigkeit eines getrennten Betriebssystems in jeder VM verursacht. Ein bekanntes Unternehmen, das Virtualisierungslösungen entwickelt, ist VMWare.

Betriebssystem-Virtualisierung (Container)

Im Gegensatz zum zuvor genannten Virtualisierungskonzept ist die Betriebssystemvirtualisierung (oder Container-Virtualisierung) ein sehr leichtes Konzept. Dieses Konzept baut auf dem bestehenden Host-Betriebssystem auf. Die logisch getrennten, virtuellen Umgebungen (d.h. Contaienr) laufen auf dem gleichen Host-Betriebssystem und verwenden den gleichen Kernel und die allgemeine physische Hardware. Die virtuelle Umgebung ist jedoch vollständig isoliert. Mit anderen Worten, der Container kann nur auf Speicherplätzen und Prozesse zugreifen, die seinem eigenen Container zugeordnet sind.

Eine der am häufigsten verwendeten Lösungen für die Containertechnologie ist Docker [1]. Docker unterscheidet zwischen Docker Images und Docker Containers. Ein solches Image besteht aus mehreren Ebenen, die einen Snapshot eines solchen Containers speichert. Docker verwendet das Union-Dateisystem, um diese Ebenen zu einem Image zusammenzuführen. Dieses Image kann dann über die Docker-Laufzeit instanziiert werden und führt zu einem lauffähigen Container. Der Container besteht aus einem Basis-Image (Lightweight-Betriebssystem), benutzerdefinierten Dateien und Metadaten. Dieser Docker-Container kann gestartet oder gestoppt werden und die Images-Snapshots kann man speichern und wiederverwenden. Weitere Orchestrierungs-Werkzeuge zur Verbesserung der Fähigkeiten einfacher Docker Container sind beispielsweise Docker Machine [2], Docker Swarm [3] und Docker Compose [4].

Docker Machine Docker Machine [2] hilft bei der Bereitstellung von Docker-Engines auf dem lokalen Computer, Cloud-Anbietern oder anderen DC's.

Docker Swarm Docker Swarm [3] wird verwendet, um Docker Container zu bündeln.

Docker Compose Docker Compose [4] hilft ein verteiltes System zu betreiben, das aus mehreren Docker Containern besteht.

Eine weitere häufig verwendete Container Technologie sind die Linux Container (LXC). LXC

[5] bieten ein Verfahren zur Virtualisierung auf Betriebssystemebene, was ebenfalls mehrere voneinander isolierte Linux-Systeme auf einem einzigen Host ermöglichen. LXC erzeugt dabei eine virtuelle Umgebung mittels *namespaces* mit eigenen, untereinander nicht sichtbaren, Prozessen jedoch geteiltem Kernel. Sogenannte *cgroups* sorgen dabei für eine Ressourcenverteilung.

Der Vorteil von Containern gegenüber VMs besteht darin, dass Container im Handumdrehen gestartet werden können, da sie nicht für jeden Container ein eigenes Betriebssystem benötigen. Container sind leichtgewichtig und können sehr schnell bereitgestellt, freigegeben und aktualisiert werden, da auf Grund des speziellen Union-Dateisystems nur bestimmte Layer aktualisiert werden müssen. Im Falle eines Fog-Computing Frameworks passen Container daher besser zu den Anforderungen. Docker wird häufig in Kombination mit der derzeit florierenden Micro Service Architecture (MSA) eingesetzt. Die MSA ist ein neuartiger Ansatz zur Entwicklung kleiner, leichter, unabhängiger und verteilter Softwarekomponenten. Eine separate Softwarekomponente heisst Microservice und wird in einem unabhängigen Container bereitgestellt. Einige der wichtigsten Vorteile davon sind folgende: Technologieheterogenität, Resilienz, Skalierbarkeit, einfache Bereitstellung und Optimierung für den Austausch. Da Microservices über standardisierte Formate wie JavaScript Object Notation (JSON) und Extensible Markup Language (XML) kommunizieren, sind die Technologien völlig unabhängig. Die in MSAs verwendete Standardkommunikationstechnologie ist der Representational State Transfer (REST).

3.3 Fog Computing

In den letzten Jahrzehnten wechselten die Berechnungsmodelle zwischen einem zentralen und einem dezentralen Rechenansatz hin und her. In den 90er Jahren folgte eine Welle der Dezentralisierung durch das Client-Server-Modell. Diese erste Welle wurde durch die sinkenden Preise für Personalcomputer und das steigende Interesse an proprietärer Rechenleistung ausgelöst. Anfang der 2000er Jahre wechselte das Rechenmodell wieder von einem dezentralen zu einem zentralen Ansatz der Cloud Computing-Paradigma. Obwohl Cloud Computing floriert und in naher Zukunft nicht ersetzt werden wird, drängt eine Kraft auf einen neuartigen dezentralen Ansatz, um die immanenten Probleme zentralisierter Systeme zu lösen, wie hohe Latenzzeiten und fehlende Standortwahrnehmung. Der Unterschied zum vorhergehenden Paradigma besteht darin, dass dieses sich nähernde Paradigma das frühere nicht ablösen, sondern erweitern wird, um spezifische Fähigkeiten zu verbessern. Dieser gegenwärtige Wandel vom zentralen Cloud Computing Paradigma hin zu einem Berechnungsparadigma markierte die Geburtsstunde des Fog-Computing.

Fog-Computing wurde 2012 als neue Technologie für das IoT in der USA eingeführt. Dieses neuartige verteilte Computing Paradigma umfasst die Idee, Rechen- und Speicherfähigkeiten näher an datenproduzierende IoT-Geräte am Rande des Netzwerks bereitzustellen. Mit dem Ziel, den Abstand zwischen den Endgeräten und der nächsten Verarbeitungsein-

heit zu verringern, führt dieses Paradigma eine zusätzliche Schicht von ressourcenreichen IoT-Geräten (Fog-Cells) ein. Diese Fog-Cells verfügen über eigene Rechen- und Speicherfähigkeiten, um Aufgabenanforderungen zu verarbeiten, Daten zu filtern und vorzubereiten. Dies erstellt einen Abstand von einem Schritt zu den Endgeräten und verkürzt somit die Latenz- und Ausführungszeit der Aufgabe. Eine weitere entscheidende Erweiterung des Fog Computing ist die hohe geografische Verbreitung dieser zusätzlichen Geräte, die auf eine nahtlose und zuverlässige Serviceabwicklung (auch bei Anschluss an bewegliche Geräte z.B. Smart Cars, Mobiltelefone) abzielt.

Genauer gesagt befinden sich die Fog-Cells am Rande des Netzwerks, um die Latenz und Ausführungszeit der Aufgaben und die Menge der über das Netzwerk gesendeten Daten zu reduzieren. Fog-Cells empfangen dann Aufgabenanforderungen von angeschlossenen IoT-Geräten (zum Beispiel Sensoren und Aktoren oder externen Initiatoren) und entscheiden, wo diese Anfrage verarbeitet werden soll. Abhängig von den Fähigkeiten der Fog-Cells entscheidet eine Argumentationskomponente, ob die Aufgabenaufforderung lokal verarbeitet oder in die Cloud übertragen wird. Die Kommunikation innerhalb der Fog Computing-Umgebung (der Fog Computing-Umgebung) ist nicht auf irgendeine Form der Kommunikation beschränkt und kann daher über WiFi, Mobilfunknetze, Bluetooth oder Ethernet erfolgen. In Richtung einer redundanten Umgebung kann diese Fog Computing-Umgebung auch mit mehreren Cloud-Anbietern verbunden oder innerhalb privater Clouds nutzen werden.

Am unteren Ende der Architektur befinden sich verschiedene IoT-Geräte, gefolgt von Routinggeräten, die mit Fog-Cells verbunden sind. Diese Routing-Vorrichtungen symbolisieren die Möglichkeit, bestehende Netzwerkgeräte (z.B. Router) wiederzuverwenden. Darüber hinaus können Nebelzellen mit öffentlichen oder privaten Clouds verbunden werden.

3.3.1 Eigenschaften

Die Hauptmerkmale des Fog-Computing-Paradigmas, das dem IoT hilft, sein Potential zu nutzen, sind nachfolgend aufgelistet.

Niedrige Latenzeit und Standortwahrnehmung

Da Fog-Cells, die sich am Rande des Netzwerks befinden und den Abstand zwischen IoT-Geräten und Cloud-Ressourcen verringern, können die Latenzzeiten und die Ausführungszeit der Aufgaben drastisch reduzieren. Darüber hinaus können auf Grund der Standortwahrnehmung von Fog-Cells standortbezogene Dienste bereitgestellt werden. Diese Dienste sind in der Lage, angeforderte Inhalte vorzuverarbeiten, zu filtern oder zwischenzuspeichern.

Hohe geographische Verbreitung

IoT-Geräte sind stark geographisch verteilt. Deshalb ist der Wechsel von der zentralen Verarbeitung in der Cloud zu einer dezentralen Verarbeitung in einer Fog Computing-Umgebung

notwendig.

Grossflächige Sensornetzwerke

Grossflächige Sensorenetze, die mit Fog-Cells kommunizieren, sind eines der Schlüsselszenarien des Fog Computing-Paradigmas. Diese Sensoren sind in der Lage, Aufgabenanforderungen an Fog-Cells zu senden. Abhängig von den verfügbaren Fog-Cells-Ressourcen verarbeitet er die Anfragen entweder selbst oder verbreitet sie zur weiteren Verarbeitung an andere Fog-Cells.

Mobiliätsunterstützung

Neben der hohen Verbreitung von IoT-Geräten ist auch die Mobilität der beteiligten Endgeräte zu berücksichtigen. Bewegliche Geräte erfordern eine dynamische Umstrukturierung der Netzwerktopologie entsprechend den betroffenen Geräten in der Fog Computing-Umgebung. Dies wird durch eine dynamische hierarchische Fog Computing-Umgebung ermöglicht, um Topologien neu zu strukturieren. Mit dem Ziel einer hohen Mobilitätsunterstützung muss die Fog-Computerlandschaft in der Lage sein, mit mobilen Geräten durch mobile Kommunikationstechnologien zu kommunizieren, wie beispielsweise LISP.

Geräteheterogenität

Diverse IoT-Geräte, die an das Netzwerk angeschlossen sind, verfügen über keine standardisierten Funktionen, Schnittstellen oder Einsatzmöglichkeiten. Daher müssen viele heterogene IoT-Geräte bei der Bearbeitung von Anfragen in einer Fog Computing-Umgebung berücksichtigt werden. Die Fog Computing-Umgebung zielt darauf ab, die Kommunikation zwischen verschiedenen Arten von IoT-Geräten durch standardisierte Kommunikation und Ressourcenvirtualisierung zu ermöglichen. Als sehr aktuelles Forschungsthema stösst Fog Computing auf einen Mangel an Methoden und konkreten Lösungen. Daher ist noch viel Forschungsaufwand erforderlich, um die oben genannten Merkmale zu ermöglichen. Ein wichtiger Verein, der die Vision von Fog Computing unterstützt und verbreitet, ist das Open Fog Consortium. Das Prinzip des Open Fog Consortium ist es, eine offene Fog-Computerarchitektur zu erhalten, um die kooperative Forschung mit mehreren Organisationen zu ermöglichen. Die Zusammenarbeit verschiedener Organisationen, die auf verschiedene Bereiche spezialisiert sind, ist ein entscheidender Punkt für die Zukunft des Fog Computing.

Fog Computing wird oft als Bereitstellungstechnologie für verschiedene Anwendungen, aus verschiedenen Bereichen erwähnt, wie zum Beispiel das Gesundheitswesen, Caching und Preprocessing. Um die vielversprechenden Verbesserungen, die sich aus diesem Paradigma ergeben, sicherstellen zu können, sind noch viele Herausforderungen zu bewältigen. Zu den zentralen Herausforderungen in der Forschung gehören Ressourcenbereitstellung, Serviceplatzierung, Sicherheit und Zuverlässigkeit, Energieminimierung, Standardisierung und Programmiermodelle. Neben dem Fog Computing haben sich zwei leicht un-

terschiedliche Paradigmen namens Mobile Edge Computing (MEC) und Mobile Cloud Computing (MCC) entwickelt.

3.3.2 Mobile Edge Computing

MEC basiert auf einer vergleichbaren Idee wie Fog Computing, legt aber den Schwerpunkt auf mobile Geräte am Rande des Netzwerkes. Die Hauptmerkmale der gesamten MEC Umgebung sind die Aufgabenteilung zwischen Geräten und die Aufgabenverteilung in die Cloud. Die mobilen Geräte sind in der Lage miteinander zu kommunizieren und Aufgabenanforderungen voneinander entgegenzunehmen. Zusätzlich befinden sich am Rande des Netzwerks sogenannte MEC-Server, die für die Verbindung der mobilen Geräte mit der Cloud zuständig sind. Bei diesem Ansatz ist der Rand des Netzwerks der Rand des Radio Access Networks (RAN). Somit repräsentieren die Basisstationen des RANs die MEC-Server, d.h. die Fog-Cells im Fog-Rechnen. Diese MEC-Server sind im Besitz von Telekommunikationsunternehmen und können Aufgabenanforderungen verarbeiten. Da der Grossteil der Kommunikation im RAN stattfindet, sind die betonten Kommunikationstechnologien im MEC die Mobilfunkt-Generationen 3G, 4G und 5G.

Eine der wichtigsten treibenden Kräfte bei der Unterstützung der Forschung im Bereich der MEC, ist das European Telecommunications Standards Institute (ETSI). Diese treibende Kraft ist natürlich mit der starken Nutzung der Telekommunikationsressourcen und weiteren Möglichkeiten für die Telekommunikationsindustrie verbunden. Zusammenfassend ist das Hauptziel von MEC die Reduzierung der Netzwerklatenz und damit die Verbesserung der Qualität mobiler Anwendungen, wie Augumented und Virtual Reality, On-Demand-Videostreaming und Mobile Gaming.

3.3.3 Mobile Cloud Computing

MCC ist aus den bereits gut erforschten Technologien Mobile Computing und Cloud Computing hervorgegangen. Die treibende Kraft dieser Technologie ist eine ressourcenreiche Cloud Computing Umgebung in Kombination mit der Verbreitung von intelligenten mobilen Geräten. Obwohl die Rechenleistung moderner mobiler Geräte steigt, gibt es immer noch keine zufriedenstellende Lösung hinsichtlich der Akkulaufzeit und der Datenspeicherkapazität. Ressourcenintensive Aufgaben wie Bilderverarbeitung und natürliche Sprachverarbeitung müssen daher in der Cloud verarbeitet werden, um Energie und Speicherkapazität zu sparen. Da Mobiltelefone mit ungenutzten Verarbeitungseinheiten jedoch allgegenwärtig sind, ist die effiziente Nutzung dieser Ressourcen unerlässlich. Derzeit wird eine riesige Menge an Rechenleistung verschwendet, in dem man ungenutzte Prozessoren für mobile Geräte vernachlässigt. Vor allem in bestimmten Zeiträumen, wie in der Nacht, könnten viele Ressourcen ohne Konflikte genutzt werden.

Die allgemeine Idee von MCC ist es, rechenintensive Aufgaben zu entlasten, den Stromverbrauch zu reduzieren und Speicherplatz zu sparen. Dies funktioniert, indem die Daten

in die Cloud, mobile Cloud oder in leistungsstarke IoT-Geräte übertragen werden. Es gibt drei verschiedene Arten von untersuchten MCC Umgebungen.

1. Zunächst lädt ein mobiles Gerät eine Aufgabenanforderung in die Cloud aus und erhält als Antwort eine Lösung.
2. Mehrere mobile Geräte bilden eine so genannte mobile Cloud und verarbeiten Aufgabenanforderungen, die von anderen benötigten mobilen Geräten kommen.
3. Ein ressourcenreiches IoT-Gerät verarbeitet, das sich am Rande des Netzwerks befindet, Aufgabenanforderungen der mobilen Geräte und gibt die Lösung zurück.

Anwendungsszenarien von MCC umfassen mobiles Gesundheitswesen, Mobile Commerce und Mobile Learning. Diese unterschiedlichen Anwendungsszenarien konzentrieren sich auf verschiedene Aspekte des MCC-Paradigmas. In einem Szenario des mobilen Gesundheitswesen spielen die von mobilen Geräten gesammelten Daten eine entscheidende Rolle. Diese Daten können lebensrettende Informationen enthalten, die sofort verarbeitet werden müssen. Im Mobile Commerce hingegen sind die neuen Kommunikationsmuster und Erreichungsmöglichkeiten die wichtigsten Perspektiven.

3.3.4 Ressourcen-Provisionierung

Eine effiziente Ressourcenbereitstellung ist heute ein wichtiger Teil jeder Infrastruktur. Die vorhandenen Ressourcen sollen dabei so effizient wie möglich genutzt und verteilt werden. Effizienz beinhaltet dabei beispielsweise die Optimierung von Kosten, Laufzeit und Energie. Von Ressourcenbereitstellung spricht man, wenn man das Verfahren zur Orchestrierung, Zuweisung und Freigabe sowie Überwachung der verfügbaren Systemressourcen anwendet.

Überwachung Eine Systemüberwachung ist erforderlich, um den Status der laufenden Dienste und der Softwareumgebung im Allgemeinen zu verfolgen. Beispiele sind dabei CPU- und RAM-Nutzung oder Änderungen sowie Fehler in der Topologie. Die gesammelten Daten werden zur weiteren Verarbeitung an einem Ort gespeichert.

Orchestrierung Hierbei werden die Informationen zur Systemüberwachung analysiert und ein Plan zur Bereitstellung erstellt.

Zuweisung und Freigabe Gemäss dem erstellten Plan bei der Orchestrierung werden hier nun die Ressourcen angemessen verteilt.

3.3.5 Orchestrierungs-Tools

Kubernetes beispielsweise ist ein Open-Source-System zur automatisierten Bereitstellung, Skalierung und Verwaltung von kontainerisierten Anwendungen. Neben Kubernetes (k8s)

gibt es neu noch eine leichtere Variante von Kubernetes mit dem Namen *k3s* (5 less than k8s). Laut eigenen Angaben ist die Installation sehr einfach und der benötigte RAM-Speicher halb so gross wie bei Kubernetes selbst. Ausserdem benötigt der Speicherplatz weniger als 40Mb. Weitere bekannte Orchestrierungs-Tools sind Nomad, Rancher, OpenShift, AZK und ContainerShip. Beim LXC Container gibt es daneben noch *libvirt* und *saltstack*.

3.4 IoT Frameworks

Laut dem OpenFog Konsortium [6] bedeutet der Begriff Fog Landscape, dass es sich um eine horizontale, systemnahe Architektur handelt, welche die Rechen-, Speicher-, Netzwerk- und Kontrollfunktionen untereinander verteilt und näher an den Rand der Geräte bringt. Sie sprechen hierbei auch von einem *cloud-to-thing* Kontinuum.

Die Herausforderung besteht dabei in der Erstellung, Ausführung und Unterstützung einer Ausführungsumgebung von IoT Applikationen in der sogenannten *Fog-Landscape*.

Es ergeben sich dabei einige Fragen die nachfolgend aufgelistet werden.

- Was für Mechanismen gibt es, um Ressourcen virtualisiert bereitzustellen?
- Was ist die Vorgehensweise und welche Werkzeuge braucht es, um eine Fog Landscape zu handhaben und Dienste darauf auszuführen?
- Wie können die zu verwendenenden Ressourcen optimal verteilt und Dienste darauf ausgeführt werden?

Es gibt bereits einige gute Ansätze, auf die wir im folgenden etwas näher eingehen. Diese teilweise mit Code bestehenden Ideen und Konzepte sollen uns als Ausgangspunkt dienen, wo möglich auch Verwendung in unserem Projekt finden.

Es soll insbesondere die Frage beantwortet werden, ob damit IoT Dienste in einem Fog-Landscape Umfeld ausgeführt, überwacht und analysiert werden können.

3.4.1 FogFrame

FogFrame [7] ist aus einer Diplomarbeit eines Studenten der TU Wien [8] entstanden. Es beinhaltet eine sehr gut dokumentierte und beschriebene Arbeit und verfügt über alle nötigen Anleitungen und ein auf Github gehostetes Repository [9] mit vollständigem Codeumfang. Diese fällt unter die Apache Lizenz 2.0 [10] und kann nach belieben verwendet und erweitert werden kann.

Kernpunkte

FogFrame ist ein Framework zur Ausführung von IoT Applikationen in einer Fog Landscape. IoT Applikationen können damit platziert (placement), eingesetzt (deployment)

und ausgeführt (execution) werden. Es bietet unter anderem eine Lösung für das Optimierungsproblem von Ressourcenverteilung und Dienstplatzierung (service placement).
Das Ressourcenmodel von FogFrame besteht aus den folgenden Komponenten.

- Fog Cells, welche IoT Geräte (Sensoren, Aktuatoren) kontrollieren und Dienste ausführen.
- Fog Control Nodes kontrollieren Fog Cells und Dienste ausführen.
- Ein Fog Control Node und damit verbundene Fog Cells formen eine Fog Colony, welche zusammen als ein Micro Data Center fungieren.
- Cloud Fog Middleware verwaltet Cloud Ressourcen und unterstützt Fog Colonies.

Das Bild 3.1 illustriert das Ressourcenmodel von FogFrame anschaulich.

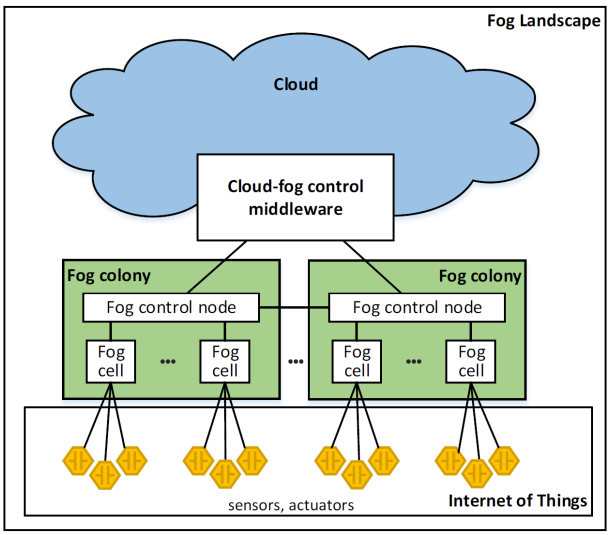


Abbildung 3.1: Ressourcenmodel von FogFrame

Architektur

Die wesentliche Architektur kann am besten durch folgende Grafik 3.2 anschaulich dargestellt werden, welche die Fog Cell Architektur sowie die Fog Control Node Architektur anschaulich darstellt.

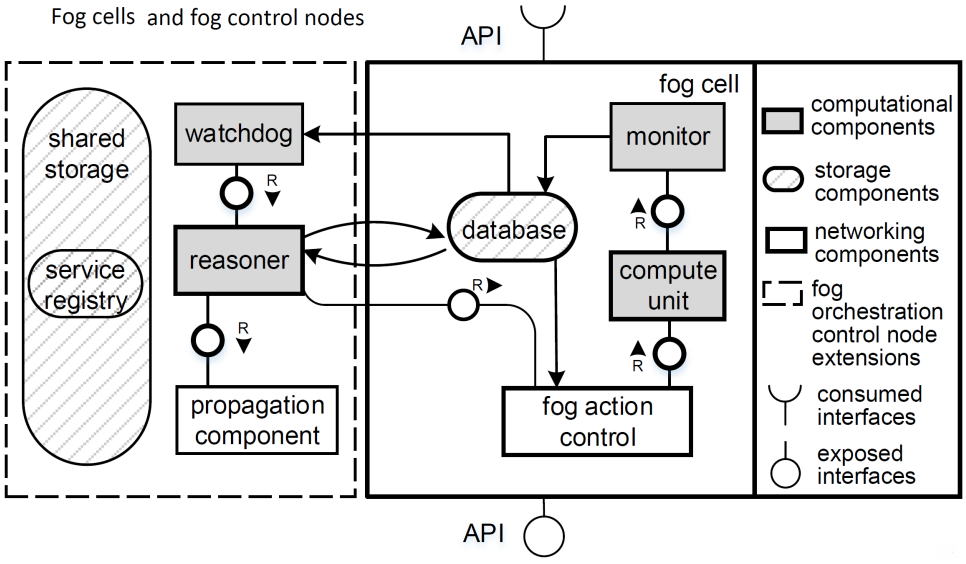


Abbildung 3.2: Fog Cell und Fog Control Node Architektur

Ein weiteres interessantes Merkmal von FogFrame ist das Hintereinanderschalten von mehreren Diensten (Services), wird im nachfolgendem Bild 3.3 illustriert.

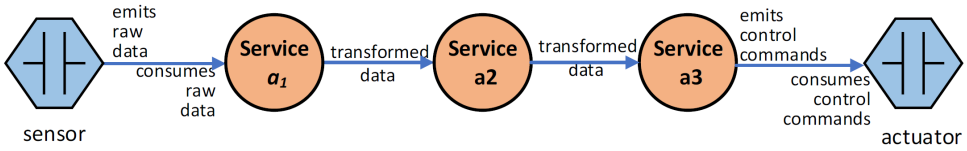


Abbildung 3.3: Applikation Model

Technologien

Raspberry Pi 3 (Hypriot OS) **Docker** **Java**

Apache Maven **OpenStack Cloud** **Redis DB**

Vorteile

+ Die Orchestrierung sowie das Deployment sind mittels Docker möglich.
+ Das Provisioning besteht schon.
+ die Koordinierte Kontrolle über die Fog Landscape wird ermöglicht.
+ Man kann die Ressourcen überwachen und analysieren.
+ Ein Dienstplatzierungsplan (Service placement plan) kann erstellt werden.
+ Applikationen können bereitgestellt und ausgeführt werden.
+ Die Neukonfiguration der Fog Landscape basierend auf Runtime-Events.
+ Es gibt eine ausführliche Dokumentation dazu.
+ Das Framework und Code ist vorhanden.
+ Zudem benötigt man nur eine Apache Lizenz 2.0.

Nachteile

- Diese Architektur wurde nur im Rahmen einer Arbeit entwickelt und wird zurzeit nicht weitergeführt.
- Es ist keine automatische Erkennung von Geräten möglich.
- Es gibt auch keine Fehlertoleranzmechanismen.

3.4.2 FogLAMP

FogLAMP [11] ist eine OpenSource-Plattform für das Internet der Dinge und eine wesentliche Komponente des Fog Computing. Es nutzt eine modulare Microservice-Architektur, die Sensordatenerfassung, -speicherung, -verarbeitung und -weiterleitung an Enterprise-Systeme und Cloud-basierte Dienste umfasst. FogLAMP kann in hochverfügbaren, eigenständigen, unbeaufsichtigten Umgebungen ausgeführt werden, welche eine unzuverlässige Netzwerkverbindung voraussetzen.

Es bietet auch die Möglichkeit, Daten, die von Sensoren kommen, zu puffern und an hochrangige Speichersysteme weiterzuleiten. Es wird davon ausgegangen, dass die darunter liegende Netzwerkschicht nicht immer verbunden oder möglicherweise nicht zuverlässig ist. Daten von Sensoren können einige Tage in FogLAMP gespeichert werden, bevor man sie aus dem FogLAMP-Speicher löscht.

Während dieser Zeit kann es über eine REST-API zur Verwendung durch lokale Analyseanwendungen abgerufen oder auch an interessierte Systeme weitergeleitet werden. Es wurde für den Betrieb in einer Linux-Umgebung entwickelt und verwendet Linux-Dienste.

FogLAMP verfügt über alle nötigen Anleitungen [12] und ein auf Github gepflegtes Repository [11] mit vollständigem Codeumfang, welcher unter die Apache Lizenz 2.0 [10] fällt und nach belieben verwendet und erweitert werden kann.

Kernpunkte

FogLAMP ist eine offene Sensor-to-Cloud-Datenstruktur für das Internet der Dinge (Internet of Things, IoT), die Personen und Systeme mit den Informationen verbindet, die sie für den Geschäftsbetrieb benötigen. Es bietet eine skalierbare, sichere und robuste Infrastruktur für die Erfassung von Daten von Sensoren, die Verarbeitung von Daten am Rand und den Transport von Daten zu anderen Managementsystemen.

FogLAMP kann über unzuverlässige (unreliable) und schnell wechselnden (intermittent) Verbindungen mit geringer Bandbreite arbeiten, welche in IoT-Anwendungen häufig vorkommen. FogLAMP ist eine Sammlung von Microservices, welche im wesentlichen folgende Komponenten umfassen:

- Kerndienste, welche Sicherheit, Überwachung und Speicherung beinhalten
- Datentransformations- und Alarmierungsdienste
- South-Dienste: Sammeln Daten von Sensoren und anderen FogLAMP-Systemen
- North-Dienste: Übertragen Daten an andere Systeme
- Datenverarbeitungsanwendungen am Rand

Dienste können einfach entwickelt und in das FogLAMP-Framework integriert werden (Plugins). In den FogLAMP-Entwicklerhandbüchern wird ausserdem ausführlich beschrieben, wie das möglich ist.

Architektur

Die wesentliche Architektur kann am besten durch folgende Grafik 3.4 anschaulich dargestellt werden.

Plugins sind grün markiert und können geladen, aktiviert und gestartet werden. Mehrere Plugins nebeneinander sind möglich. **Microservices** sind hellblau gekennzeichnet und können ebenfalls nebeneinander im selben Betriebsumfeld existieren. Diese können aber auch in mehreren Umgebungen verteilt werden. **Tasks** sind blau vermerkt, diese unterscheiden sich von Microservices hinsichtlich der Verwendung. Tasks haben in der Regel eine beschränkte Lebensdauer und wenn sie ausgeführt wurden und ihr Ziel erreicht haben, beenden diese sich. Der **FogLAMP Core** ist rosa und beinhalten die gemeinsamen Kernmodule der Plattform.

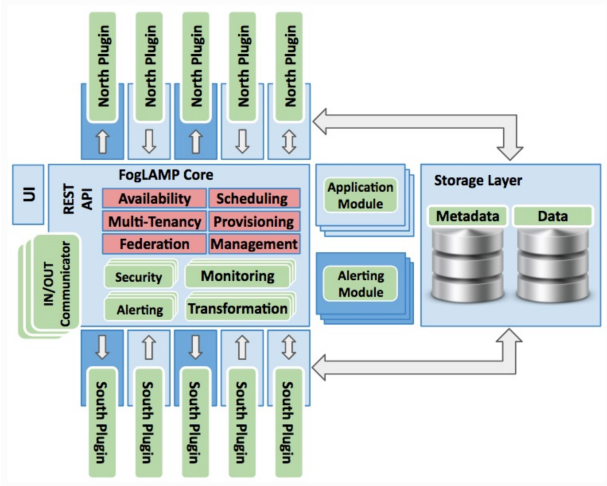


Abbildung 3.4: FogLAMP Architektur

Der Kern besteht aus diesen im nachfolgenden aufgelisteten Aufgaben:

- **Hochverfügbarkeit** von FogLAMP als Clusterplattform.
- **Planung** von Aufgaben auf der Plattform (Scheduling).
- **Zentrale Verwaltung** aller Komponenten (Microservices, Module und Plugins) der FogLAMP-Plattform.
- **Mandantenfähigkeit** für externe Entitäten (Anwendungen, Benutzersitzungen, Geräte) und interne Entitäten (geplante Aufgaben, laufende Dienste).
- Automatische, unbeaufsichtigte, sichere **Bereitstellung** (Provisioning) und **Update** von FogLAMP. Diese können während dem laufenden Betrieb und ohne Dienstunterbruch eingespielt werden. Bei Problemen können diese zurückgesetzt werden. Bei einer bestehenden Cluster-Installation wäre dies ohne Ausfallzeit (Downtime) möglich.
- **Sicherheit** ist ein wichtiger Bestandteil von FogLAMP und es gibt einen zentralen Dienst, welcher eine sichere Kommunikation sowie Authentifizierung ermöglicht.

Technologien

Bei dieser Architektur wurden die folgenden Technologien eingesetzt:

Microservices **REST API** **C/C++**

Java **Python** **SQLite**

Vorteile

+ Es gibt eine ausführliche Dokumentation (https://foglamp.readthedocs.io) dazu.
+ Das Framework und Code (https://github.com/foglamp/FogLAMP) ist bereits vorhanden.
+ Es gibt ein Dashboard (GUI) dazu.
+ Es ist Plugin-basiert.
+ Es handelt sich um eine Microservice-Architektur (MSA).
+ Die Architektur verwendet REST API.
+ Die Architektur verwendet eine Apache License 2.0.

Nachteile

- Es gibtbt keine Orchestrierungs- und Deployment-Logik.

3.4.3 PyOT

PyOT [13] ist eine webbasierte Makroprogrammieroberfläche für das IoT. PyOT verfügt über alle nötigen Anleitungen und ein auf Github gepflegtes Repository [14] mit vollständigem Codeumfang, welche unter die GNU GPL v3.0 [15] fällt und mit gewissen Restriktionen einhergeht. Erwähnenswert hierbei ist, dass neben der Erlaubnis zur Veränderung und kommerziellen Nutzung jedoch zwingend die gleiche Lizenz sowie der Sourcecode offengelegt und Änderungen protokolliert werden.

Nach unserem Verständnis und in diesem Zusammenhang bedeutet Makroprogrammierung soviel wie vorhandene vorgefertigte Funktionen, die verwendet werden können, um

Gruppen von Nodes oder ganzen Netzwerken zu programmieren, Sensoren und Aktuatoren zu kontrollieren. Diese können mittels synchroner oder asynchroner Kommunikation verwendet werden. Anhand von dem beigefügten IPython Interface und der vorhandenen Shell kann dies bewerkstelligt werden.

Kernpunkte

- Die Remote-Verwaltung von IoT-basierten Netzwerken wird ermöglicht.
- CoAP-Ressourcen und 6LoWPAN-Knoten werden als übergeordnete Python-Objekte abstrahiert.
- Eine Web-Oberfläche für die Interaktion mit CoAP-Ressourcen wird angeboten.
- Eine interaktive Makroprogrammierung (mit Hilfe der Shell) wird angeboten.
- Die Ressourcenerkennung und -indizierung basieren auf dem CoRE-Ressourcenverzeichnis.
- Flexible Ereignisbehandlung (automatische Reaktion, wenn IoT-Ressourcen Alarmbenachrichtigungen senden) sind möglich.
- Asynchrone Tasks für die Kommunikation mit IoT-Knoten auf PyoT Workers sind ebenfalls möglich.

Architektur

Die Architektur wird wiederum am besten mit dieser Grafik 3.5 anschaulich illustriert.

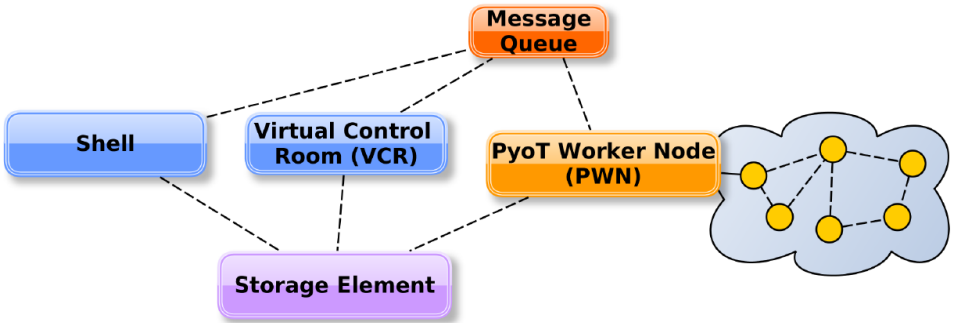


Abbildung 3.5: PyOT Architektur

Der PyOT Worker Node verwendet AMQP als Message-orientierte Middleware (MOM) mit einer Queue sowie CoAP zur Kommunikation mit den IoT Geräten wie nachfolgende Grafik 3.6 zeigt.

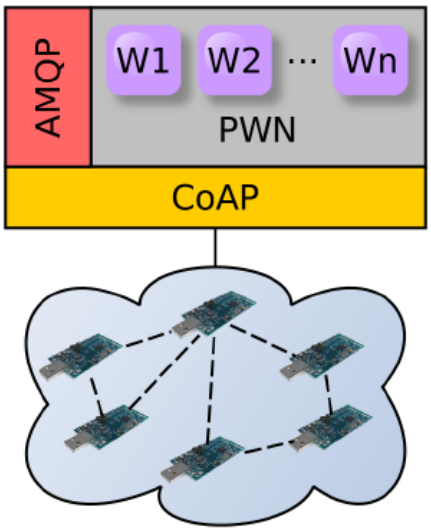


Abbildung 3.6: PyOT Worker Node (PWN)

Technologien

- Web-Based Macroprogramming
- WSN (Wireless Sensor Network)
- CoAP (Constrained Application Protocol)
- 6LoWPAN (IPv6 over Low power Wireless Personal Area Network)
- (I)Python
- Django (ORM - Object-Relational-Mapper)
- RabbitMQ (AMQP - Advanced Message Queuing Protocol)
- mySQL
- Contiki Cooja Simulation

Vorteile

+ Das Framework und Code (https://github.com/tecip-nes/pyot) sind bereits vorhanden.
+ Es gibt eine Simulation dazu.
+ Beinhaltet einen Ready-To-Run PyOT VM Packer.

Nachteile

- Beinhaltet eine Gnu GPLv3 -> Offenlegung, gleiche Lizenz, Statusänderungen.
- Es sind nur eine mangelhafte Dokumentation und Anleitung vorhanden.
- Der letzter Commit ist bereits 4 Jahre her (2015).
- Es beinhaltet einen VM Packer mit Dead-Links.

3.4.4 Foggy

Foggy [16] ist ein Framework für die kontinuierliche Bereitstellung automatisierter IoT-Anwendungen im Fog Computing Umfeld. Offener Quellcode dazu konnte bis anhin jedoch nicht gefunden werden, trotzdem gefällt das Konzept und scheint erwähnenswert.

Kernpunkte

- Version Control Server
- Continuous Integration Tool
- Container Registry
- Node (IoT Geräte mit Rechen- und Speichermöglichkeit [e.g. Gateway, Raspberry Pi])
- Orchestration Server (Ressourcen- und Applikationskatalog)

Architektur

In folgender Grafik 3.7 ist zu sehen, wie diese Komponenten aufgebaut sind und Ihr Zusammenspiel funktioniert.

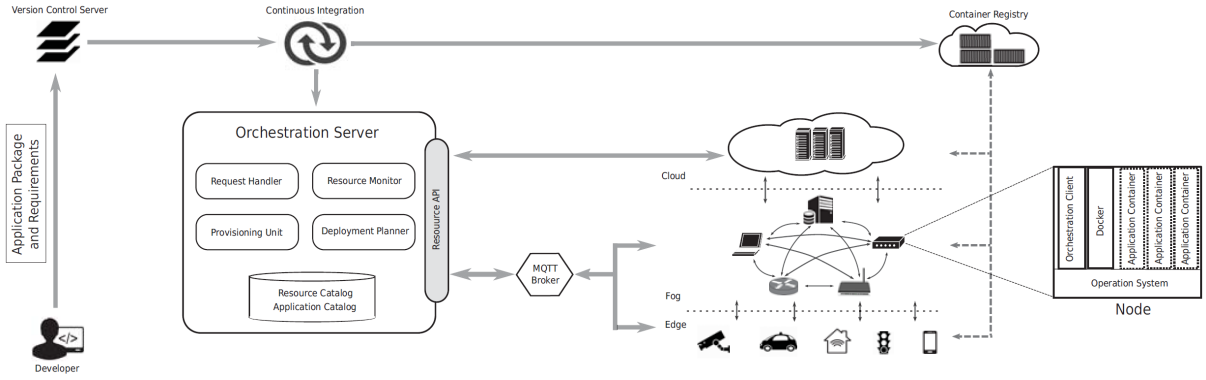


Abbildung 3.7: Foggy Framework Architektur

Technologien

Raspberry Pi 3 (Hypriot OS)

Docker (Registry)

Github

Concourse (CI)

Java

Jersey

REST

cAdvisor

Vorteile

+ Es handelt sich um eine Microservice-Architektur (MSA).
+ Es gibt eine Virtualisierung (Docker).
+ Zudem gibt es ebenfalls ein Orchestrierungs-Konzept.
+ Auch ein Continuous Integration Konzept ist vorhanden.

Nachteile

- Es ist kein Sourcecode vorhanden.

3.4.5 Fog Computing Platform

Fog Computing Platform [17] ist ein verteilte Analyse-Plattform, welche mittels TensorFlow einen Machine Learning respektive Deep Learning Ansatz verfolgt, um nützliche Informationen zu extrahieren. Hierbei steht insbesondere die Verteilung von Ressourcen im Vordergrund. Ausserdem wird Kubernetes als Manamgement-Werkzeug zur Überwachung und Bereitstellung von Docker Containern verwendet.

Kernpunkte

- Virtualisierung
- Zentraler Server
- Monitoring
- Nodes and Edges
- Numerische Berechnung mit Datenflussgraphen

Architektur

Eine vage Darstellung der Architektur respektive der Komponenten kann in nachfolgender Grafik 3.8 betrachtet werden. Dabei kann jedoch noch nicht wirklich von einer fundierten Architektur gesprochen werden, die Konzepte dahinter sind aber vage ersichtlich.

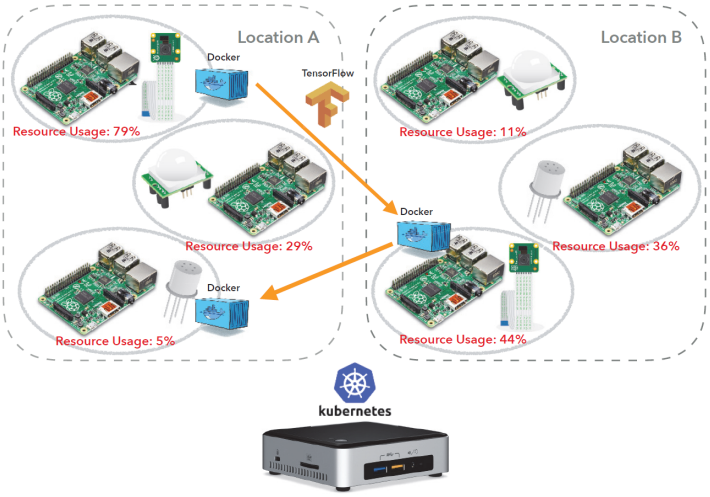


Abbildung 3.8: Fog Computing Platform Architektur

Technologien

Raspberry Pi 3 (Hypriot OS) **Docker**

TensorFlow **Kubernetes**

Vorteile

+ Die Architektur verfügt über einen Machine Learning/Deep Learning Ansatz zur Lastverteilung.

Nachteile

- Es gibt keinen Sourcecode dazu.
- Ist möglicherweise sehr rechenintensiv.

3.4.6 Weitere Frameworks

- Fogernetes https://github.com/ls1intum/fogernetes FogVideo!
- Kubeedge https://github.com/kubeedge/kubeedge GO!

3.4.7 Vergleich

Zusammenfassend noch eine Übersicht der evaluierten Frameworks in Tabelle 3.5.

	FogFrame	**FogLAMP**	**PyOT**	**Foggy**	**FCP**
Erweiterbar	Ja	Ja	Ja	Ja	Ja
Anpassbar	Ja	Ja	Ja	Ja	Ja
Skalierbar	Ja	Ja	Ja	Ja	Ja
Konfigurierbar	Ja	Ja	Ja	Ja	Ja
Koordinierbar	Ja	Ja	Ja	Ja	Ja
Fehlertolerant	Ja	?	Nein	Ja	Ja
Dokumentation	Ja	Ja	(Ja)	(Ja)	(Ja)
Sprache	Java	C/C++/Java/Python	Python	-	-
Lizenz	Apache Lizenz 2.0	Apache Lizenz 2.0	GnuGPLv3	-	-
Orchestrierung	Ja	Ja	Nein	Ja	Nein
CI, Registry usw.	Ja	Nein	Nein	Ja	Nein

Tabelle 3.5: Vergleich Frameworks

3.5 Microservice Frameworks und Komponenten

In diesem Abschnitt werden einige wichtige Microservice Frameworks für Python aufgelistet und beschrieben.

Nach Martin Fowler [18] ist eine Microservice Architektur eine bestimmte Art und Weise, eine Softwareanwendung zu entwerfen, welche einzelne von einander unabhängige Services einsetzt. Diese Services bestehen in einem bestimmten Verhältnis zueinander und haben ihre eigene Verantwortung innerhalb der Softwareanwendung. Diese Services können im Hintergrund beispielsweise mittels RPC, HTTP oder MQTT kommunizieren.

Flask Flask [19] ist ein Web Microservice-Framework für Python, das auf Werkzeuge [20] und Jinja2 [21] basiert. Damit ist es möglich, rasch einzelne unabhängige Web Services zu schreiben. Es ist einfacher zu erlernen und nützlicher, um Services mit einem Routing zu versehen und Logik dazu zu schreiben.

Nameko Nameko [22] ist ein Microservice-Framework für Python, mit dem sich Entwickler auf Anwendungslogik konzentrieren und die Testbarkeit fördern können.

Es ist im Gegensatz zu Flask ein Microservice-Framework und kein Web Microservice-Framework, daher erwähnt Nameko auch explizit, dass sie nur eine limitierte HTTP-Unterstützung anbieten.

Aiohttp Aiohttp [23] [24] ist ein asynchrones HTTP-Client/Server-Framework für *asyncio* und Python und somit ein sehr schnelles und mächtiges Framework zur asynchronen Verarbeitung von Abfragen. Der Fokus liegt hierbei stark auf der Umsetzung von Parallelität und Gleichzeitigkeit im Hintergrund.

3.6 Rules Engines

Im nachfolgenden werden wichtige Rules Engine Umsetzungen aufgelistet und wesentliche Punkte hervorgehoben.

business-rules Venmo Business-Rules [25] ist eine einfach gehaltene Rules Engine mit definierbaren Bedingungen und Aktionen. Bedingungen sind beliebig verschachtelbar und es können eine oder mehrere Aktionen dazu definiert werden (Business-rules existieren bereits für Python).

json-rules-engine Json-Rules-Engine [26] ist eine sehr mächtige und doch leichtgewichtige Rules Engine, bei der Regeln aus simplen JSON-Strukturen bestehen. Dies ermöglicht eine einfache Erstellung der Regeln, welche gut zu lesen sind. Es gibt ausserdem eine volle Unterstützung für *All* und *Any* Boolean Operatoren und es ist möglich diese rekursiv zu verschachteln (Json-rules-engine existiert bereits für JavaScript).

Drools Drools [27] ist ein Business-Rule-Management-System (BRMS), welches für das Implementieren von Expertensystemen verwendet wird. Hierfür verwendet Drools einen regelbasierten Ansatz. Dies bedeutet, dass anhand von vordefinierten Regeln das System selbst auf unterschiedliche Ereignisse reagieren kann.

Das Business-Rule-Management-System verwendet den JSR-94 Standard für den Bau, die Wartung und Durchsetzung von Geschäftsprozessen innerhalb von Organisationen. Dieser Standard definiert eine einfache Programmierschnittstelle um eine Rule-Engine mit einer Java Plattform zu verknüpfen (Drools existiert bereits für Java).

durable-rules Durable-Rules [28] ist eine weitere vielversprechende Rules Engine, welches ein polyglot-Framework zur konsistenten und skalierbaren Koordination von Events in Echtzeit bietet.

Mit durable-rules können Informationen zu Ereignissen verfolgt sowie analysiert werden. Dabei können Daten aus mehreren Quellen kombiniert werden, um komplexes Verhalten zu modellieren.

Eine vollständige Vorwärtsverkettungsimplementierung (wie Rete) wird verwendet, um Fakten und massive Ereignisströme in Echtzeit zu bewerten. Mit einer einfachen und dennoch leistungsstarken, metasprachlichen Abstraktion können einfache und komplexe Regelsätze sowie Steuerungsflussstrukturen (Flussdiagramme, Zustandsdiagramme, verschachtelte Zustandsdiagramme und zeitgesteuerte Flüsse) definiert werden (Durable-rules existiert für Ruby, Python und JavaScript (node.js)).

3.7 Sensoren

Am Anfang von IoT steht immer ein Sensor, denn nur damit können Objekte Zustände erfassen und Aktionen ausführen. Diese beiden Tätigkeiten wie auch die Verbindung zum Netz, machen diese Gegenstände ohne menschliche Hilfe *intelligent*.

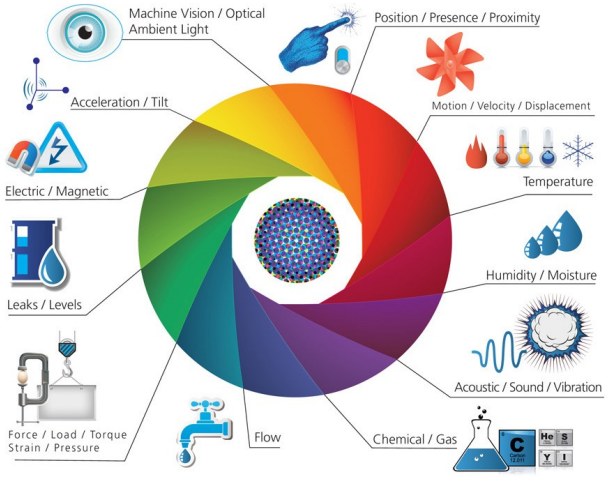


Abbildung 3.9: IoT Sensoren und Aktuaren

3.7.1 Postauto

Während unserer Bachelorarbeit werden wir einen intelligenten Gateway implementieren, der sich in einem Postauto befindet. Das Postauto besteht aus hochmodernen Multisensor-Technologien und beinhaltet unter anderem einen CAN-Bus, an dem ebenfalls diverse Sensoren angeschlossen sind. Die Systemarchitektur des Postautos ist im Internet leider nicht auffindbar, trotzdem erhalten wir einen Einblick von den verwendeten Sensoren.

Abbildung 3.10: Ein autonomer Shuttle Bus als Beispiel

3.7.2 CAN-Bus

Ein CAN-Bus (Controll Area Network) ist ein serielles Bussystem, das den Datenaustausch zwischen Steuergeräten ermöglicht. Moderne Fahrzeuge verfügen über eine Vielzahl an elektronischen Systemen und zum Teil über 50 Steuergeräte, die miteinander kommunizieren müssen. Ein CAN-Bus wird deshalb eingesetzt, da eine solche Verkabelung der einzelnen Steuergeräte kaum noch möglich ist.

Vorteile Durch die einfache Verkabelung der einzelnen Steuergeräte an solche Bussysteme hat man eine deutliche Reduzierung von Kabeln, Steckern und Gewichten. Die Übertragung erfolgt über eine verdrillte Zweidrahtleitung, was preiswert ist. Zudem sind die Sensoren mehrfach nutzbar. Die Sensoren (oder andere Steuergeräte) können ohne grossen Aufwand direkt zum bestehenden CAN-Bus hinzugefügt oder entfernt werden. Bei einem Ausfall hat man keinen Einfluss auf das Bussystem.

Nachteile Die Übertragungsgeschwindigkeit ist begrenzt. Aus diesem Grund kann die Busleistung nicht beliebig lang ausgeführt werden. Bei hohen Übertragungsgeschwindigkeiten kann ein nicht richtig terminiertes Kabelende den gesamten Bus lahmlegen. Deshalb müssen die Enden der Busleitung mit Abschlusswiderständen terminiert werden, um die Signalgüte zu optimieren.

3.7.3 Mögliche Sensoren

Ein autonomes Postauto ist mit vielen Sensoren ausgestattet und wird während Echtzeit überwacht. Die wichtigsten Sensoren, die wir in unserer Arbeit genauer betrachten, haben wir in den nächsten Unterabschnitten beschrieben. Sensoren, die nur einen Status zurückgeben (wie zum Beispiel Tür offen / Tür zu), werden wir hier nicht behandeln.

Trägheitsmessgerät (IMU)

Es gibt verschiedene Typen solcher Trägheitsmessgeräte, die in einem autonomen Fahrzeug vorzufinden sind. Ein solches Gerät kann aus einem Akzelerometer (Beschleunigungssensor) und einem Gyroskop (Kreiselinstrument) bestehen, während ein anderes noch zusätzlich einen Magnetometer verwendet. IMUs werden hauptsächlich für Geschwindigkeitsmessungen, zur Orientierung und für die Gravitationskraft eingesetzt.

LIDAR-Sensoren

LIDAR ist die englische Abkürzung für *light detection and ranging* oder auch *laser detection and ranging*. Mittels dieser Methode kann man eine 3D-Wahrnehmung der Umgebung zur Kartografierung der Standorte erstellen. Zudem gewährleistet diese Methode eine genaue Positionierung und Sicherstellung, damit Hindernisse erkannt werden.

Um jedoch eine hohe Sicherheit gewährleisten zu können, genügt ein einziger LIDAR-Sensor nicht. Beim autonomen Shuttle-Bus (Postauto, Navya) werden beispielsweise direkt mehrere solcher Sensoren verwendet, die alle unterschiedlich ausgerichtet sind. So kann das ganze Umfeld rundherum überprüft werden.

Kamera-Stereovision

Dieser Sensor erkennt Hindernisse und Abschätzungen der Position im Verhältnis zum Fahrzeug. Zudem analysiert er die Verkehrsumgebung (Strassenschilder, Ampeln, etc.) und extrahiert die benötigten Informationen.

Bewegungssensor

Bewegungssensoren sind sehr praktisch und weit verbreitet. Damit lassen sich elektrische Verbraucher (zum Beispiel das Licht) automatisch an- und ausschalten. Das hilft Energie zu sparen und steigert den Wohnkomfort. Häufig werden sie für die Sicherheit eingesetzt, so erkennt man auch, wenn sich jemand in einen überwachten Bereich befindet und geben dann das Schaltsignal an eine Alarmanlage weiter oder schalten beispielsweise einen Strahler an. Mittels Bewegungssensoren lassen sich die Lageänderungen in eine elektrische Grösse umsetzen. Dieses Bauteil ist im einfachsten Fall ein elektrisch-mechanisches Bauteil, wie beispielsweise ein Neigungsschalter. Oft werden Beschleunigungssensoren eingesetzt. Mithilfe dessen werden die Bewegungen über den ermittelten Messwerte als Integration berechnet, da diese Sensoren einfach und günstig sind.

GPS Sensor

Global Positioning System (GPS) Sensoren werden verwendet, um weltweit auf Basis der speziellen GPS Satellitensignale ihre genaue Position zu berechnen. Zudem können die Geschwindigkeit und die Distanz über die Positionsveränderungen des GPS Empfängers berechnet werden. GPS wird in vielen Bereichen der Luftfahrt, des Militärs, der Fahrzeugnavigation oder für die Kartenerstellung, Landvermessung und Wissenschaft verwendet.

Odometer

Ein Odometer ist eine Einrichtung an einem Fahrzeug, der auf der Basis eines Weggebers die zurückgelegte Wegstrecke anzeigt. Typische Synonyme sind Kilometerzähler und Tageskilometerzähler.

Thermometer

Da Temperatursensoren praktisch überall eingesetzt werden, wollten wir mehr darüber erfahren und haben unsere Ergebnisse unten zusammengefasst.

Die Messung der Widerstandsthermometern basiert auf der Widerstandsänderung von Edelmetallen unter Temperatureinfluss. Anhand der Zufuhr eines konstanten Messstroms kann über die Spannungsänderung auf die Widerstandsänderung geschlossen werden, da jedes Metall eine spezifische Widerstandskennlinie besitzt und sich die Spannung sowie der Widerstand proportional verhalten. Um eine Eigenerwärmung des Widerstandsthermometers zu minimieren, wird in der Regel der Messstrom möglichst gering gewählt.

Wenn zwei Drähte aus unterschiedlichen Werkstoffen an einer Seite leitend verbunden werden und dort einer Temperaturdifferenz ausgesetzt sind, liegt an den offenen Enden eine Spannung an. Diese Spannung ist abhängig von der Temperaturdifferenz und den elektrischen Eigenschaften der beiden Werkstoffen. Die Temperatur kann dann über die Spannung gemessen werden, weil sie und deren Kennlinien bekannt sind.

Hygrometer

Die Feuchtigkeit wird anhand der Temperatur, des Drucks, der Masse oder einer mechanischen / elektrischen Änderung einer Substanz berechnet. Die heutigen elektronischen Feuchtigkeitsmesser nutzen entweder die Kondensationstemperatur, die Änderung elektrischer Kapazitäten oder die Widerstände und sind dadurch präziser. Es gibt unterschiedliche Methoden, die Feuchtigkeit zu messen. Sie wird am genauesten mit Hygrometern gemessen.

Luftdruck

Mit einem Luftdruck-Sensor werden die Räder auf Druckverlust überwacht. Der Sensor schlägt sofort Alarm, wenn etwas nicht in Ordnung ist. Die Luftdruck-Kontrolle erfolgt in den meisten Fahrzeugen jedoch über ein Reifendruckkontrollsystem (direkt oder indirekt), da nicht überall zwangsläufig ein Reifendrucksensor eingebaut ist.

Gas

Der Gassensor wandelt die chemische Information in der Umgebungsluft in ein Signal um, welches man weiterverwenden kann. Dadurch, dass dieser Sensor direkt in Kontakt mit der Aussenwelt stehen muss und nicht sonderlich verpackt ist, ist er anfälliger für Alterung, Vergiftung (Umwelteinflüsse, die den Sensor unempfindlich werden lassen) und könnte auch einen falschen Alarm von sich geben.

Die Messung erfolgt über eine Reihe unterschiedlicher Umsetzungen, diversen physikalischen Grössen und Sensorprinzipien. Man muss zuerst wissen, welches Gas man detektieren will und welche Konzentrationen zu erwarte sind.

RFID

Radio-Frequency identifacation (RFID) bezeichnet eine Technologie für Sender-Empfänger-Systeme zum automatischen berührungslosen Identifizieren und Lokalisieren von Objekten sowie Lebewesen mit Radiowellen. Ein solches System besteht aus einem Transponder (Funketikett), der sich am Gegenstand (oder am Lebewesen) befindet. Dieser enthält einen Code zur Kennzeichnung. Diese Transponder können so klein wie ein Reiskorn sein und implantiert werden. Die Kopplung geschieht durch (vom Lesegerät erzeugte) magnetische Wechselfelder in geringer Reichweite oder durch hochfrequente Radiowellen.

Bluetooth Low Energy (BLE)

Bluetooth LE unterstützt die typische Bluetooth-Architektur mit einer Host-CPU für den Upper Layer Stack und einem Radio / Controller für den Lower Layer Stack. Beide sind über das Hosting Controller Interface (HCI) verbunden. Die Radio / Controller Hardware ist in den Varianten *Single-Mode Devices* oder *Dual-Mode Device*s verfügbar. Single-Mode-Hardware wird in Sensoren eingesetzt und unterstützt Bluetooth LE.

Rollen Geräte, die über Bluetooth LE kommunizieren, arbeiten entweder als Master oder als Slave. Diese Funktionen der Rolle bleiben mindestens für die Dauer der Verbindung bestehen.

States Bluetooth LE kann unterschiedliche Zustände (StandBy, Advertiser, Scanner, Initiator, Connection) einnehmen. Die Master-Rolle ist ein Scanner (Scan für Advertiser-Events ohne Verbindungsaufbau) und Initiator (Aufbau einer Verbindung zu einem Advertiser). Die Eigenschaften eines BLE Controllers sind abhängig vom Hersteller und haben eventuell Limitierungen betreffend der Anzahl unterstützter Verbindungen.

Vergleich Classic Bluetooth Classic wird typischerweise zur Übertragung von Streaming Daten verwendet und verhält sich sehr robust bei Störungen durch andere Sender im gleichen Frequenzband. Kopfhörer sind die klassische Anwendung von Classic Bluetooth. Im Gegensatz zu Classic ist LE ideal für Produkte geeignet, die nur eine periodische Datenübertragung und kein kontinuierliches Datenstreaming erfordern. Dadurch eignet sich Bluetooth LE hervorragend für IoT-Anwendungen.

3.8 Simulationen

Simulationen sind eine willkommene Bereicherung für das automatische Testing und der Überprüfung unseres Infrastruktur-Setups. Aus diesem Grund beschreiben und evaluieren wir hier einige Simulationen, welche uns für unser Projekt hilfreich erscheinen könnten.

Die Simulation soll in der Lage sein, verschiedene Sensordaten zu simulieren und diese mittels verschiedenen Schnittstellen wie MQTT [29], CoAP [30] oder REST [31] Daten zu liefern. Weiter sollte es möglich sein, diese virtualisiert laufen zu lassen und sie so an die Testumgebung anzugewöhnen.

3.8.1 Simplesoft

SimpleSoft [32] stellt auf Anfrage eine SimpleIoTSimulator™ Demo zur Verfügung, welches beliebig skalierbar ist und mit verschiedenen Protokollen wie MQTT, CoAP oder Modbus umgehen kann.

Aus unserer Sicht ist diese Simulation für unser Projekt bezüglich Preis, Lizenfragen und Umfang zu gross und durch die Nutzung via GUI nicht praktisch einsetzbar.

3.8.2 IBM Sensor Simulation

Bei IBM Cloud (früher IBM Bluemix) [33] ist es möglich, mittels Erzeugung eines Templates eine IoT Infrastruktur in wenigen Schritten mit einer kleinen Sensor Simulation zu erstellen.

3.8.3 CARLA

CARLA [34] [35] ist ein Open-Source-Simulator für die Erforschung autonomer Fahrzeuge. Sie wurde von Grund auf erstellt, um die Entwicklung, Schulung und Validierung autonomer Fahrsysteme zu unterstützen. Neben dem Open-Source-Code und den Protokollen bietet CARLA offene digitale Objekte (Stadtpläne, Gebäude, Fahrzeugen), die zu diesem Zweck erstellt und frei verwendet werden können. Die Simulationsplattform unterstützt eine Reihe von Sensoren, Umgebungsbedingungen sowie die vollständige Kontrolle aller statischen und dynamischen Akteure, die Erstellung von Karten und vieles mehr.

Einige wesentliche Eigenschaften davon sind folgende:

- Skalierbarkeit über eine Server-Multi-Client-Architektur
- Flexible API
- Autonome Fahrzeugsensor Suite
- Schnelle Simulation für Planung und Kontrolle
- Kartengenerierung
- Simulation von Verkehrsszenarien

CARLA funktioniert nur mit einer externen Grafikkarte zum Rendering und hat dadurch enorme Ressourcenanforderungen, welche man bewältigen müsste.

3.8.4 iFogSim

iFogSim [36] ist eine Simulation der Fog Infrastuktur und kann damit auch Sensoren simulieren. Es existiert ein Github Repository mit Code in Java [37]. Das Projekt scheint jedoch umfangreicher, als benötigt.

3.8.5 SensorTrafficGenerator

SensorTrafficGenerator [38] ist eine einfache in Python geschriebene Applikation, welche randomisierte Daten zur IoT Simulation liefert. Zurzeit werden das Ein- oder Ausschalten von Geräten, Temperatur, Kamerabewegung sowie GPS simuliert.

3.8.6 MQTT Generator

MQTT Generator [39] ist ein einfaches Python-Skript, um Sensordaten aus einer Konfigurationsdatei zu generieren und an einen MQTT-Broker zu senden.

3.9 Geofencing

Der Begriff Geofeancing ist eine Zusammensetzung aus dem Wort geografisch und fence (englisch für Zaun). Mit Geofencing ist eine Art geographischer Zaun gemeint, das meist ein Gebiet mit einem bestimmten Radius ist, in welchem sich ein Objekt befindet. Es gibt verschiedene Anwendungsgebiete für diesen Begriff wie zum Beispiel für Werbung, wenn man an sich in einem bestimmten Umkreis befindet oder als Ortungssystem.

3.9.1 Wie funktioniert Geofencing?

Ein Administrator oder Entwickler muss zunächst in einer GPS- oder RFID-fähigen Software eine virtuelle Grenze um einen bestimmten Ort herum festlegen, um Geofencing nutzen zu können. Diese Grenze kann ein Umkreis von einigen Metern um einen Standort im Google Maps sein, wenn dies bei der Entwicklung einer mobilen APP mit Hilfe von APIs festgelegt wurde. Geofencing eignet sich aber nicht nur für mobile Apps, es wird auch verwendet, um Fahrzeuge in der Logistikbranche oder im Bereich der Landwirtschaft zu orten. Auch Sicherheitsmassnahmen können mit Geofencing vorgenommen werden. Zum Beispiel, wenn sich eine Drohne in einer Flugverbotszone befindet (Flughäfen, Stadien, etc.), kann eine Nachricht an den Benutzer gesendet oder gar die Drohne gesteuert werden.

Automatische Ereignisse bzw. Funktionen sollen beim Ein-oder Austreten aus dieser fest definierten, geografischen Fläche ausgelöst werden. In regelmässigen Abständen sendet das GPS Signal vom beobachteten Objekt seine Position. Befindet sich diese erhaltene Position im Geofence-Raum, so wird eine Aktion ausgelöst.

3.9.2 Proximity Technologien

Es gibt verschiedene Technologien, die ein ortsrelevante Inhalte triggern. Diese Technologien werden wir etwas genauer unter die Lupe nehmen.

iBeacon Ein iBeacon wird über Bluetooth gesendet, das auf einem internetfähigen Endgerät zur Verfügung steht. Diese Technologie wird von iOS und Android unterstützt und für beide Betriebssysteme ist jeweils eine APP notwendig, damit sie funktioniert. Die kleinen Beacon-Geräte werden für eine solche Umsetzung eingesetzt. Der Radius ist zwischen 2 und 100 Metern einstellbar und funktioniert Indoor sowie Outdoor (Tough-Beacons).

Geofence Für Geofence wird ebenfalls ein internet- sowie ein GPS-fähiges Endgerät benötigt und wird von den Betriebssystemen iOS, Android und Windows Mobile unterstützt. Wie beim iBeacon werden auch hier für jedes Betriebssystem eine App sowie Beacons verlangt. Die Radius-Einstellung entspricht genau der, des iBeacon und kann Indoor (schlechtes GPS-Signal) sowie Outdoor eingesetzt werden.

QR-Code Beim QR-Code ist nicht nur ein internetfähiges Endgerät erforderlich, sondern auch ein QR-Code-Scanner / Kamera. Es werden alle Betriebssysteme unterstützt wie beim Geofence, jedoch verlangt keines davon eine App. Um etwas überhaupt scannen zu können, braucht es einen gedruckten QR-Code. Ohne diesen funktioniert es nicht. Im Gegensatz zu den vorherigen beiden Technologien ist hier der Radius etwas schwach und kann nur in einem unmittelbaren Bereich genutzt werden, funktioniert jedoch Indoor und Outdoor (materialabhängig).

NFC Bei der NFC-Technologie wird ein internet- sowie NFC-fähiges Endgerät benötigt und wird ebenfalls von den drei Betriebssystemen (iOS, Android, Windows Mobile) unterstützt - ohne Verwendung einer App. Anders als bei den vorherigen Technologien werden NFC-Tags vorausgesetzt. Zudem ist wie beim QR-Code der Radius nicht sehr gross und funktioniert nur innerhalb einer Distanz von maximal 10 cm. Trotzdem kann NFC Indoor sowie Outdoor (materialabhängig) eingesetzt werden.

Geofencing spielt in der Zukunft eine grosse Rolle und wird vermehrt in verschiedenen Bereichen eingesetzt. In unserem Projekt möchten wir ebenfalls Geofencing nutzen, um den genauen Ort zu bestimmen und kundenspezifische Aktionen auslösen, wenn man beispielsweise kein GPS-Signal mehr hat oder wenn man sich im Ausland befindet.

3.10 Clean Architecture

Wer sich mehr mit der Clean-Architecture Idee beschäftigen möchte, dem seien folgende Links ans Herz gelegt. Die Idee zur Umsetzung in Python ist dem Beispielblog von TheDigitalCatOnline [40] entnommen, in welchem in mehreren Schritten eine hervoragende

Einführung bereitsteht. Ausserdem wird auch auf eine generelle sprachunabhängige Erklärung auf den CleanCoder Blog [41] verwiesen.

Wir beschränken uns hier in nachfolgenden auf eine kurze Erklärung der Architekture mit ihren wesentlichen Merkmalen, um die Vorteile der Clean-Architecture aufzuzeigen.

Grundsätzlich ist die wichtigste Eigenschaft solcher Architekturen die Trennung von Verantwortlichkeiten (separation of concern), dabei wird oft mit verschiedenen Schichten gearbeitet, dies ist ebenso der Fall bei der Clean-Architecure. Die Vorteile liegen dabei klar auf der Hand:

- Unabhängigkeit bez. gewählter Frameworks/Libraries
- Unabhängigkeit bez. gewähltem UI
- Unabhängigkeit bez. gewählter Datenbank
- Vereinfachte Testbarkeit
- Allgemein Teile sind unabhängig untereinander

Im wesentlichen besteht die Clean-Architecture aus den Schichten Entities, Use Cases, Interface Adapters, Frameworks and Drivers, wie aus nachfolgender Grafik 3.11 vom Clean-Coder Blog [41] hervorgeht.

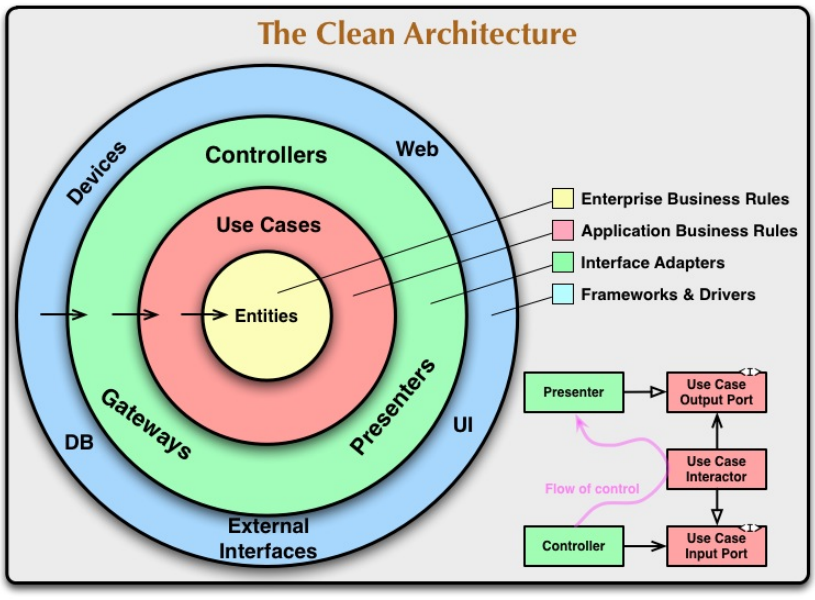


Abbildung 3.11: Clean Architecture Layering

3.10.1 Dependecy Rule

Die Dependency-Regel besagt, das Source-Code-Abhängigkeiten immer nach innen gerichtet sind. Nichts innerhalb eines inneren Kreises soll Kenntnis darüber haben, was in einem äusseren Kreis existiert. Namentlich sind dies beispielsweise Funktionen, Klassen, Variablen und andere Software-Entitäten.

3.10.2 Entities

Entitäten kapseln unternehmensweite Geschäftsregeln, welche in der Regel Objekte mit Methoden oder aber ein Set von Datenstrukturen und Funktionen sein kann. In unserem Fall sind dies DomainModel Objekte von beispielsweise einer Sensor Nachricht, einer Regel oder Representationen von Listen und Aufzählungen.

Es ist sehr unwahrscheinlich, dass diese Objekte sich ändern, wenn sich etwas ausserhalb Befindliches ändert.

3.10.3 Use Cases

Hier befinden sich die applikationsspezifischen Regeln. Hier werden alle Systemweiten Use Cases gekapselt, diese orchestrieren den Datenfluss zu und von den Entities, um das Use Case Ziel zu erreichen.

Wenn sich etwas im UI oder der Datenbank ändert, erwartet man hier keine Änderungen. Sollten sich aber irgendwelche use Case Anforderungen ändern, erwarten wir hier Änderungen.

Diese Schicht ist somit in sich isoliert vom Rest.

3.10.4 Interface Adapters

In dieser Schicht befinden sich die Controller und Views. Da die Modelle nur Daten sind, müssen diese für den zu verwendenden Use Case in ein geeignetes Model umgewandelt werden, dafür wird beispielsweise der JSON Standard mit einem JSON Serializer verwendet. Dies ermöglicht die Daten zwischen der am bequemsten zu speichernden Form für die Persistenz sowie in die am besten geeignete Form für die Bearbeitung von Use Cases umzuwandeln.

3.10.5 Frameworks and Drivers

Die äusserste Schicht beherbergt die externene Schnittstellen und Komponenten wie ein Web Framework oder eine Anbindung an eine Datenbank. Mittels Web Framework ist es beispielsweise möglich von extern via API darauf zuzugreifen. Ausserdem möchte man Daten mittels Dantenbankanbindung von externen Datenbanken abrufen. Meist wird der

Ort für die Datenbankanbindung Repository genannt und der Ort für das Web Framework REST.

4 Anforderungsanalyse und Design

4.1 Funktionale Spezifikationen

4.1.1 Domain Modell

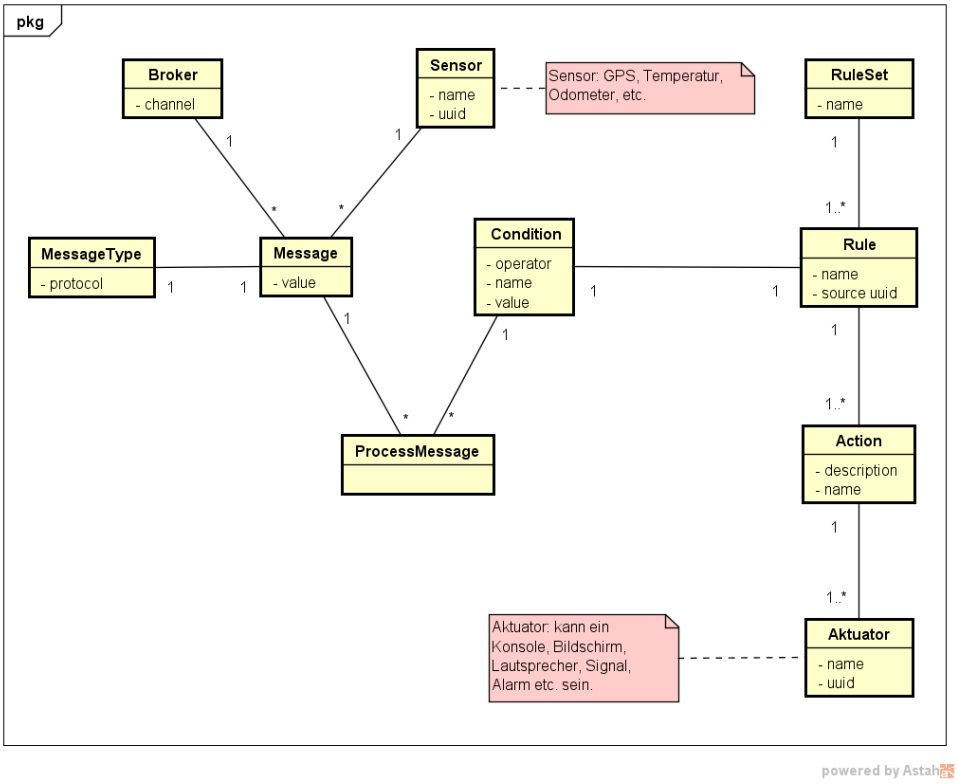


Abbildung 4.1: Rule Engine System

4.1.2 Akteure

Primäre Akteure

Primäre Akteure sind die eigentlichen Benutzer des Systems.

Benutzer Ein Benutzer, der Zugriff zum System hat.

Sekundäre Akteure

Sekundäre Akteure überwachen sowie warten das System und unterstützen den Primär-Akteur bei der Erreichung seines Zieles.

Sensor Alle möglichen Sensoren, die über den Can-BUS [42] oder sonstiges angeschlossen werden.

Broker Beispielsweise ein MQTT-Broker, der die Nachrichten erhält und weitergibt.

Aktuator Ein Aktuator kann ein Lautsprecher, eine Konsole, ein Bildschirm oder ähnliches sein.

Datenbank Die Datenbank (z.B. Redis) erfasst alle wichtigen Daten (temporär).

4.1.3 Use Cases

Anhand des Use Case Diagramms 4.2 ist ersichtlich, wie die Akteure mit dem System interagieren. Es zeigt eine grobe Übersicht der Verwendung des Systems aus Benutzersicht.

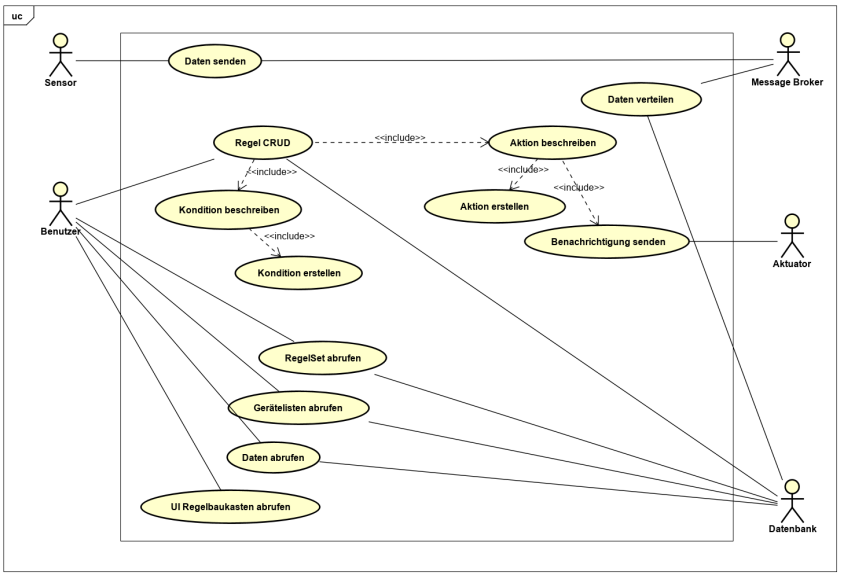


Abbildung 4.2: Use Case Diagram

Use Case 1

Name Daten senden

Aktoren Sensor, Broker

Der Sensor sendet ununterbrochen Daten, im Prototyp werden diese Daten nur simuliert. Diese Daten werden für die Weiterverarbeitung benötigt, nicht gefiltert und genau so gesendet, wie sie vom Sensor einfliessen.

Use Case 2

Name Regel CRUD

Aktoren Benutzer, Aktuator

Der Benutzer soll eine Regel erstellen, lesen, aktualisieren und löschen können. Diese Regeln können ganz unterschiedlich sein und verschiedene Parameter enthalten.

Use Case 3

Name Aktion beschreiben

Aktoren (Benutzer), Aktuator

Dieser Use Case ist Bestandteil von *Regel CRUD* und beschreibt eine Aktion. Eine Aktion hängt vom Aktuator ab und kann zum Beispiel sein: Ton ausgeben. Nicht jeder Aktuator kann jede Aktion ausführen.

Use Case 4

Name Kondition beschreiben

Aktoren (Benutzer), Aktuator

Eine Kondition muss beschrieben werden und ist somit ebenfalls ein Bestandteil vom Use Case *Regel CRUD*. Eine Kondition kann mehrere Konditionen beinhalten und könnte wie folgt beschrieben sein: Maximale Temperatur darf nicht grösser als 22 C° sein.

Use Case 5

Name Benachrichtigung senden

Aktoren (Benutzer), Aktuator

Als Bestandteil des Use Cases *Aktion beschreiben* wird hier eine Benachrichtigung an einen beliebigen Aktuator gesendet. Was eine Aktion sein kann, ist im Use Case weiter oben beschrieben.

Use Case 6

Name RegelSet abrufen

Aktoren (Benutzer), Aktuator

Der Benutzer soll alle möglichen Regeln abrufen können. Eine Regel besteht aus Aktionen und Konditionen, die weiter oben etwas genauer erklärt sind.

Use Case 7

Name Gerätelisten abrufen

Aktoren (Benutzer), Aktuator

Auch eine Geräteliste (Sensoren, Aktuatoren) soll dem Benutzer aufgelistet werden können. Die Sensoren und Aktuatoren werden zu dieser Liste hinzugefügt, sobald eine Verbindung besteht.

Use Case 8

Name Daten abrufen

Aktoren (Benutzer), Aktuator

Die Daten, die in einer Datenbank gespeichert werden, sollen dem Benutzer zur Verfügung stehen. Als Daten gelten einzelne Konfigurationen, Regeln, Konditionen oder sonstige Angaben.

Use Case 9

Name Kondition erstellen

Aktoren (Benutzer), Aktuator

Eine Kondition zu erstellen ist nicht nur ein Bestandteil vom Use Case *Kondition beschreiben*, sondern auch eine Aktion, die von der Rule Engine ausgeführt werden kann. Hier sollen die gewünschten Bedingungen erfasst werden.

Use Case 10

Name Daten verteilen

Aktoren Sensor, Broker

Der Broker verteilt die Daten, die vom Server oder vom Benutzer gesendet werden. Auch hier werden die Daten genau so weitergesendet und nicht vorher gefiltert.

Use Case 11

Name Aktion erstellen

Aktoren (Benutzer), Aktuator

Das Erstellen einer Aktion ist zwar ein Bestandteil vom Use Case *Aktion beschreiben*, aber auch eine Aktion der Rule-Engine. Die Rule Engine kann die gewünschten Events erfassen.

Use Case 12

Name UI Regelbaukasten abrufen

Aktoren Benutzer

Damit ist es als Benutzer möglich im Browser ein grafisches User Interface zur Erstellung von Regeln aufzurufen.

4.1.4 User Storys

User Story 1: Regeln setzen und ausgeben

Als Anwender möchte ich Regeln setzen, um Daten eines Sensors an einen Aktuator, wie beispielsweise die Konsole oder den Monitor, zu senden.

User Story 2: Regeln setzen und abfragen

Als Anwender möchte ich Regeln setzen können, um Daten fortlaufend in einer Datenbank zu speichern und diese abzufragen. Mittels Bedingungen und Handlungen soll es möglich sein komplexe Sachverhalte abzubilden. Beispielsweise möchte ich als Benutzer den Durchschnitt eines oder mehrerer Sensoren akkumulieren und fortlaufend in der Datenbank speichern. Ich möchte diesen Wert später mit einer Abfrage auslesen können.

User Story 3: Werte von Regeln anpassen

Als Anwender möchte ich spezifische Werte einer bestehenden Regel anpassen können, um eine Bedingung oder Aktion anzupassen und zu ändern.

User Story 4: Positionsbestimmung mittels GPS und Odometer

Als Anwender möchte ich eine Regel erstellen, um GPS Daten zu überwachen und beim Fehlen von GPS Daten den ungefähren Standort mittels Odometer zu berechnen, zu speichern und auszugeben.

User Story 5: Einsatz Geofencing

Als Anwender möchte ich mittels Geofencing Regeln für bestimmte Bereiche definieren können, um dort gewisse Aktionen auszulösen. Beispielsweise weiss man, dass in bestimmten Bereich, welcher mittels Geofencing bestimmt wird, kein GPS-Signal empfangen werden kann. An diesen Hotspots muss zwingend der Odometer zum Einsatz kommen, um den Standort trotzdem genau bestimmen zu können, um beispielsweise eine Haltestelle anzusagen.

User Story 6: Temperaturschwellwert und Alarm

Als Anwender möchte ich einen Schwellwert für den Temperatursensor setzen, um bei Überschreiten dieses Wertes ein Ereignis auszulösen Dies kann eine Nachricht oder eine Aktion sein. Beispielsweise kann eine Konsolenausgabe oder auch ein Befehl an ein anderes angeschlossenes Gerät versendet werden.

User Story 7: Sensordaten filtern

Als Anwender möchte ich Daten von einem bestimmten Sensor (zum Beispiel Temperatur) mittels Filter sammeln, um spätere Auswertungen durchzuführen.

User Story 8: Länderwechsel erkennen

Als Leitstellenverantwortlicher möchte ich eine Alarmierung, wenn das Fahrzeug das Land verlässt.

User Story 9: Routenabweichung erkennen

Als Leitstellenverantwortlicher möchte ich alle Abweichungen der Route von Fahrzeugen erhalten, um über die Art und den Grund der Abweichung mittels Rücksprache an den Fahrer informiert zu werden.

User Story 10: Umleitungen anzeigen

Als Fahrer möchte ich über Umleitungen rechtzeitig informiert werden sowie Vorschläge zu Alternativrouten erhalten, um frühzeitig darauf reagieren zu können.

User Story 11: Haltestellen ansagen

Als Fahrer möchte ich, dass die Haltestellen einer Strecke über den Lautsprecher angesagt werden. Diese Ansagen müssen vor Erreichung der Haltestelle in einer gewissen Distanz ausgeführt werden.

User Story 12: Echtzeitdaten mittels App abfragen

Als Fahrgast möchte ich mittels einer App Echtzeitdaten des Fahrzeugs erhalten, um auf dem Laufenden zu bleiben. Dies können Position, Geschwindigkeit, nächste Haltestelle oder beispielsweise Innen- und Aussentemperatur sein.

User Story 13: Echtzeitdaten Position für rechnergesteuertes Betriebsleitsystem

Als rechnergesteuertes Betriebsleitsystem möchte ich zur Kontrolle und Koordination der verschiedenen Fahrzeuge die exakte Position des Fahrzeugs in Echtzeit erhalten, beispielsweise, um bei einem Notfall Blaulichtorganisationen dorthin zu leiten.

User Story 14: Anbieten von geobasierten Werbeinhalten

Als Werbeanbieter möchte ich anhand der Position standortbezogene Werbung anbieten, welche darauf abzielt, den Fahrgästen in der Nähe befindliche Geschäfte, Firmen und Freizeitmöglichkeiten anzubieten.

User Story 15: Eco-Drive Daten

Als Fahrer möchte ich Daten zum Treibstoffverbrauch, dem Reifendruck, den Aussentemperaturen und Fahrstil erhalten, um meinen ökologischen Fahrstil zu ermitteln und gegebenenfalls mittels Verbesserungsvorschlägen anzupassen.

User Story 16: Motorüberwachung

Als Fahrer möchte ich eine Überwachung sämtlicher, kritischer Werte (wie Schwellenwerte von Temperaturen des Motors, Getriebes oder der Abgase) erhalten, um diese bei Problemen anzuzeigen und dementsprechend den Garagenbetrieb zu benachrichtigen.

User Story 17: Füllstandsanzeige

Als Fahrer möchte ich über alle Füllstände wie Motoren-, Getriebeöl sowie Kühlwasser auf dem laufenden bleiben, indem diese auf der persönlichen Anzeige dargestellt werden. Dafür ist schliesslich keine manuelle Kontrolle mehr nötig - was viel Zeit spart.

User Story 18: Kontrolle Beschleunigungssensor

Als Leitstellenverantwortlicher möchte ich darüber informiert werden, wenn ein Beschleunigungssensor eines Fahrzeugs dermassen enorme Wertunterschiede liefert, das ein möglicher Unfall nicht ausgeschlossen werden kann, um schneller einen Notruf abzusetzen.

User Story 19: Pünktlichkeitsmessung

Als Disponent möchte ich die Soll- und Istzeiten der An- und Abfahrszeitpunkte an Haltestellen erhalten, um Probleme zu erkennen und Verbesserungsvorschläge anzubringen.

User Story 20: Verbrauchsgerechte Fahrtenaufteilung

Als Buchhalter möchte ich für jede getätigte Fahrt genaue Angaben dazu erhalten, für welchen Auftrag die gefahrenen Distanzen verrechnet werden müssen. Damit soll die Verteilung der Kosten auf die Auftragsarten (wie den regionalen öffentlichen Verkehr (RöV), touristischer Fahrten, Schülerverkehr oder Auftragsverkehr) vereinfacht werden.

User Story 21: Fahrgastzählungen

Als Disponent möchte ich Angaben zur automatischen Fahrgastzählung pro Fahrzeug und Haltestelle zum Ein- und Ausstieg erhalten, um eine bessere Planung des Angebots mittels Anpassung der Fahrzeuggrösse, Fahr- oder Haltezeiten zur Verfügung zu stellen.

User Story 22: Datenkorrelation

Als Anwender soll es möglich sein, mehrere Sensoren miteinander zu verknüpfen und zu vergleichen, um Regeln zur Korrelation zu erstellen. Als Beispiel soll der Temperaturwert des Motors mit der Geschwindigkeit korreliert werden, um bei Unstimmigkeit einen Event, wie einen Alarm, zu triggern. Wenn die Geschwindigkeit beispielsweise weniger als 30 km/h beträgt und der Motor über einen gewissen Zeitraum noch immer eine Temperatur über 350° Grad Celsius misst, könnte etwas nicht mehr in Ordnung sein und der Fahrer soll benachrichtig werden.

4.1.5 Nicht funktionale Anforderungen

Nicht funktionale Anforderungen beschreiben, *wie* das System spezifische Funktionen ausführt und *wie* es darauf reagieren soll. Diese nicht funktionalen Anforderungen können beispielsweise mittels S.M.A.R.T Kriterien [43] zu ihrer Zielerreichung überprüft werden. S.M.A.R.T steht hierbei für *spezifisch*, *messbar*, *erreichbar*, *relevant* sowie *zeitgebunden*. Im nachfolgenden werden die, für unser Projekt wichtigsten nicht funktionalen Anforderungen, beschrieben.

Rahmenbedingungen (Environment Constrains)

Eine wesentliche Bedingung, welche vorgegeben wurde, ist die lauffähigkeit auf einem ARM Prozessor System mit mässiger Leistung (1.2Ghz ARM DualCore, 1GB RAM, 4GB Flash Speicher). Diese Werte sind vergleichbar mit einem Raspberry Pi 3 Model B.

Effizienz (Efficiency)

Verbrauchsverhalten (Resource Behavior)
Da unser System nicht als alleiniger Verbraucher auf einem System laufen wird, sollte nach unseren Schätzungen der Verbrauch der CPU nicht mehr als 1/4 des Gesamtsystems ausmachen. Gleiches gilt auch für den RAM Verbrauch, welcher ebenfalls nicht mehr als 1/4 in Anspruch nehmen sollte. In Zahlen ausgedrückt sollten wir einen maximalen Ressourcenverbrauch von 300Mhz, sowie 250Mb RAM anstreben.

Skalierbarkeit (Scalability)
Die maximale Anzahl an speicherbarer Daten ist an verschiedene Bedingungen geknüpft. Relevant sind hier die effektive Datengrösse einer Nachricht eines Sensors, die Anzahl angeschlossener Sensoren selbst und der gewünschte Zeitraum der Speicherung der Nachrichten. Rechnet man beispielsweise mit 10 Sensoren, welche pro Sekunde eine Nachricht in der Grösse von 100 Bytes absetzen und wir diese für 24h speichern möchten, ergibt dies einen ungefähren Speicherbedarf von umgerechnet: 10 x 100 x 3600 x 24 = 86’400’000 Bytes. Dies entspricht aufgerundet knapp 100Mb.

Es ist garantiert das ein User synchron auf die Daten zugreifen kann. Dies könnte zwar durch einen Load-Balancer und beispielsweise einem Unicorn-Worker-Prozess - welcher den Gateway steuert - erhöht werden. Da aber auf Redis die Daten nur synchron abgefragt werden können, spielt dies keine Rolle und ist für die geringe Anzahl an Abfragen auch nicht von Bedeutung.

Zuverlässigkeit (Reliability)

Wiederherstellbarkeit (Recoverability)
Regeln werden persistiert und bei einem Neustart des Systems neu eingespielt. So wird garantiert, dass keine Regeln verloren gehen.

	Fehlertoleranz (Fault Toleranz) Für die API Abfragen gibt es eine Fehlerhandhabung mit verschiedenen Fehlerklassen wie System-, Parameter- oder Ressourcenfehlern.
Änderbarkeit (Maintainability)	*Überprüfbarkeit (Auditability)* Mittels Logging werden API Aufrufe, relevante Informationen, Werte von Funktionsaufrufen protokolliert und in Logfiles aufgezeichnet.
Übertragbarkeit (Portability)	*Erweiterbarkeit (Extendability)* Logik zur Erweiterung von Regeln ist in einer Datei festgehalten und kann um neue Funktionalitäten ergänzt werden.
Benutzbarkeit (Usability)	*Verständlichkeit (Understandability)* Es wurde Wert darauf gelegt, dass Regeln einem einfachen Muster folgen, um diese unkompliziert zu erstellen, einfach zu erlernen und zu halten. Eine Regel soll nur die zur Erfüllung notwendigen Parameter und Eigenschaften besitzen. Namentlich sind dies Bedingungen (Conditions) sowie Aktionen oder Handlungen (Actions). Ausserdem sollen sie in einem einfachen und verständlichen Format definiert werden können.

Die Zielerreichung der nicht funktionalen Anforderungen kann im Kapitel 6.1 nachgeschlagen werden.

4.1.6 Sequenzdiagramme

Im folgenden Abschnitt werden die wichtigsten Sequenzdiagramme auf der Basis von UML (Unified Modelling Language) aufgezeigt und deren Abläufe beschrieben. Ziel ist es, die verschiedenen Aktionen zwischen den einzelnen mit sich kommunizierenden Komponenten aufzeigen.

Rule erstellen

Der Benutzer schickt mittels Browser oder CLI eine Regel im JSON Format an den Gateway API Endpunkt. Da der Gateway REST-konform umgesetzt wurde, muss somit zur Erstellung einer Regel mittels POST-Request Daten im JSON Format an den dafür definierten Endpunkt geschickt werden (siehe dazu auch Abschnitt 4.2.4). Der Gateway leitet diese Anfrage anschliessend an den Rule-Manager weiter, welcher die Regel validiert und wenn für gültig befunden, anschliessend als Regel in REDIS speichert. Dadurch wird eine Rule-ID (beispielsweise eine eindeutige UUID) erstellt und an den Rule-Manager zurückgegeben, welcher diesen wiederum dem Gateway weiterleitet und dieser schliesslich dem Benutzer mitteilt.

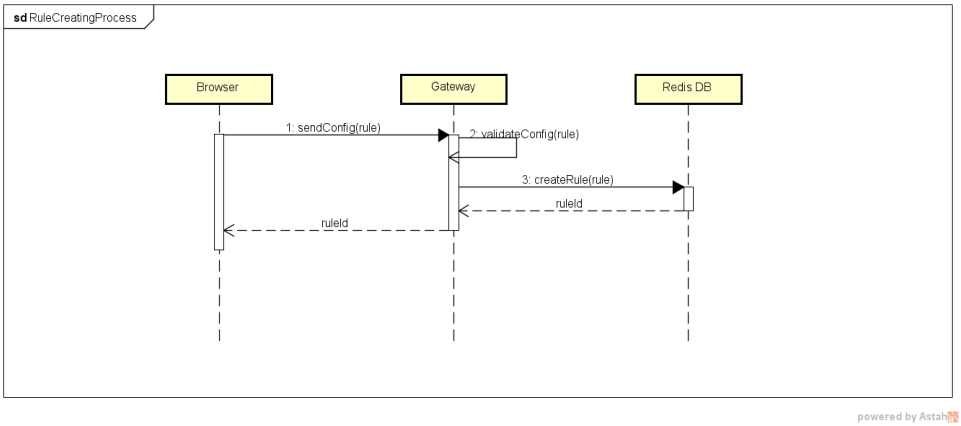


Abbildung 4.3: Rule erstellen

Rule updaten

Der Benutzer schickt mittels Browser oder CLI eine Regel im JSON Format an den Gateway API Endpunkt. In der Regel müssen zwingend eine gültige Rule-ID sowie die zu updatenden Daten enthalten sein. Da auch hier der Gateway REST-konform umgesetzt wurde, muss somit zum Updaten einer Regel ein PUT-Request mit Daten im JSON-Format an den dafür definierten Endpunkt geschickt werden. Der Gateway leitet diese Anfrage anschliessend an den Rule-Manager weiter, welcher die Regel validiert. Weiter holt sich der Rule-Manager die bestehenden Rules aus der Redis Datenbank, um damit zu prüfen, ob diese Regel schon existiert oder nicht. Wenn eine Regel existiert, muss anschliessend überprüft werden, ob die mitgeschickten Parameter beziehungsweise Werte vorhanden sind. Wenn ja, sollen diese angepasst und neu in Redis gespeichert werden. Sind Parameter mitgeschickt worden, welche es noch nicht gibt, müsste in einem weiteren Schritt geprüft werden, ob diese erlaubt sind. Wenn ja, können sie zur bestehenden Regel hinzugefügt werden. Als mögliche Antworten können „Regel existiert nicht“, „Regel ID“oder „Parameter nicht erlaubt“zurückgegeben werden.

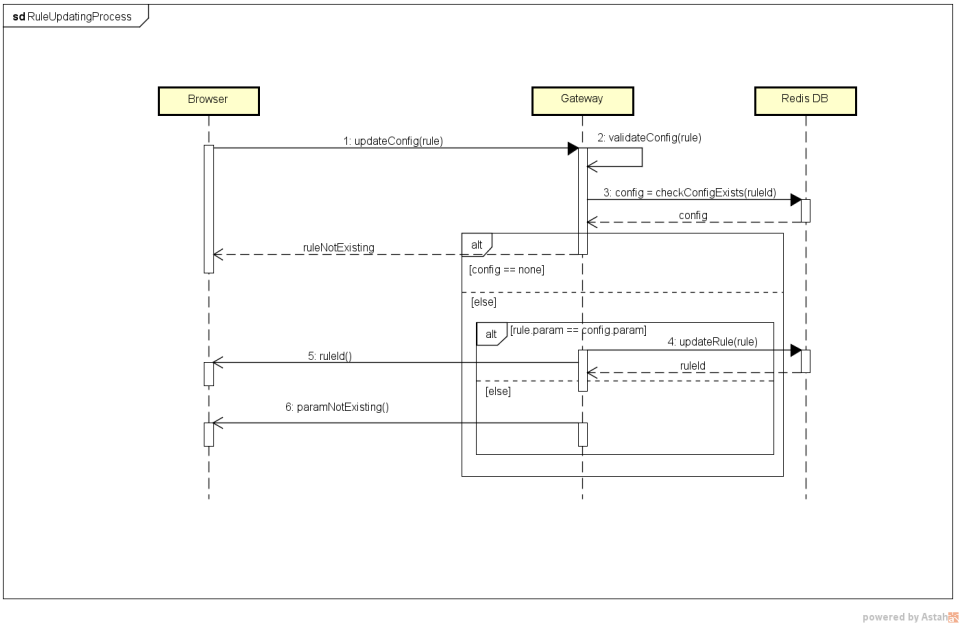


Abbildung 4.4: Rule updaten

Message bearbeiten

Sensoren senden (mittels publish) Daten an den MQTT Broker mittels vordefiniertem Channel. In userem Fall lautet dieser Channel *sensors*. Der Connection Manager registiert sich (mittles subscribe) auf diesen Channel und sammelt die Daten, welche er an Redis zur Speicherung weiterleitet. Diese ermöglicht eine Sammlung von Daten über einen gewissen Zeitraum und fungiert als Caching.

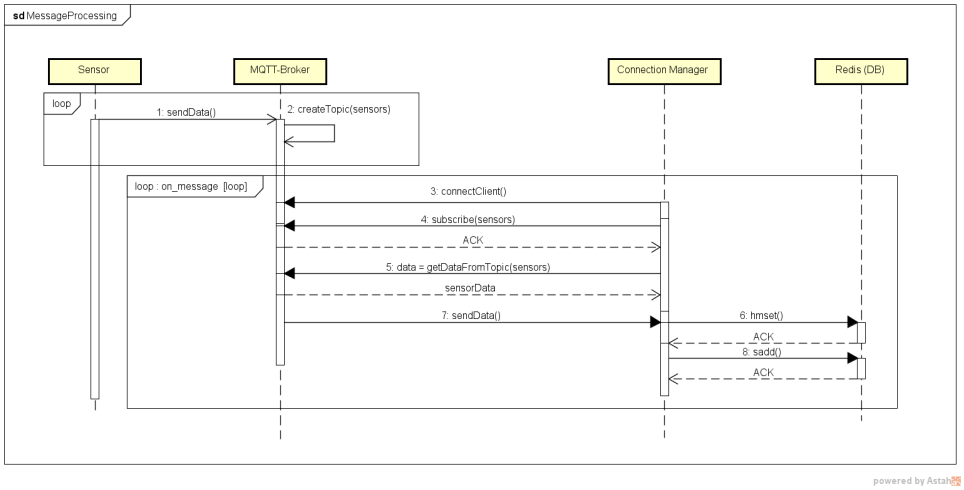


Abbildung 4.5: Message bearbeiten

Echtzeitanalyse

Die Daten sollen in Echtzeit analysiert werden können. Dabei greift der Echtzeit-Service auf die Regelsätze in Redis zu. Er holt sich die Sensor-Daten des Brokers und verarbeitet für alle eingegangenen Daten, ob eine der gesetzten Regeln zutrifft und eine Aktion ausgelöst werden muss.

Optional: Es soll möglich sein, nur noch Regeln bei Änderungen neu zu prüfen. Dafür müsste der Dienst so umgeschrieben werden, dass er via Redis auf bestimmte Daten zugreift. Beispielsweise könnte dies in Redis mittels dem Publish/Subscribe Pattern, welcher Redis unterstützt, umgesetzt werden. Diese Implementation ist noch offen.

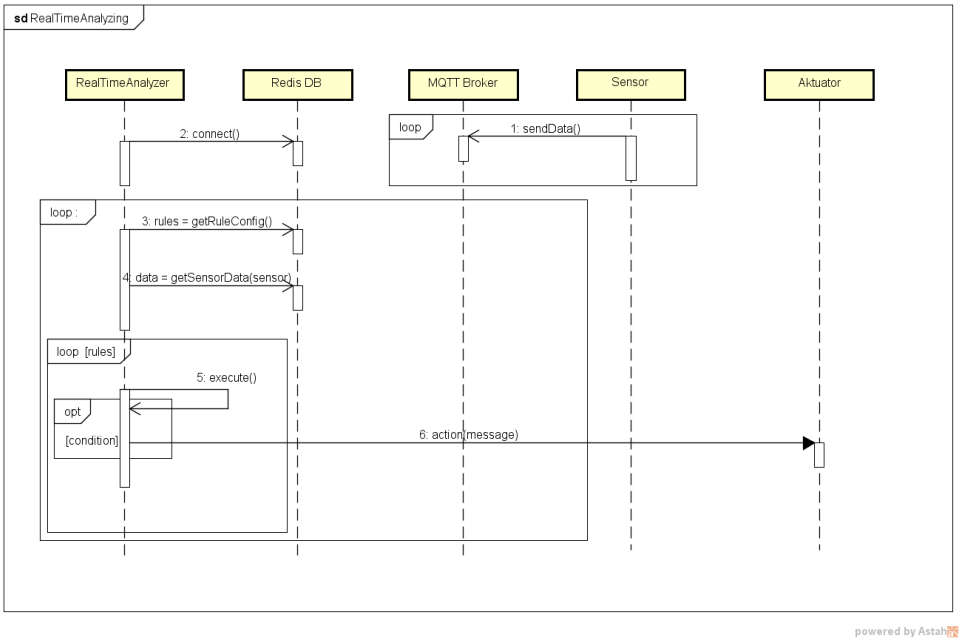


Abbildung 4.6: Echtzeitanalyse

Aktuatoren-Behandlung

Die Kommunikation mit den Aktuatoren, ebenso wie die mit den Sensoren, findet via MQTT Broker statt. Aktuatoren verwenden in unserem Aufbau zwei verschiedene Channels. Der allgemeine Channel *actuators* ist dazu da, dass Aktoren ihre Aktuator-ID versenden und diese der Connection-Manager erfährt. Der zweite Channel ist der aktorspezifische individuelle Channel. Dieser Channel besteht aus dem allgemeinen Channel Teil *actuators* sowie der Aktuator-ID. Ein Beispiel für den fiktiven Aktuator 1 mit aktuator-id könnte folgendermassen lauten: *actuators/actuator1*. Dadurch wird eine Differenzierung der einzelnen Aktuatoren möglich.

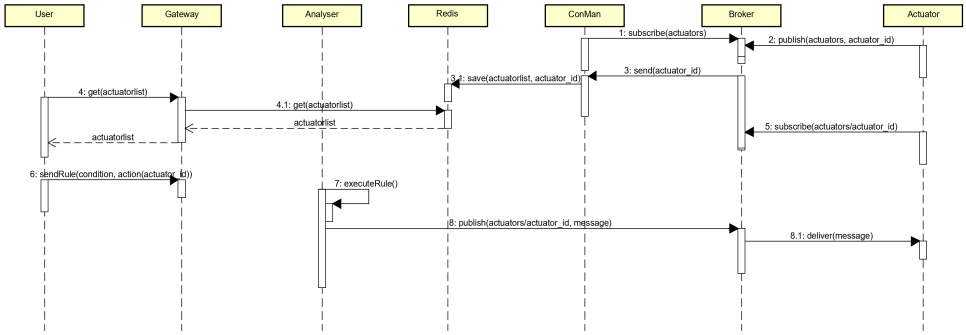


Abbildung 4.7: Aktuatorbehandlung

4.2 Technische Spezifikationen

Die technischen Spezifikationen definieren, wie die funktionellen und nicht-funktionellen Anforderungen des Systems umgesetzt werden können, um die Kundenanforderungen zu erfüllen. Konkrete Design- und Technologieentscheidungen werden getroffen.

4.2.1 Architektur

Auf der Grafik 4.8 wird eine Gesamtübersicht einer möglichen Architektur gezeigt. Nach Absprache des Kunden ist diese jedoch nicht Teil der Anforderungen und wird daher nur als Teil der möglichen Aspekte einer Fog Computing Engine, respektive Frameworks, hier aufgezeigt.

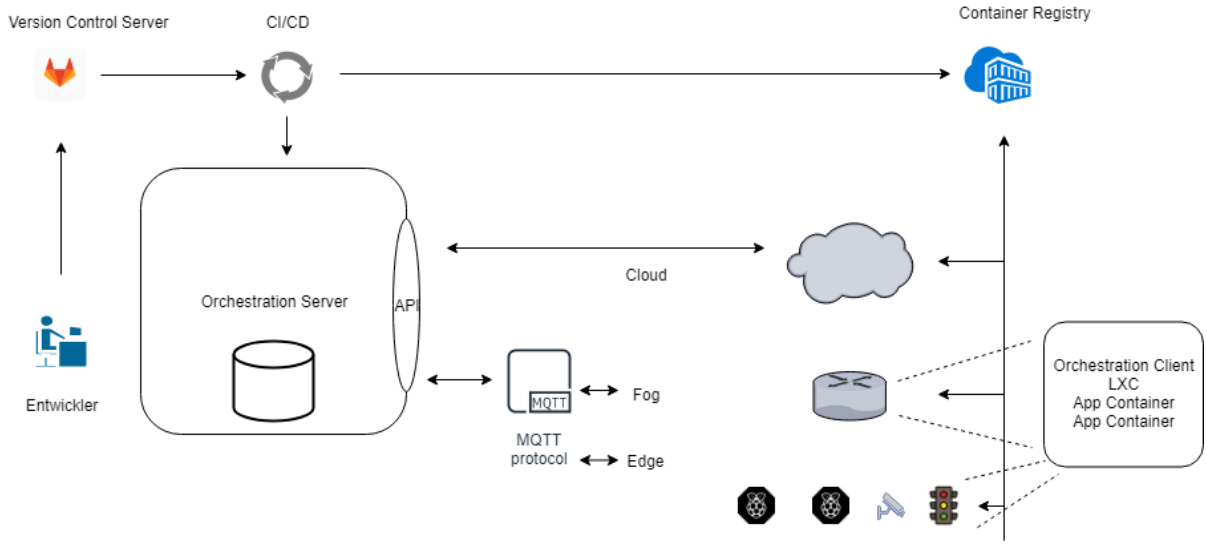


Abbildung 4.8: Architektur Deployment

Nachfolgendes Deployment-Diagram 4.9 zeigt alle wesentlichen Komponenten unserer Rule-Engine. Dies ist der Schwerpunkt und der Fokus unserer Arbeit.

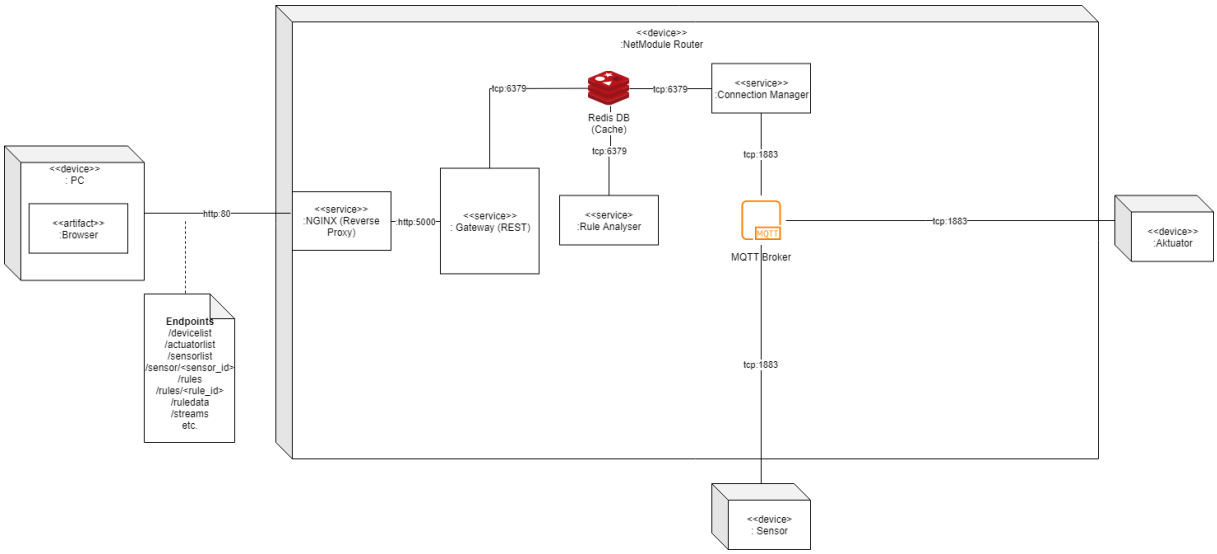


Abbildung 4.9: Komponenten der Rule-Engine

Eine mögliche Regel im JSON Format könnte den folgenden Inhalt wie in 4.1 aufweisen.

```
[{
  "conditions": {
    "all": [{
      "name": "sensor_id",
      "operator": "equal_to",
      "value": "d6d65a2a-6339-11e9-aa33-dbb318e8bbf7"
    }, {
      "name": "current_value",
      "operator": "greater_than",
      "value": 20
    }]
  },
  "actions": [{
    "name": "send_to_actuator",
    "params": {
      "actuator_id": "b2892a16-633a-11e9-b685-37d81680a95d"
    }
  }]
}]
```

Listing 4.1: Beispiel einer Regel als JSON

4.2.2 Design- und Technologieentscheide

Datenbank

Als Datenbank haben wir Redis und InfluxDB in Betracht gezogen. Um abzuwiegen, welche Datenbank für uns die passendere ist, haben wir eine Nutzwertanalyse, wie im Bild 4.10 zu sehen ist, erstellt.

Benefit-Analysis		Redis		InfluxDB	
Beschreibung	Gewicht. [%]	Wert (3)	Total	Wert (3)	Total
Ressourcenverbrauch (1)	30	3	90	2	60
Performance	20	2	40	5	100
Benutzbarkeit (2)	10	4	40	1	10
Installierbarkeit	10	2	20	4	40
Speichermöglichkeit (4)	10	5	50	3	30
Umfang	10	4	40	3	30
Zugriffsmöglichkeit	10	3	30	2	20
			310		**290**

Abbildung 4.10: Nutzwertanalyse: Datenbanken

	Redis	**InfluxDB**
Ressourcen	Es sind keine Messdaten vorhanden, daher nehmen wir an, dass nicht auf dieses Kriterium geschaut wird (Python-like Verbrauch).	Man setzt folgende Hardware-Spezifikation voraus: mind. 4 Cores, 4 - 8 GB RAM, SSD (500); das ist für unseren Gebrauch ungünstig, da die zur Verfügung gestellte Hardware nur 2 Cores hat.
Performance	Die Performance ist abhängig von der eigens geschriebenen Zusatzlogik sowie der Verarbeitung von Messages (zur Zeit sychron).	Redis ist sehr performant und bis zu 20 Mal schneller als ein SQL Server.

	Redis	**InfluxDB**
Benutzbarkeit	Die Benutzbarkeit ist abhängig von der eigens geschriebenen Zusatzlogik und beinhaltet zur Zeit die sychrone Verarbeitung von Messages.	Dank *Chronograf* hat man eine einfache Benutzeroberfläche, mit der man diverse Alarme und Konditionen setzen kann. Man hat stets den Überblick über die gefilterten Daten. Man kann die API benutzen, um mit Alarmen oder allgemein Daten zu arbeiten. Doch kann man mittels API keine neuen Alarme erstellen oder sie bearbeiten. Dafür muss man das Dashboard anpassen.
Installierbarkeit	Dank Python sind die Business-Rules sowie Redis sehr einfach als Modul einzubetten und zu verwenden.	Man kann Influx ganz einfach auf Ubuntu installieren. Jedoch muss man diverse Schritte durchführen, um eine Datenbank zu erstellen, die Konfigurationsdatei anzupassen und so weiter. Man muss alles installieren (TICKstack) - kann aber von den verschiedenen Möglichkeiten profitieren.
Speicher	Redis bietet die Möglichkeit Hashes, Listen, Key/Values sowie Streams zu speichern	Da beim Speichervorgang jegliche Schreibweisen erlaubt sind (zum Beispiel boolean: t, T, true, True, TRUE), ist es für den Benutzer etwas einfacher, da sich in solchen Fällen keine Fehler einschleichen. Für die Speicherung verwendet man die SQL-Schreibweise. Bei einem SELECT-Befehl erhält man eine Tabelle-Form oder kann mittels GET-Request auch ein JSON zurückerhalten. Daten werden immer mit einem Timestamp gespeichert. Jedes Paar ist ein field set (field key = minors, key value = adults) Man kann via API die Regeln nicht anpassen!

	Redis	**InfluxDB**
Umfang	Der Umfang der Business-Rules muss selbst definiert werden, kann jedoch mit etwas Aufwand selbständig erweitert werden. Die Art der Speicherung muss in Redis selbst gewählt werden.	TICKstack beinhaltet viele bereits erwähnte "Komponente", die nach der Installation sofort einsatzbereit sind. Leider kann man jedoch allgemein im Google vielleicht nicht viel zu Influx finden, weil es ein noch relativ neues Produkt ist.
Zugriff	Für die Business Rules können mittels JSON Variablen und Aktionen definiert und so via selbst erstelltem REST Wrapper in die Logik eingebaut werden. Redis bieten den Zugriff nur mittels Keys und ist im Grunde auch nur ein Key/Value Store bei dem Daten jedoch als Hash, Listen sowie Streams abgelegt werden können. Es wird primär zum Caching verwendet.	Die InfluxDB erlaubt keine JOINS. Die Daten können nur bis zu einer Woche rückverfolgt werden - spätere Daten sind dann nicht mehr erhältlich. CRUD ist nicht möglich! Wenn man also viele Änderungen vollziehen möchte, ist InfluxDB nicht die richtige Wahl. InfluxDB ist nicht RESTful! InfluxDB retourniert JSON und die Daten erscheinen im „results"array. Keine Sets, Lists oder ähnliches werden unterstützt. Man kann mit Python programmieren.

Programmiersprache

Als Programmiersprache haben wir C, Python, Micro-Python und Java in Betracht gezogen. Um abzuwiegen, welche Programmiersprache für uns die passendere ist, haben wir eine Nutzwertanalyse, wie im Bild 4.11 ersichtlich ist, erstellt.

Benefit-Analysis		Java		C		Python		Micro-Python	
Beschreibung	Gewicht. [%]	Wert (3)	Total	Wert (3)	Total	Wert (3)	Total	Wert (3)	Total
Umfang	30	4	120	2	60	5	150	3	90
Unterstützung	20	5	100	3	60	5	100	3	60
Analysemöglichkeit	10	3	30	3	30	5	50	4	40
Erlernbarkeit	10	4	40	2	20	4	40	4	40
Verbraucsverhalten	10	4	40	5	50	2	20	4	40
Kenntnisse	10	4	40	2	20	3	30	2	20
Installierbarkeit	10	3	30	3	30	4	40	2	20
			400		**270**		**430**		**310**

Abbildung 4.11: Nutzwertanalyse: Programmiersprachen

	Java	**C**
Umfang	Es sind umfangreiche Biliotheken vorhanden.	Dank Umfangreichen Bibliotheken in jeglichen Bereichen, wie Datenanalyse, Auswertung, Mathematik, IoT hat diese Programmiersprache gewaltigen Einfluss.
Unterstützung	Die Programmiersprache wird überall unterstützt.	Python hat eine enorm grosse Community und ist im Bereich IoT sehr aktiv.
Analyse	Hier ist keine Analyse vorhanden.	Hervorragende Tools zur Analyse sind vorhanden.
Erlernbarkeit	Java ist einfach zu erlernen.	C ist eine eher schwierige und fehleranfällige Sprache.
Verbrauch	Der Verbrauch in Java ist eher gering.	Dank direkter Memory-Nutzung und Low-Level ist C enorm ressourceneffizient und schnell.
Kenntnisse	Es sind gute Vorkenntnisse durch das Studium vorhanden.	Es ist etwas vorhanden.
Installierbarkeit	Für die Installation wird ein Java JVM Environment vorausgesetzt.	Für die Installation wird ein GCC-Compiler benötigt.

	Python	**Micro-Python**
Umfang	Dank umfangreichen Bibliotheken in jeglichen Bereichen, wie Datenanalyse, Auswertung, Mathematik, IoT, kann man extrem viel mit Python erreichen.	Micro-Python enthält nur ein Subset von Python und ist noch in einem Early Stadium.
Unterst.	Python hat eine enorm grosse Community und ist im Bereich IoT sehr aktiv.	Da Micro-Python noch relativ neu ist gibt es nur eine geringere Community.
Analyse	Es sind hervorragende Tools zur Analyse vorhanden.	Es sind zwar Analyse-Tools vorhanden, aber noch eingeschränkt.

	Python	**Micro-Python**
Erlernbar	Python ist schnell und einfach zu erlernen.	MicroPython ist schnell und einfach zu erlernen, da das Prinzip gleich ist wie bei Python.
Verbrauch	Der Ressourcenverbrauch ist im Vergleich zu anderen Sprachen eher hoch. Es können jedoch Teile bei Wunsch in C oder nach C mittels entsprechenden Werkzeugen portiert werden.	In Micro-Python fällt der Ressourcenverbrauch geringer aus als bei Python.
Kenntnisse	Die Kenntnisse zur Sprache sind durchzogen, aber akzeptabel.	Die Kenntnisse zur Sprache sind durchzogen, aber akzeptabel.
Installierbarkeit	Für die Installation wird ein Python-Interpreter benötigt, welcher bei Linux meist standardmässig beigefügt ist.	Hier wird ebenfalls ein Python-Interpreter benötigt, welcher bei Linux meist standardmässig beigefügt ist.

Regel

Als Regeln haben wir Business-Rules und Durable-Rules ausgesucht. Um abzuwiegen, welche Regeln für uns die passenderen sind, haben wir eine Nutzwertanalyse, wie im Bild 4.12 ersichtlich ist, erstellt.

Benefit-Analysis		**Business-Rules**		**Durable-Rules**	
Beschreibung	**Gewicht. [%]**	**Wert (2)**	**Total**	**Wert (2)**	**Total**
Verständlichkeit	40	5	200	3	120
Umfang	30	3	90	5	150
Performance (1)	20	2	40	2	40
Installierbarkeit	10	5	50	5	50
			380		**360**

Abbildung 4.12: Nutzwertanalyse: Regeln

	Business-Rules	**Durable-Rules**
Verständlichkeit	Die Business Rules sind sehr einfach gehalten und gut verständlich.	Durable-Rules ist eine komplexere Rules Engine mit mehr Möglichkeiten und etwas schwierigerer Umsetzung der Anforderungen. Diese Rules Engine enthält Rules als auch Forward Inference (chaining), Flow Structures und Pattern Matching.
Umfang	Der Umfang muss selbst definiert werden, kann jedoch mit etwas Aufwand selbständig erweitert werden.	Durable Rules haben im Vergleich einen erheblich grösseren Umfang mit Rules, Forward Inference (chaining), Flow Structures und Pattern Matching.
Performance	Die Performance ist abhängig von der eigens geschriebenen Zusatzlogik sowie der Verarbeitung von Messages (zur Zeit sychron). Zudem bezieht sich unsere Annahme auf professionellen Anwendungen wie Influx/Kapacitor und Drools.	Die Annahme bezieht sich auf professionellen Anwendungen wie Influx/Kapacitor und Drools.
Installierbarkeit	Die Business-Rules Komponente ist dank Python sehr einfach als Modul einzubetten und zu verwenden.	Die Durable-Rules Komponente ist dank Python sehr einfach als Modul einzubetten und zu verwenden.

Protokoll

Als Kommunikationsprotokoll haben wir uns mittels Nutzwertanalyse für HTTP, CoAP, XMPP oder MQTT entscheiden müssen. Um abzuwiegen, welches Kommunikationsprotokoll für uns das geeignetste ist, haben wir hier ebenfalls eine Nutzwertanalyse, wie im Bild 4.13 ersichtlich ist, erstellt.

Benefit-Analysis		HTTP		CoAP		XMPP		MQTT	
Beschreibung	Gewicht. [%]	Wert (3)	Total	Wert (3)	Total	Wert (3)	Total	Wert (3)	Total
Daten senden	30	3	90	3	90	1	30	3	90
Daten empfangen	30	1	30	3	90	1	30	3	90
Overhead	5	1	5	3	15	1	5	3	15
1/1 Kommunikation	10	3	30	3	30	1	10	2	20
1/N Kommunikation	10	1	10	1	10	1	10	2	20
Kommunikationsweg	5	1	5	1	5	1	5	3	15
Programmiersprachen	5	2	10	2	10	2	10	3	15
Erlernbarkeit	5	2	10	3	15	2	10	3	15
			190		**265**		**110**		**280**

Abbildung 4.13: Nutzwertanalyse: Protokolle

	HTTP	**CoAP**
Daten senden	Die Daten können einfach übermittelt werden.	Ein serverseitiger Push ist bei Änderungen von Ressourcen definierbar.
Daten empfangen	Es ist keine Server-Push Kommunikation nativ möglich.	CoAP implementiert das Request / Response Pattern.
Overhead	Bei HTTP gibt es viel Protokoll-Overhead.	Bei CoAP gibt es wenig Header.
1/1 Komm.	Eine 1 zu 1 Kommunikation ist möglich.	Eine 1 zu 1 Kommunikation ist möglich.
1/N Komm.	Eine 1 zu N Kommunikation ist nicht möglich.	Eine 1 zu N Kommunikation ist nicht möglich.
Komm.-Weg	Der Kommuinikationsweg ist unidirektional.	Der Kommuinikationsweg ist nicht bidirektional
Prog.-Sprache	HTTP unterstützt fast alle Programmiersprachen.	Auch CoAP unterstützt fast alle Programmiersprachen.
Erlernbarkeit	HTTP ist einfach zu erlernen.	CoAP ist sehr effizient und man hat einen leichte Umstieg von HTTP auf CoAP.

	XMPP	**MQTT**
Daten senden	XMPP wird nicht mehr gross eingesetzt.	MQTT implementiert das Publish / Subscribe Pattern.
Daten empfangen	XMPP wird nicht mehr gross eingesetzt.	MQTT implementiert das Publish / Subscribe Pattern.

	XMPP	**MQTT**
Overhead	XMPP generiert einen grossen Protokoll-Overhead.	MQTT verursacht hingegen wenig Protokoll-Overhead.
1/1 Komm.	Es sind viele Features vorhanden, die für Instant Messenger- Anwendungsfälle optimiert sind.	Publish / Subscribe
1/N Komm.	Es sind viele Features vorhanden, die für Instant Messenger-Anwendungs- fälle optimiert sind.	MQTT implementiert das Publish / Subscribe Pattern.
Komm.-Weg	Es gibt eine hohe Fragmentierung bei den Server- und Client-Features.	Bei MQTT ist eine echte Push-Kommunikation möglich.
Prog.-Sprache	XMPP ist für viele Sprachen verfügbar.	MQTT unterstützt alle gängigen Programmiersprachen.
Erlernbarkeit	XMPP basiert auf XML und ist erweiterbar.	MQTT verfügt über hohe Skalierbarkeit, viele Protokoll-Features und ist für ressourcenbeschränkte Geräte geeignet.

4.2.3 Datenspeicherung

Die Daten werden alle In-Memory in Redis gespeichert. Dies bringt den Vorteil einer enormen Performancesteigerung, da die Daten stets In-Memory gehalten werden. Bei einem Ausfall oder Reboot des System ist es möglich, die Daten zu persistieren oder von Zeit zu Zeit Snapshots davon zu erzeugen. Da wir auf Embedded Hardware arbeiten und dabei oft auf Flashspeicher zurückgegriffen wird, ist es ratsam, nicht allzu oft auf die Disk zu schreiben. Daher bietet sich die Möglichkeit der Snapshots an. Dabei muss ein Tradeoff in Kauf genommen werden, dass gewisse Daten verloren gehen.

Da der Kunde die Daten bei Bedarf selber abholt, kann auf eine Persistierung auf dem Gerät jedoch verzichtet werden.

Redis bietet eine grosse Anzahl an Datentypen an - von der einfachen Key/Value Speicherung über Redis Listen, Redis Sets und Redis Hashes bis hin zu Redis Streams. Bei Redis kann dabei jeweils mittels **Key** auf die Daten zugegriffen werden. Es gibt die Möglichkeit mittels Redis Command-Line-Interface **redis-cli** abfragen zu formulieren, um Daten auszulesen. Beispiele von solchen Abfragen sind in nachfolgendem Listing 4.2 ersichtlich.

```
user@ubuntu:~$ redis-cli
127.0.0.1:6379> get 'SENSOR_ID'
127.0.0.1:6379> xrange 'sensordata' '-' '+'
127.0.0.1:6379> smembers 'sensorlist'
127.0.0.1:6379> hgetall 'rules'
127.0.0.1:6379> hget 'rules' 'RULE_ID'
127.0.0.1:6379> get 'ruledata'
```

Listing 4.2: Redis CLI Befehle

Bei einem einfachen Beispiel wird ein Key/Value-Paar gespeichert, wie es exemplarisch anhand der Tabelle 4.7 gezeigt wird.

Der *redis-cli* Befehl zur Abfrage eines Redis Key/Value-Paares lautet **get**.

key	value

Tabelle 4.7: Redis key/value Paar

Ein einfaches unsortiertes Set in Redis kann exemplarisch wie in nachfolgender Tabelle 4.8 dargestellt werden.

Der *redis-cli* Befehl zur Abfrage eines Redis Sets lautet **smembers KEY**.
Auf der nächsten Tabelle 4.9 sieht beispielsweise ein HashSet folgendermassen aus.

key
value1
value2

Tabelle 4.8: Redis Set

Mittels *key* und *field* kann dabei direkt auf eines der Daten dieser Redis HashSet zugegriffen werden.

Der *redis-cli* Befehl zur Abfrage eines Redis HashSets lautet **hgetall KEY** oder für einzelne Felder in einem bestimmten Key **hget KEY FIELD**.

key	
field1	value1
field2	value2

Tabelle 4.9: Redis HashSet

Die Darstellung von Streams ist exemplarisch in Nachfolgender Tabelle 4.10 ersichtlich.

Der *redis-cli* Befehl zur Abfrage eines Redis Streams lautet **xrange KEY START END [COUNT count]**. Beispielsweise ist es möglich den Anfang und das Ende mit Minus (-) und Plus (+) anzugeben, welche sich auf die minimale, respektive Maximale Sequenznummer (ID) beziehen. Man kann ausserdem eine bestimmte Anzahl mit *count* angeben. Es ist auch möglich Sequenznummern als Anfangs- und Endwerte anzugeben.

key	
1) Sequence1 (ID1)	values1
2) Sequence2 (ID2)	values2

Tabelle 4.10: Redis Streams

Nachfolgend werden nun verschiedene Speicherorte unterschiedlicher Daten aufgezeigt, auf die mittels den vorher beschriebenen Befehlen und Schlüsseln zugegriffen werden kann.

Geräte

Der Datensatz der Sensorliste wird auf der Tabelle 4.11 dargestellt.

sensorlist
sensor-id-1
sensor-id-2

Tabelle 4.11: Sensorliste

Die Tabelle 4.12 zeigt den Datensatz der Aktuatorliste an.

actuatorlist
actuator-id-1
actuator-id-2

Tabelle 4.12: Aktuatorliste

Regeln

Die Regeln werden ebenfalls mittels Redis HashSet gespeichert und beinhalten als *field* die Rule-ID.

rules	
rule-id-1	rule-1
rule-id-2	rule-2

Tabelle 4.13: Rules

Die Rule Daten, also die möglichen zu setzenden Werte für Regeln, sind ebenfalls in Redis gespeichert. Der Datensatz in Redis sieht wie nachfolgend in 4.14 beschrieben aus.

Daten

Jeder Sensor speichert in Redis als Key/Value Paar unter seiner eigenen individuellen Device-ID(UUID) als Key die letzten übermittelten Metadaten und Werte, wie nachfolgendes exemplarisches Beispiel 4.15 illustriert.

sensor-id	[Metadata/Value]

Tabelle 4.15: Redis Device Daten als key/value Paar

ruledata	[Parameter/Conditions/Actions]

Tabelle 4.14: Redis ruledata als Key/Value Paar

Ausserdem werden in Redis unter dem Key *sensordata* alle erhaltenen Sensordaten chronologisch in einem Stream gehalten, welche bei Bedarf auf verschiedene Weise mittels Endpoints abgerufen werden kann. Beispielsweise ist es möglich, eine gewisse Anzahl an Daten oder auch über einen gewissen Zeitraum Daten abzuholen. Da ein kontinuierlicher Stream an Daten generiert wird, ist es dank einer einzigartigen ID möglich (welche standardmässig aus dem Unix Zeitstempel in Millisekunden besteht und mit einer Sequenznummer am Schluss aufgebaut ist) Daten gezielt abzufragen.

<millisecondsTime>-<sequenceNumber>

Tabelle 4.16: Stream ID

Da die ID aus dem Unix Zeitstempel besteht, werden diese auch gerade chronologisch geordnet gespeichert und nacheinander ausgegeben, wie die nachfolgende Tabelle 4.17 zeigt.

Seq1
Value1
Seq2
Value2

Tabelle 4.17: Sensordaten Stream

Eine mögliche Ausgabe kann in Listing 4.3 betrachtet werden.

```
1) 1) "1556101549024-0"
   2)  1) "id"
    2) "1057c680-633a-11e9-8abf-a7cb277ba226"
    3) "value"
    4) "145"
    5) "lat"
    6) "90"
    7) "lng"
    8) "10"
    9) "unit"
     10) "km/h"
     11) "type"
     12) "speed"
     13) "description"
     14) "Tempo"
2) 1) "1556101550005-0"
   2)  1) "id"
```

```
    2) "1132fac0-633a-11e9-9a30-23bf466785d0"
    3) "value"
    4) "340"
    5) "lat"
    6) "50"
    7) "lng"
    8) "50"
    9) "unit"
    10) "C"
    11) "type"
    12) "temperature"
    13) "description"
    14) "Motor"
3) 1) "1556101551005-0"
 2)  1) "id"
    2) "d7e72836-6339-11e9-a8aa-27b2c181f989"
    3) "value"
    4) "26"
    5) "lat"
    6) "90"
    7) "lng"
    8) "90"
    9) "unit"
     10) "%"
     11) "type"
     12) "humidity"
     13) "description"
     14) "Indoor"
```

Listing 4.3: Beispielausgabe Sensordaten Stream

4.2.4 API Endpunkte

In diesem Abschnitt werden die wichtigsten API Endpunkte zwischen dem Benutzer und dem Gateway beschrieben.

Startseite mit Endpunkten	
Description	Die Startseite mit den verschiedenen Endpunkten.
Request	
URL	GET http://localhost:5000/
Response	
Description	Rückgabe der Startseite mit den Endpunkten.
Content	HTML
Success	200: Mainpage with Endpoints
Failure	400: An error occurred
	500: SystemError

Tabelle 4.18: API Frontpage

Swagger API User Interface	
Description	Zeigt die API Dokumentation mittels Swagger an.
Request	
URL	GET http://localhost:5000/apidocs
Response	
Description	Rückgabe der Swagger API Dokumentation.
Content	HTML
Success	200: Swagger API User Interface
Failure	400: An error occurred
	500: SystemError

Tabelle 4.19: API Swagger

Regel User-Interface	
Description	Ein User-Interface um Regeln zusammenzustellen.
Request	
URL	GET http://localhost:5000/ui
Response	
Description	Rückgabe des Regel User-Interfaces.
Content	HTML
Success	200: Rule User Interface
Failure	400: An error occurred
	500: SystemError

Tabelle 4.20: API Rule UI

Sensorlist holen	
Description	Beinhaltet alle angeschlossenen Sensoren und gibt die ID der Geräte zurück.
Request	
URL	GET http://localhost:5000/sensorlist
Response	
Description	Rückgabe der Sensor IDs im JSON Format.
Content	JSON
Success	200: List of SensorIDs
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.21: API Sensorlist

Aktuatorlist holen	
Description	Beinhaltet alle angeschlossenen Aktuatoren und gibt die ID der Geräte zurück.
Request	
URL	GET http://localhost:5000/actuatorlist
Response	
Description	Rückgabe der Aktuator IDs im JSON Format.
Content	JSON
Success	200: List of ActuatorIDs
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.22: API Aktuatorlist

Device Info holen	
Description	Gibt die aktuellen Werte des Sensors zurück.
Request	
URL	GET http://localhost:5000/sensor/<sensor_id>
Response	
Description	Metadaten inkl. letzter gespeicherter Wert als JSON.
Content	JSON
Success	200: SensorValues
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.23: API Device Info

Regeln holen	
Description	An diesen Endpunkt können alle gesetzten und gültigen Regeln abgerufen werden.
Request	
URL	GET http://localhost:5000/rules
Response	
Description	Rückgabe aller Regeln als Liste im JSON Format.
Content	JSON
Success	200: List of all valid Rules
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.24: API Regeln holen

Regel holen	
Description	An diesem Endpunkt kann eine spezifische Rule im JSON Format abgerufen werden.
Request	
URL	GET http://localhost:5000/rules/<rule_id>
Response	
Description	Die Regel im JSON Format.
Content	JSON
Success	200: Rule with Rule-ID
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.25: API Regel holen

Regel erstellen	
Description	An diesen Endpunkt kann eine neue Rule im JSON Format gesendet werden.
Request	
URL	POST http://localhost:5000/rules
Content	JSON
Response	
Description	Bei Erfolg Rückgabe der Rule mit Rule-ID oder Fehler.
Content	JSON
Success	200: Rule with Rule-ID
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.26: API Regel erstellen

Regel updaten	
Description	An diesem Endpunkt kann eine Rule im JSON Format aktualisiert werden.
Request	
URL	PUT http://localhost:5000/rules/<rule_id>
Content	JSON
Response	
Description	Bei Erfolg Rückgabe der Rule mit Rule-ID oder Fehler.
Content	JSON
Success	200: Rule with Rule-ID
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.27: API Regel updaten

Regel löschen	
Description	An diesem Endpunkt kann eine Regel gelöscht werden.
Request	
URL	DELETE http://localhost:5000/rules/<rule_id>
Content	JSON
Response	
Description	Bei Erfolg Rückgabe der Rule mit Rule-ID oder Fehler.
Content	JSON
Success	200: Rule with Rule-ID
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.28: API Regel löschen

Rule Daten holen	
Description	Über diesen Endpunkt können alle möglichen Variablen, Parameter, Bedingungen und Aktionen abgerufen werden. Dies sind gültige Werte, welche man zur Erstellung von Regeln setzen kann.
Request	
URL	GET http://localhost:5000/ruledata
Response	
Description	Rückgabe der Parameter, Bedingungen und Aktionen im JSON Format.
Content	JSON
Success	200: List of Parameters, Conditions and Actions
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.29: API Rule Daten

SensorMessage Stream holen	
Description	Über diesen Endpunkt können alle Sensordaten, welche als Stream in Redis gespeichert sind abgerufen werden.
Request	
URL	GET http://localhost:5000/streams
Response	
Description	Chronologisch geordnete SensorMessages als Stream im JSON Format.
Content	JSON
Success	200: List of SensorMessages as Stream
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.30: API SensorMessage Stream holen

SensorMessage Stream nach ID holen	
Description	Über diesen Endpunkt können SensorMessages über einen bestimmten Sensor nach ID gefiltert abgerufen werden.
Request	
URL	GET http://localhost:5000/streams/filter/id/<filter_by_id>
Response	
Description	Chronologisch geordnete SensorMessages als Stream im JSON Format.
Content	JSON
Success	200: List of SensorMessages as Stream
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.31: API SensorMessage Stream nach ID holen

SensorMessage Stream nach Typ holen	
Description	Über diesen Endpunkt können SensorMessages über einen bestimmten Sensor nach Typ gefiltert abgerufen werden.
Request	
URL	GET http://localhost:5000/streams/filter/type/<filter_by_type>
Response	
Description	Chronologisch geordnete SensorMessages als Stream im JSON Format.
Content	JSON
Success	200: List of SensorMessages as Stream
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.32: API SensorMessage Stream nach Typ holen

SensorMessage Stream anhand Beschreibung holen	
Description	Über diesen Endpunkt können SensorMessages über einen bestimmten Sensor gefiltert nach Beschreibung abgerufen werden.
Request	
URL	GET http://localhost:5000/streams/filter/description/<filter_by_desc>
Response	
Description	Chronologisch geordnete SensorMessages als Stream im JSON Format.
Content	JSON
Success	200: List of SensorMessages as Stream
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.33: API SensorMessage Stream anhand Beschreibung holen

SensorMessage Stream anhand Einheit holen	
Description	Über diesen Endpunkt können SensorMessages über einen bestimmten Sensor gefiltert nach Einheit abgerufen werden.
Request	
URL	GET http://localhost:5000/streams/filter/unit/<filter_by_unit>
Response	
Description	Chronologisch geordnete SensorMessages als Stream im JSON Format.
Content	JSON
Success	200: List of SensorMessages as Stream
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.34: API SensorMessage Stream anhand Einheit holen

SensorMessage Stream anhand Wert holen	
Description	Über diesen Endpunkt können SensorMessages über einen bestimmten Sensor gefiltert nach Wert abgerufen werden.
Request	
URL	GET http://localhost:5000/streams/filter/value/<filter_by_value>
Response	
Description	Chronologisch geordnete SensorMessages als Stream im JSON Format.
Content	JSON
Success	200: List of SensorMessages as Stream
Failure	400: ParameterError
	404: ResourceError
	500: SystemError

Tabelle 4.35: API SensorMessage Stream anhand Wert holen

4.2.5 Handhabung Messages

Grundsätzlich werden alle Sensordaten in Redis als Streams gespeichert. Dabei kann definiert werden, wie viele Daten im Speicher bleiben, bevor sie automatisch entfernt werden. Damit wird ein Volllaufen des begrenzten Speichers verhindert.

Der letzte Wert, eines jeden angeschlossenen Sensors, wird unabhängig voneinander als eigener Eintrag gehalten und regelmässig erneuert. Ausserdem werden aus den Daten die eindeutigen Identifikationen der angeschlossenen Geräte entnommen und in Redis gehalten, um jeweils eine aktuelle Liste der angeschlossenen Sensoren und Aktuatoren in einem eigenen Eintrag zu halten. Weiter kann Redis verwendet werden, um Daten zu aggregieren und diese Werte mittels Redis in einem eigenen Eintrag zu halten, um anschliessend darauf wieder zuzugreifen.

4.2.6 Handhabung Regeln

Regeln bestehen aus Bedingungen (Conditions) und Aktionen (Actions), welche beliebig verschachtelt werden können. Dabei können komplexe Konstrukte von Regeln erstellt werden, bei denen *Und (All)* und *Oder (Any)* miteinander verknüpft werden können. So werden beliebige algebraische Strukturen gebildet.

Einige Beispiele von Regeln im JSON-Format werden im nachfolgenden anhand von ausgewählten User Storys aufgezeigt und beschrieben.

Regel User Story 4: Positionsbestimmung mittels GPS und Odometer

Hier haben wir zwei Möglichkeiten für die Erstellung einer Regel. In dieser User Story geht es darum, eine aktuelle Position zu erhalten und anhand dieser Daten eine beliebige Aktion auszuführen. Im ersten Beispiel tracken wir die Daten, sobald die Kondition mit der Beschreibung *GPS* auf einen Sensor zutrifft.

```
[{
  "conditions": {
    "all": [{
      "name": "description",
      "operator": "equals",
      "value": "gps",
      "params": {
        "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e"
      }
    }]
  },
  "actions": [{
    "name": "track_data"
  }]
```

```
}]
```

Anders als beim ersten Beispiel tracken wir hier die Daten erst, wenn die Longitude und Latitude einen Wert beinhalten. Wenn diese zwei Variablen *None* beinhalten, handelt es sich um einen anderen Sensor.

```
[{
  "conditions": {
    "all": [{
      "name": "current_value",
      "operator": "equals_not",
      "latitude": "None",
      "params": {
        "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e"
      },
      {
      "name": "current_value",
      "operator": "equals_not",
      "longitude": "None",
      "params": {
        "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e"
      }
    }]
  },
  "actions": [{
    "name": "track_data"
  }]
}]
```

Regel User Story 5: Einsatz Geofencing

Beim Einsatz von Geofencing könnten wir, wie im unteren Regel-Beispiel ersichtlich, eine Push-Nachricht an einen Benutzer senden, sobald eine gewisse Position erreicht wird. Hier wird die Aktion aber nur bei **einer einzigen Position** ausgelöst. Dieses Ereignis wird wahrscheinlich äusserst selten auftreten.

```
[{
  "conditions": {
    "all": [{
       "name": "current_value",
        "operator": "equal_to",
        "latitude": 47.223453,
        "params": {
          "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e"
        },
        "name": "current_value",
        "operator": "equals",
        "longitude": 8.817309,
```

```
        "params": {
          "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e"
        }
      }
    }]
  },
  "actions": [{
    "name": "send_push_message",
    "params": {
      "message": "Geofence at HSR building 1"
    }
  }]
}]
```

Zudem wäre eine andere Möglichkeit, einen Radius anzugeben und so einen virtuellen *Zaun* (Geofence) zu erstellen. Der Radius wird allerdings in SI-Einheiten (km) angegeben und dafür wird natürlich auch die entsprechende Ausgangsposition (zum Beispiel die aktuelle Position) benötigt.

```
[{
  "conditions": {
    "all": [{
      "name": "description",
      "operator": "equal_to",
      "value": "gps",
      "params": {
        "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e"
      }
    }]
  },
  "actions": [{
    "name": "check_geofence_circle",
    "params": {
      "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e",
      "radius": 10,
      "center_latitude": 47.223498264901025,
      "center_longitude": 8.817343711853027,
      "message": "Geofence circle at HSR building 1"
    }
  }]
}]
```

Listing 4.4: Geofence Regel in einem Umkreis

Regel User Story 6: Temperaturschwellwert und Alarm

Dieses Beispiel zeigt eine einfache Regel zur Überwachung eines Temperaturwerts. Sobald die Temperatur mehr als 30 C° misst, wird ein Alarm ausgelöst. Die Regel wird identisch wie bei der User Story 16 aufgebaut. Auch hier wird der Alarm als Konsolenausgabe übermittelt.

```
[{
  "conditions": {
    "all": [{
      "name": "current_value",
      "operator": "greater_than",
      "value": 30,
      "params": {
        "sensorid": "Sensor1"
      }
    }]
  },
  "actions": [{
    "name": "console_output"
  }]
}]
```

Regel User Story 13: Echtzeitdaten Position

Mit diesem Beispiel möchten wir aufzeigen wie eine Regel aussehen könnte, um Echtzeitdaten fortlaufend aufzuzeichnen. Man möchte beispielsweise die aktuelle Position tracken. Das funktioniert im Prinzip gleich wie bei der User Story 4.

```
[{
  "conditions": {
    "all": [{
      "name": "description",
          "operator": "equals",
          "value": "GPS-sensor",
          "params": {
            "sensorid": "Sensor2"
          }
      }
    ]}
  },
  "actions": [{
    "name": "track_data"
  }]
}]
```

Regel User Story 16: Motorüberwachung

In diesem Beispiel wird eine einfache Regel zur Überwachung eines Wertes erstellt. Wenn der *Sensor1* den Wert *150* überschreitet, soll eine Meldung ausgegeben werden.

```
[{
  "conditions": {
```

```
    "all": [{
      "name": "current_value",
      "operator": "greater_than",
      "value": 150,
      "params": {
        "sensorid": "Sensor1"
      }
    }]
  },
  "actions": [{
    "name": "console_output"
  }]
}]
```

Regel User Story 22: Datenkorrelation

In diesem Beispiel wird eine Regel erstellt, bei welcher geprüft wird ob der *Sensor1* einen Wert grösser als *150* hat und ob *Sensor2* gleichzeitig einen Wert kleiner als *50* hat. Treffen beide Bedingungen zu, soll eine Aktion mit einer Meldung ausgegeben werden.

Hinweis: Durch das zusätzliche Feld *params* ist es möglich, gezielt Werte von einem bestimmten Sensor zu überprüfen. Damit ist es auch möglich Regeln zu erstellen, bei welchem mehrere Sensoren gleichzeitig involviert sind.

```
[{
  "conditions": {
    "all": [{
      "name": "current_value",
      "operator": "greater_than",
      "value": 150,
      "params": {
        "sensorid": "Sensor1"
      }
    },
    {
      "name": "current_value",
      "operator": "less_than",
      "value": 50,
      "params": {
        "sensorid": "Sensor2"
      }
    }]
  },
  "actions": [{
    "name": "console_output"
  }]
}]
```

4.2.7 Handhabung Filter

Bezüglich der Filterung bieten sich verschiedene Möglichkeiten an. So besteht die Möglichkeit, mittels Endpoints vordefinierte Filter auf den Datenstream anzuwenden oder Regeln zum setzen von Filtern zu erstellen. Weiter denkbar sind Filter bei der ersten Instanz oder dem Service ()der die Daten des Brokers entgegennimmt) zu setzen, zu implementieren und beispielsweise eine Zustandsänderung zu verfolgen. Ausserdem könnte man eine Vorfilterung (je nach Art der Daten) typspezifisch setzen. Im folgenden werden diese Möglichkeiten noch im Detail beschrieben.

Endpoints Mittels vordefinierten Endpoints ist es gegebenfalls möglich, über Zeit, Anzahl, Typ oder Identifikation zu filtern.

Regeln Mittels Regeln wäre es möglich, verschiedene Filter auf typspezifische Daten oder Sensoren zu setzen.

Zustandsänderung Denkbar wäre auch eine Zustandsänderungsverfolgung, bei welcher nur Werte weitergegeben und in Redis Datenbank gespeichert werden. Dies wenn sich tatsächlich ein Wert ändert oder auch nur, wenn ein gewisser Unterschied zu vorher (also ein Deltawert) überschritten wird.

Typspezifisch Anhand der Art der Daten (also Typ-bezogen) können verschiedene Filter gesetzt werden. Beispielsweise liefern GPS Daten relativ häufig frequentierte Datenwerte, welche es nicht zwingend jede Sekunde zu speichern gilt. Da ergibt es sich, diese Werte etwas zu sammeln und nur in einem gewissen Zeitintervall weiterzugeben. Entweder wird diese Logik schon im Voraus festgelegt oder man bestimmt diese selbst mittels Regeln.

5 Realisierung

5.1 Umgebungen

Es gibt drei verschiedene Umgebungen, auf denen wir uns bewegen. Namentlich zu nennen sind die Entwicklungs-, die Test- sowie Produktionsumgebung. Die Entwicklungsumgebung ist für die Entwicklung des Codes auf unseren privaten Rechnern mittels einer virtuellen Machine eingerichtet worden. Die Testumgebung, um den Code und das Setup zu testen, läuft auf einem Raspberry Pi 3 Model B mit vorinstalliertem Raspbian Betriebssystem.

Die Produktionsumgebung beherbergt das produktive System und ist auf dem netModule NB2800 Router eingerichtet. Dort sollte praktischerweise die produktive Version laufen. Dadurch, dass diese Hardware jedoch relativ spät erschienen ist und die Hardware sich nur um die Anzahl der Prozessorkerne unterscheidet, konnte weiterhin auf dem Raspberry Pi 3 gearbeitet werden. Zur Vollständigkeit wird jedoch noch ein Setup auf dem netModule Router mittels Virtualisierung und LXC Linux Containern beschrieben. Der Aufbau ist im Prinzip der gleiche wie auf dem Raspberry Pi.

5.1.1 Entwicklungsumgebung

Eine Entwicklungsumgebung wurde für jeden Entwickler eigenständig mittels Linux VM, Docker, Docker-Compose sowie den benötigten Abhängigkeiten aufgesetzt. Die VM basiert auf einem Ubuntu 16.04 auf welchem Docker, Docker-Compose und VSCode [44] mit diversen Extensions [45] eingerichtet wurde.

Mittels GitLab (als Versionierungs-Tool) und einem voll funktionierenden Continuous Integration (CI) Zyklus, kann anhand verschiedener Jobs - wie Linting (flake8) und Testing (pytest) - eine saubere und möglichst fehlerfreie Software garantiert werden. Es ist zudem möglich automatisiert oder manuel mittels Continuous Delivery/Deployment (CD) lauffähige Software auf bestimmte Hardware zu schicken und dort mittels GitLab Runner Shell auszuführen.

Mittels Issue Tracking und Merge Requests und dabei durchlaufender Pipelines mit Linting und Testing, wird sichergestellt, dass nur der Code, welcher den Richtlinien (Python Coding Guidelines) entsprechen sowie sauber durchlaufende Tests bestehen, in den Master hizugefügt wird. Die Anleitung sowie Details und Scripte zur Aufsetzung der Entwicklungsumgebung, dem GitLab Runner sowie der GitLab CI/CD Pipeline Konfiguration kann

im Anhang unter 7.6 nachgeschlagen werden.

5.1.2 Testumgebung

Das Deployment auf Raspberry Pi wurde mittels GitLab Runner (bereits auf dem Gerät installiert) [46] [47], laufender Shell sowie einem GitLab Deploy Job umgesetzt. Genaue Angaben zum Setup sowie der Anleitung zur Einrichtung können im Anhang unter 7.6 nachgeschlagen werden.

Raspberry Pi 3

Die Hardware kann von der RaspberryPi Website [48] sowie aus nachfolgender Tabelle 5.1 entnommen werden.

Typ	Raspberry Pi 3 Model B
Prozessor	1.2Ghz ARM
Kerne/Threads	**4**/1
RAM	1.0 GB
Speicher	16.0 GB Flash SD Card

Tabelle 5.1: Raspberry Pi 3 Model B Spezifikationen

5.1.3 Produktionsumgebung

Das gleiche Deployment auf den netModule Router wurde wie beim Raspberry Pi mittels GitLab Runner, laufender Shell sowie einem GitLab Deploy Job umgesetzt. Genaue Angaben zum Setup sowie der Anleitung zur Einrichtung können ebenfalls im Anhang unter 7.6 nachgeschlagen werden.

Einrichtung netModule NB2800

Die Hardware des netModule NB2800 Routers konnte durch die netModule Website [49], dem netModule Wiki FAQ [50] sowie dem Industriepartner in nachfolgender Tabelle 5.2 auf folgende Werte festgelegt werden.

Typ	netModule NB2800
Prozessor	1.3Ghz ARM
Kerne/Threads	**2**/1
RAM	1.0 GB
Speicher	4.0 GB Flash SD Card

Tabelle 5.2: netModule NB2800 MultiVehicle Spezifikationen

Das Thema Virtualisierung sowie LXC wurde bereits im Abschnitt 3.2.4 erwähnt und wird hier nicht mehr weiter erläutert.

In diesem Abschnitt wird deutlicher auf die Installation der Lizenz und das Setup des LXC Containers eingegangen.

Voraussetzung zur Arbeit in LXC Linux Containern auf einem netModule NB2800 Router ist es, im Besitz einer gültigen Lizenz zum Betreiben einer Virtualisierung zu sein [51]. Eine Testlizenz wurde uns freundlicherweise zur Verfügung gestellt.

Beim ersten Aufsetzen werden wir mit einem Assistenten begrüsst, bei welchem wir lediglich ein Passwort für den Administrator User **admin** setzen müssen, danach gelangen wir wie in Bild 5.1 zu sehen ist auf das Home User-Interface.

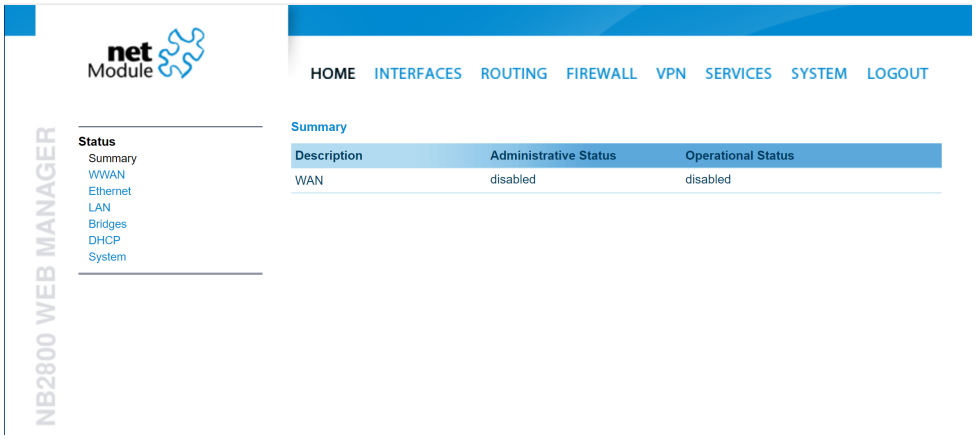


Abbildung 5.1: netModule NB2800 Home User-Interface

Von dort gelangen wir via **System** an den Ort, an welchem wir den Tab **Virtualization** sowie **Licensing** sehen sollten, um eine neue Lizenz hinzuzufügen. Da wir den Tab Virtualisierung noch nicht sehen, laden wir kurzerhand die uns via E-Mail gelieferte Test-Lizenz hoch. Nach dem Aktualisieren der Lizenz und dem Neustart der Dienste, sehen wir unseren Virtualisierungs-Tab. Dies ist im Bild 5.2 ersichtlich.

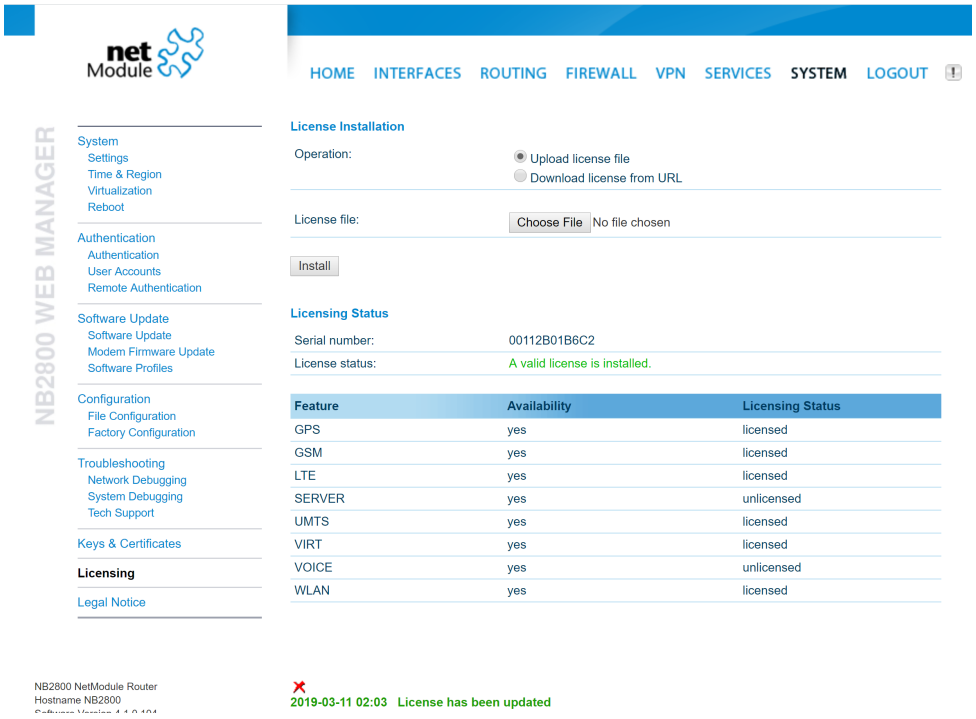


Abbildung 5.2: netModule NB2800 System und Lizenz User-Interface

Wir wechseln schliesslich weiter zu **System - Virtualisation**, wo uns folgendes Bild 5.3 erwartet.

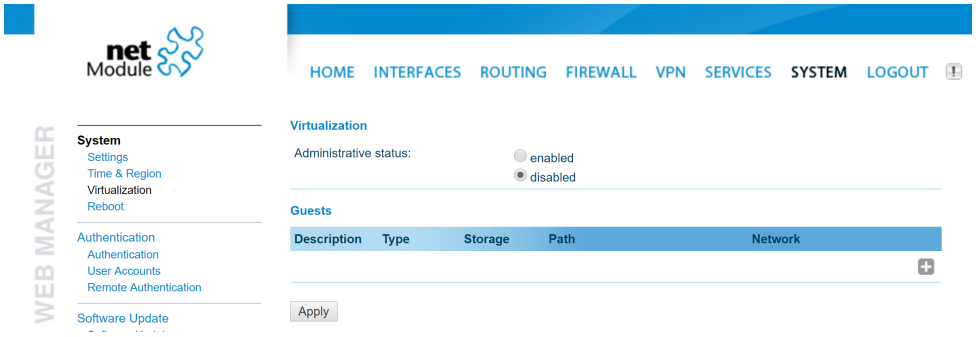


Abbildung 5.3: netModule NB2800 Virtualisierungs User-Interface

Mithilfe folgender Anleitung [52] richten wir uns einen neuen Ubuntu LXC Linux Container ein. Dafür klicken wir auf das Pluszeichen und folgen den Anweisungen.

Wir entscheiden uns für das automatische Setup, da wir im linken unteren Feld sehen, das unsere Version 4.1.0.104 dies unterstützt. Damit kann man den Pfad einfachheitshalber angeben, ohne dass man den Container herunterladen und den Pfad manuell eingeben muss. Dafür müssen wir den netModule Router noch schnell an das Netz hängen und können dan folgenden Link dazu verwenden `ftp://share.netmodule.com/router/public/virt/ubuntu_bionic.tar.xz`.

Bevor wir jedoch fortfahren können, müssen wir unter **Interfaces - Ethernet - IP Settings** den LAN2 Port von LAN zu WAN wechseln, um von dort ins World-Wide-Web zu gelagen. Wir nehmen LAN2, da wir lokal via PC schon auf LAN1 verbunden sind und unsere Arbeiten darauf ausführen. Wir stellen sicherheitshalber die Firewall auf ON und verbieten jeglichen Traffic von ausserhalb. Über **System - Troubleshooting - Network Debugging** können wir noch mittels eines Pings auf beispielsweise 8.8.8.8 prüfen, ob wir verbunden sind.

Da wir den internen Flash Storage Speicher nicht aufrufen konnten, haben wir uns kurzerhand entschieden, eine externe USB Festplatte anzuschliessen, auf welchem wir den Container installieren können.

Anschliessend können wir wie in der Anleitung [52] beschrieben, weiter fortfahren. Das Ergebnis dazu sehen wir anschliessend in folgendem Bild 5.4.

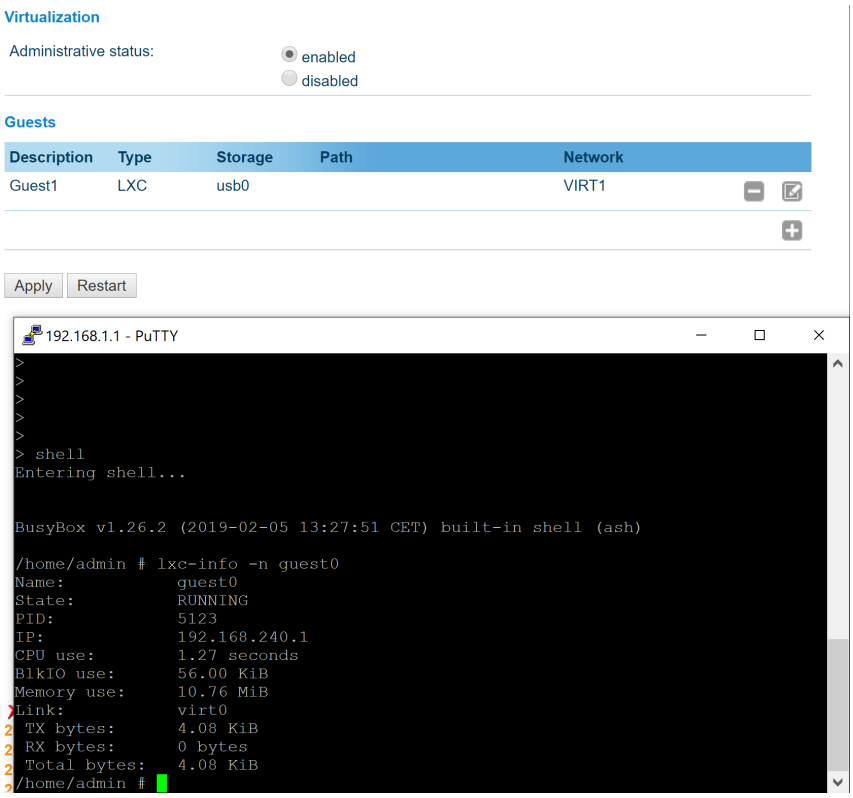


Abbildung 5.4: netModule NB2800 Container erstellt

Die Installation eines Gitlab Runners auf dem netModule Router ist identisch wie auf dem Raspberry Pi 3.

5.2 Systemübersicht

In nachfolgendem ASCII Diagramm 5.5 ist ersichtlich, wie die einzelnen umgesetzten Dienste und Komponenten miteinander interagieren.

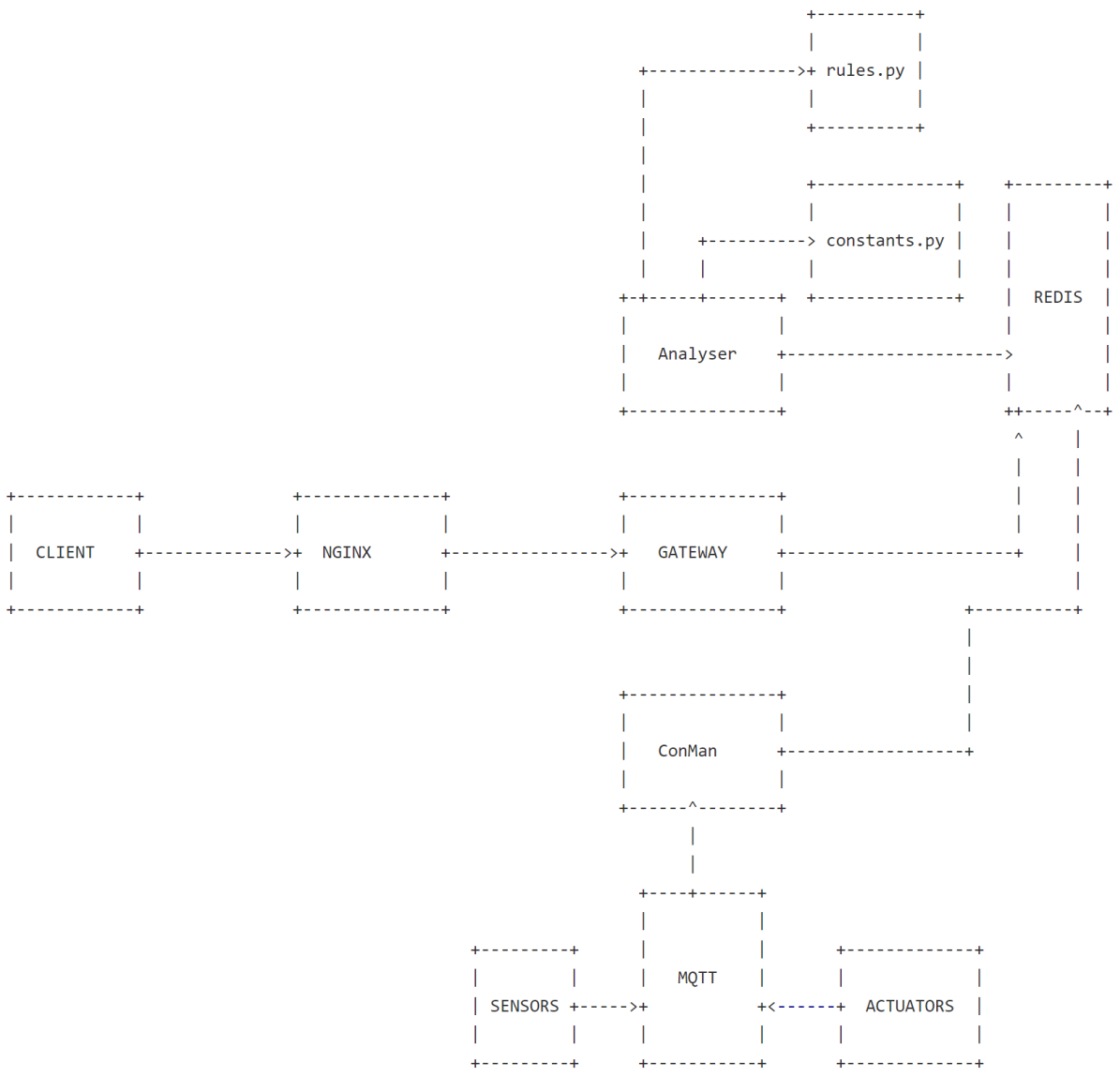


Abbildung 5.5: Systemübersicht ASCII

5.3 Technologien

Neben einem Linux Betriebssystem, welches ARM Prozessoren sowie Python3 unterstützt, benötigen wir ausschliesslich Mosquitto als MQTT Broker sowie Redis als Datenbank.

5.3.1 MQTT

Es wurde **Mosquitto** als MQTT Message Broker verwendet. Genaue Details zu MQTT, dessen Vor- und Nachteile, könnnen im Abschnitt 4.2.2 nachgelesen werden.

Adresse	Port
127.0.0.1	1883

Tabelle 5.3: Mosquitto MQTT Adresse und Port

5.3.2 Redis

Zur Speicherung der Daten und als Caching wurde die In-Memory Datenbank **Redis** verwendet. Genaue Details zu Redis, dessen Vor- und Nachteile, können ebenfalls im Abschnitt 4.2.2 nachgelesen werden.

Adresse	Port
127.0.0.1	6379

Tabelle 5.4: Redis Adresse und Port

5.4 Dienste

In diesem Abschnitt werden die einzelnen Dienste wie der Gateway, ConMan, Analyser sowie die Simulatoren genauer beschrieben.

5.4.1 Gateway

Der Gateway ist unser Einstiegspunkt für den Anwender, welche alle möglichen Endpunkte definiert, mit Regeln umgehen kann und Abfragen zu Messages und Gerätelisten ermöglicht. Verschiedene REST-konforme Schnittstellen wurden dazu definiert, welche nachfolgend noch im Detail beschrieben werden.

Der Gateway läuft als Python Flask Applikation und beinhaltet eine Swagger API Definition der REST API Endpunkte.

Um zu zeigen, dass es auch in Python möglich ist, eine saubere Architekture zu erstellen, objektorientiert zu arbeiten sowie mittels Tests Fehler zu minimieren, wurde der Gateway Service aus dem Prototyp extrahiert und in einem eigenen Repository so weiter entwickelt, dass er mittels einer testgetriebenen Entwicklung (test-driven development - TDD) umgesetzt werden konnte. Dafür wurde das in Python vorgesehene *pytest* [53] Modul zum Unit-Testing verwendet.

Struktur

In diesem Abschnitt wird der Aufbau des Gateway Dienstes, welcher der Clean-Architecture nachempfunden ist, beschrieben.

Die Struktur kann in folgendem Listing als Tree 5.1 betrachtet werden.

```
    ├── backup_rules
    ├── gateway
    │   ├── domain
    │   ├── repository
    │   ├── rest
    │   ├── serializers
    │   ├── shared
    │   └── use_cases
    ├── requirements
    ├── tests
    │   ├── domain
    │   ├── repository
    │   ├── rest
    │   ├── serializers
    │   ├── shared
    │   └── use_cases
```

```
└── venv
    ├── bin
    ├── include
    ├── lib
    ├── lib64 -> lib
    └── share
```

Listing 5.1: Gateway-Struktur

Wie schon im Kapitel 3.10 beschrieben, besteht die Python Clean-Architecture aus mehreren Schichten, welche von einander isoliert und immer gegen innen gerichtet sind.

Die *Entities* befinden sich im **Domain-Ordner**, die *Applikation Business-Rules* sind im **Use_Case Ordner**, der *Interface-Adapter* mit der View und dem Controller im **Serializer-Ordner**, die *Frameworks- und Drivers* mit der Datenbankanbindung im **Repository-Ordner** sowie dem Web-Framework Flask im **rest-Ordner**. Zum Serializer-Ordner ist hierbei anzumerken, dass es naheliegend ist, dass wir nur die Darstellung der Daten und Weiterleitung dieser Daten sowie die Umwandlung dieser im JSON Format möchten und wir dies für unseren Zweck als ausreichend erachtet haben.

Daneben gibt es noch zu jedem dieser Ordner einen gleichnamigen Ordner im **Test-Ordner**, wo sich alle *pytest*-Tests befinden. Der **Backup_Rules-Ordner** beherbergt die persistierten Regeln im JSON-String-Format und im **venv-Ordner** befindet sich die lokal erzeugte virtuelle Python-Umgebung.

REST API

Auf die REST API wurde im Abschnitt 4.2.4 Analyse schon vertieft eingegangen und wir haben uns an diese gehalten. Ein API Auszug befindet sich im Anhang unter 7.8, welcher durch Swagger erstellt und mittels eines geeigneten Tools in ein lesbares Format transformiert wurde.

Installation

Eine genaue Installationsanleitung kann im Anhang 7.6 entnommen werden.

5.4.2 ConMan

Der Connection-Manager oder Message-Manager (im nachfolgenden kurz *ConMan* genannt) ist das Herzstück zu den Sensor- und Aktuatordaten. Er übernimmt die Kontrolle über den MQTT Broker, bindet sich auf die verschiedenen Channels wie *Sensors* oder *Actuators* Channel und holt sich, sobald Daten auf einem der Channel vorhanden sind, diese Daten und leitet sie weiter an Redis zur Speicherung. Es gibt zwei unabhängige Threads, der eine ist für die Sensoren zuständig und der andere für die Aktuatoren.

5.4.3 Analyser

Der Analyser ist dafür verantwortlich, die Regeln mit den vorhanden Sensordaten zu prüfen. Dazu greift er jeweils für jeden Sensor den letzten Wert aus Redis ab und vergleicht die Regeln mit den Werten auf Übereinstimmung. Trifft dabei eine Regel, also eine Bedingung einer Regel (Condition) zu, wird eine Handlung (Event/Action) ausgeführt. Dies kann eine Benachrichtigung eines Aktuators oder eine einfache Konsolenausgabe sein.

Versuch 1: Venmo Business-Rules

Zu Beginn des Projektes wurde mit einer einfachen Business-Rules von Venmo [25] gearbeitet, um einen Prototyp und den ersten Architekturprotyp für den Durchstich zu erreichen. Im Verlaufe des Projekts wurde anhand einer Nutzwertanalyse verschiedene weitere Möglichkeiten evaluiert und in Betracht gezogen. Die Nutzwertanalyse hat jedoch ganz knapp gezeigt, dass für unseren Fall die Venom Business-Rules ausreichend sind.

In einem späteren Zeitpunkt des Projekts und im fortlaufendem Stadium sind wir jedoch auf einige interessante Probleme und Schwierigkeiten gestossen, die wir in drei Kernprobleme aufteilen können. Grund dafür war die Kundenanforderung, mehrere Sensorwerte miteinander zu vergleichen. Dies stellte sich mit der jetzigen Venmo Business-Rules als schwierig oder gar unmöglich heraus. Die erwähnten Kernprobleme sowie ein Lösungsansatz werden folgend beschrieben und erläutert.

Kernprobleme und Erweiterung

Bei der verwendeten Venmo Business-Rule Library gibt es drei Kernprobleme, die es zu lösen gilt.

1. Die Daten sollen unabhängig und ohne Verlust geholt und in ein passendes Objekt abgefüllt werden.

2. Um mehrere Sensoren gleichzeitig auszuwerten, braucht es das oben erwähnte passende Objekt, in dem ein Dictionary von Message Objekten steckt und durchsucht werden kann (Mittels Sensor-ID ein Dictionary durchsuchen (key = ID, value = Sensordaten).

3. Man soll auf einzelne Objekte referenzieren können, beispielsweise mit Parametern (Business-Rule erweitern mit Variablen, bei denen Parameter angegeben werden können oder ein ID Feld mitgegeben werden kann).

Um die genannten Kernprobleme zu lösen und die Anforderungen zu erfüllen, muss einerseits die Business-Rule erweitert und anderseits zusätzliche Logik implementiert werden.

Glücklicherweise ist dieses Bedürfnis, Parameter in Variablen zu setzten, schon früher aufgekommen. Ein Fork der ursprünglichen Venmo Business-Rules [25] macht dies bereits

möglich und nennt sich für uns namentlich Yoyowallet Business-Rules [54].

Nach der Evaluation und dem Testing des neuen Forks wurde die Venmo Business-Rule mit der geforkten, weiterentwickelten yoyowallet Business-Rules Version ausgetauscht und die Regeln angepasst. Dazu wurde ein Python Virtual Environment (venv) erzeugt, die geforkte Version klont und in das virtuelle Environment installiert.

Versuch 2: Yoyowallet Business-Rules

Bei der neuen yoyowallet Business-Rules [54] ist es nun möglich für definierte Variablen, Parameter mitzugeben um innerhalb von Bedingungen (Conditions) auf einzelne spezifische Sensoren mittels Sensor-ID zu referenzieren. Dies ermöglicht uns nun Regeln zu erstellen, in denen verschiedene Sensorwerte miteinander verglichen werden.

Eine funktionierende Beispielregel, in der mehrere Sensorwerte miteinander verglichen werden, ist in nachfolgendem Listing 5.2 beschrieben.

```
[{
    "conditions": {
        "all": [{
            "name": "current_value",
            "params": {
                "sensorid": "1132fac0-633a-11e9-9a30-23bf466785d0"
            },
            "value": 150,
            "operator": "greater_than"
        }, {
            "name": "current_value",
            "params": {
                "sensorid": "1057c680-633a-11e9-8abf-a7cb277ba226"
            },
            "value": 50,
            "operator": "less_than"
        }]
    },
    "actions": [{
        "name": "send_to_actuator",
        "params": {
            "message": "Multi-Sensor-Korrelation-Test",
            "actuator_id": "b2892a16-633a-11e9-b685-37d81680a95d"
        }
    }]
}]
```

Listing 5.2: Multi-Sensor-Regel

Weitere Beispiele von Regeln sind im Abschnitt 4.2.6 erwähnt, in welchem Regeln mit den User Stories in Verbindung gebracht werden.

Ein nettes Feature, welches die Erstellung von Regeln stark vereinfacht ist ein simples grafisches User-Interface [55], welches auf dem API Endpunkt */ui* aufrufbar ist. Nachfolgende Grafik 5.6 zeigt dessen Darstellung im Browser.

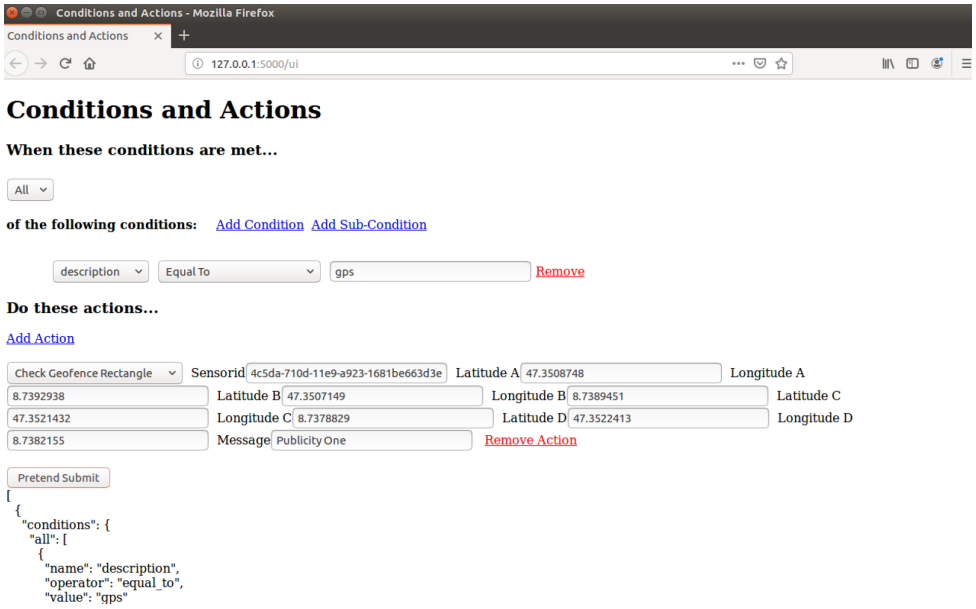


Abbildung 5.6: UI zur Erstellung von Regeln

Wie die nötigen Variablen zum Schreiben von Bedingungen sowie Aktionen zum Ausführen von Ereignissen beim Eintreffen von Bedingungen geschrieben werden, wird im Anhang unter 7.7 anhand einer Anleitung mit Beispielen verständlich erklärt. Grundsätzlich braucht es ein Set von Variablen, mit Hilfe dieser Bedingungen formuliert werden können sowie ein Set von Aktionen, welche im Prinzip Methoden definieren, die ein Ereignis beschreibt und auslöst, wenn eine der formulierten Bedingungen in einer Regel zutreffen.

5.5 Simulatoren

Zu Demonstrationszwecken, zum Testen unserer Applikation und Logik haben wir uns sehr früh für Simulationen entschieden. Ausserdem war die Hardware selbst erst sehr spät vorhanden und wir mussten auf Alternativen ausweichen.

5.5.1 Sensoren

Die Sensoren liefern Werte, welche für uns zur Auswertung und zur Erstellung von Regeln nützlich sind. Dies können Werte über den Motor, die Geschwindigkeit, den Ort oder der Umgebung sein.

Im nachfolgenden Abschnitt wird der SensorMessage Aufbau beschrieben, welcher für unser Projekt zur Einfachheit verwendet wurde. Dieser Aufbau ist keinesfalls konzeptionell durchdacht und sollte, wenn möglich, noch angepasst werden. Für uns war diese Struktur aber für jegliche Arten von Use Cases ausreichend. Aus diesem Grund haben wir uns dazu entschieden, keine Zeit auf mögliche verbesserte und optimierte Strukturen zu verlieren.

SensorMessage Aufbau

Zur Vereinfachung und zum Testen haben wir uns für die Struktur des gefundenen Simulator-Scripts entschieden. Es bietet jegliche Daten an, die zu einer Versuchsumsetzung vorhanden sind, um einen Proof-of-Concept zu demonstrieren.

Ebenfalls zur Vereinfachung ist *lat* und *lng* bei allen Sensoren dabei, obwohl dies nur eine signifikante Wichtigkeit bei den GPS Sensoren hat. Grund dafür ist, dass wir damit nicht mehrere Domain Modelle für verschiedene Sensoren führen müssen und dadurch den Aufwand minimieren können.

In der Realität wäre es natürlich denkbar, verschiedene Message Typen für verschiedene Sensor Typen zu halten. Auf das wurde aus bereits erwähnten Gründen verzichtet. Nachfolgend ist eine SensorMessage mit Beispieldaten, wie sie als Payload dem MQTT-Broker übergeben wird, dargestellt.

```
{
    "sensor_id": "1132fac0-633a-11e9-9a30-23bf466785d0",
    "value": 46,
    "lat": 0,
    "lng": 0,
    "unit": "C",
    "typ": "temperature",
    "description": "Motor"
}
```

Eine weitere SensorMessage mit Beispieldaten eines GPS Sensors.

```
{
    "sensor_id": "1057c680-633a-11e9-8abf-a7cb277ba226",
    "value": 0,
    "lat": 8.82123,
    "lng": 47.23907,
    "unit": "",
    "typ": "gps",
    "description": "GPS-Sensor1"
}
```

5.5.2 Aktuatoren

Ein Aktuator ist ein Gerät, welches im Gegensatz zu Sensoren keine Daten im herkömmlichen Sinne liefert, sondern Daten entgegennimmt und darstellt. Dies kann beispielsweise ein Monitor, ein Lautsprecher, ein Touchscreen oder eine Nachricht an ein Handy mittels SMS sein.

5.6 Geofencing-Konzept

Um Tests durchführen zu können, nehmen wir eine Strecke auf der sich ein Tunnel befindet, wie im Bild 5.7 ersichtlich. Anschliessend halten wir ein paar mögliche Koordinaten fest. Natürlich ergibt diese Strecke keinen Sinn, da man in einem Tunnel keinen GPS-Empfang hat. Trotzdem genügen diese Daten für ein erstes Experiment mit unserem GPS-Simulator.

Abbildung 5.7: Strecke für erste Tests

5.6.1 Geofence als Bedingung in einer Regel

Für das Geofencing haben wir zwei verschiedene Varianten ausprobiert. Die erste Möglichkeit bestand darin, den Geofence-Bereich als Regel zu überprüfen. Dieser Ansatz funktioniert aber nur mit einem Viereck. Man gibt die vier verschiedenen Punkte (Latitude und Longitude) vom gewünschten Bereich als einzelne Konditionen an und erstellt eine Aktion (zum Beispiel: send to actuator). Wenn sich also das GPS-Signal innerhalb dieses Bereichs befindet, wird die gewünschte Aktion ausgeführt.

```
[{
  "conditions":{
    "all":[{
      "name":"sensor_id",
      "operator":"equal_to",
      "value":"7594c5da-710d-11e9-a923-1681be663d3e",
      "params":{
        "sensorid":"7594c5da-710d-11e9-a923-1681be663d3e"
      }
    },
    {
      "name":"longitude",
      "operator":"greater_than_or_equal_to",
      "value":8.7378829,
      "params":{
        "sensorid":"7594c5da-710d-11e9-a923-1681be663d3e"
      }
    },
    {
      "name":"longitude",
      "operator":"less_than_or_equal_to",
      "value":8.7392938,
      "params":{
        "sensorid":"7594c5da-710d-11e9-a923-1681be663d3e"
      }
    },
    {
      "name":"latitude",
      "operator":"greater_than_or_equal_to",
      "value":47.3507149,
      "params":{
        "sensorid":"7594c5da-710d-11e9-a923-1681be663d3e"
      }
    },
    {
      "name":"latitude",
      "operator":"less_than_or_equal_to",
      "value":47.3522413,
      "params":{
        "sensorid":"7594c5da-710d-11e9-a923-1681be663d3e"
      }
```

```
    }]
  },
  "actions":[{
    "name":"send_to_actuator",
    "params":{
      "actuator_id":"b2892a16-633a-11e9-b685-37d81680a95d"
    }
  }]
}]
```

Listing 5.3: Beispiel einer Geofence Regel als JSON

5.6.2 Geofence als Aktion in einer Regel

Bei der zweiten Möglichkeit haben wir den Bereich in einer Funktion (Aktion) abgebildet. Wir haben also eine Funktion (check geofence circle) geschrieben, die den Bereich anhand der gegebenen Koordinaten und des Radius' berechnet. Die Regel besteht dann aus einer einzigen Bedingung, welche den aktuellen Sensor überprüft (if Sensor == GPS) und daraufhin die von uns erstellte Funktion aufruft. Wenn man die Funktion als Aktion auswählt, muss man noch diverse Parameter (Latitude, Longitude, Radius in km, etc.) mitgeben, damit sie überhaupt funktioniert.

Regel 1 Für die erste Regel haben wir die Koordinaten vom HSR Gebäude 1 ausfindig gemacht. Wir möchten, dass eine Nachricht an den Aktuator gesendet wird, wenn sich das GPS-Signal innerhalb von 10km - vom HSR Gebäude 1 - befindet.

```
[{
  "conditions": {
    "all": [{
      "name": "description",
      "operator": "equal_to",
      "value": "gps",
      "params": {
        "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e"
      }
    }]
  },
  "actions": [{
    "name": "check_geofence_circle",
    "params": {
      "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e",
      "radius": 10,
      "center_latitude": 47.223498264901025,
      "center_longitude": 8.817343711853027,
      "message": "Geofence circle at HSR building 1"
    }
  }]
```

```
}]
```

Listing 5.4: Geofence Regel in einem Umkreis

Regel 2 Als zweite Regel überprüfen wir die Koordinaten der ersten Möglichkeit als Aktion.

```
[{
  "conditions": {
    "all": [{
      "name": "description",
      "operator": "equal_to",
      "value": "gps",
      "params": {
        "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e"
      }
    }]
  },
  "actions": [{
    "name": "check_geofence_rectangle",
    "params": {
      "sensorid": "7594c5da-710d-11e9-a923-1681be663d3e",
      "latitude_a": 47.3508748,
      "longitude_a": 8.7392938,
      "latitude_b": 47.3507149,
      "longitude_b": 8.7389451,
      "latitude_c": 47.3521432,
      "longitude_c": 8.7378829,
      "latitude_d": 47.3522413,
      "longitude_d": 8.7382155,
      "message": "Geofence square in tunnel"
    }
  }]
}]
```

Listing 5.5: Geofence Regel in einem Rechteck

5.6.3 Aktionen ausführen

Momentan ist es so, dass die Aktion, die man in der Regel definiert hat, jedes Mal ausgeführt wird, wenn man sich in dem vorgegebenen Bereich befindet. Das ist nicht angenehm, weil man dann beispielsweise ständig die gleiche Benachrichtigung auf dem Smartphone oder der Konsole hat. Um das Problem zu umgehen haben wir uns folgendes ausgedacht.

Man könnte eine Variable (ein Flag) in die Datenbank (Redis) einpflegen und die standardmässig auf **False** setzen. Sobald man eine Geofence-Regel / Aktion erstellt und man sich das erste Mal in diesem Bereich befindet, wird diese Variable in der Datenbank auf **True** gesetzt. Bei der Überprüfung des Bereichs wird diese Variable natürlich auch beachtet.

Wenn sich also das GPS-Signal in diesem Bereich befindet und die Variable **True** ist, wird ein globaler Counter gestartet. Wenn der Counter 1 ist, wird die Aktion gestartet – weil sich das Signal zum ersten Mal in diesem Bereich befindet. Die restlichen Male geschieht nichts. Sobald sich das Signal ausserhalb vom Bereich befindet, werden der Counter und die Variable in der Datenbank wieder zurückgesetzt. So verhindern wir, dass bei jeder Änderung die gleiche Aktion ausgeführt wird.

5.6.4 Berechnungen

Damit das Geofence-Konzept überhaupt funktioniert, muss der gewünschte Bereich anhand der gegebenen Koordinaten genaustens berechnet werden. Da die Koordinaten immer als Gleitkommazahl gegeben ist, muss bei der Bedingung ausserordentlich aufgepasst werden. Solche Zahlen kann man nicht einfach miteinander vergleichen, da man noch eine Toleranz-Variable einrechnen muss.

Bei diversen Recherchen haben wir uns auch darüber informiert, wie viele Nachkommastellen es braucht, damit man eine Position auf ein Meter genau angeben kann. Diesbezüglich geht es genau um sieben Nachkommastellen. Wir runden also auf genau acht Nachkommastellen mathematisch auf oder ab, damit wir die gewünschten Positionen genaustens miteinander vergleichen können.

Check Geofence Rectangle

Die Berechnung der Rechteck-Form ist etwas spezieller und funktioniert wie folgt. Wenn man die linke Grafik betrachtet, sieht man sechs verschiedene Punkte beziehungsweise Koordinaten, die in der Karte eingetragen sind. Wir möchten beispielsweise überprüfen, ob sich der Punkt X im abgesteckten Bereich, wie im rechten Bild 5.9 ersichtlich, befindet. Am Anfang wird diese Fläche ABCD berechnet und zwischengespeichert.

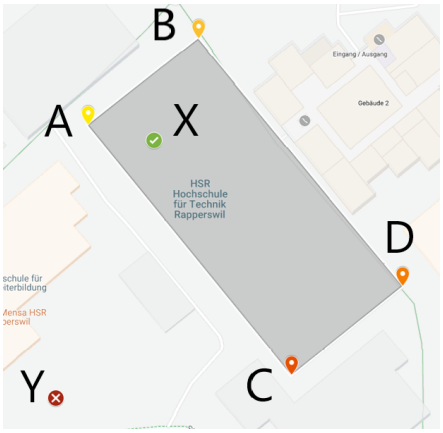


Abbildung 5.8: Koordinaten

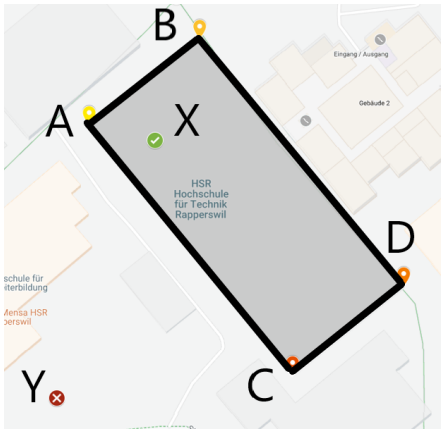


Abbildung 5.9: Fläche ABCD

Danach Bildet man Dreiecksflächen vom Punkt X zu den einzelnen Punkten und berechnet diese, wie in den untenstehenden Bildern 5.10 5.11 5.12 sowie 5.13 demonstriert.

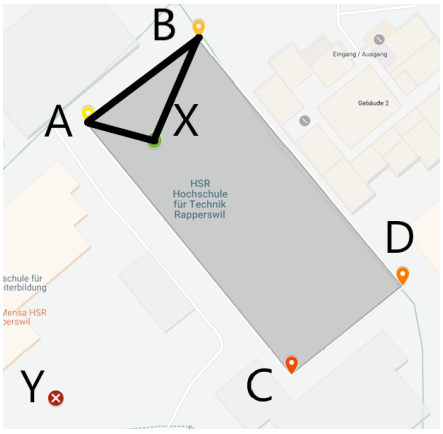


Abbildung 5.10: Fläche XAB

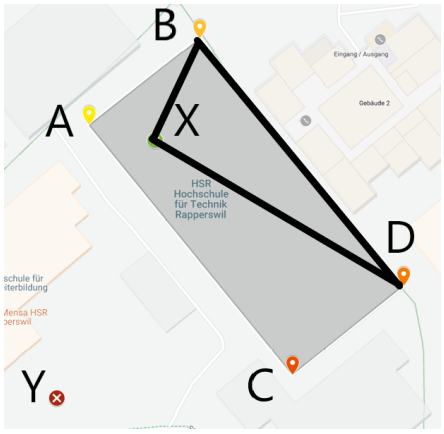


Abbildung 5.11: Fläche XBD

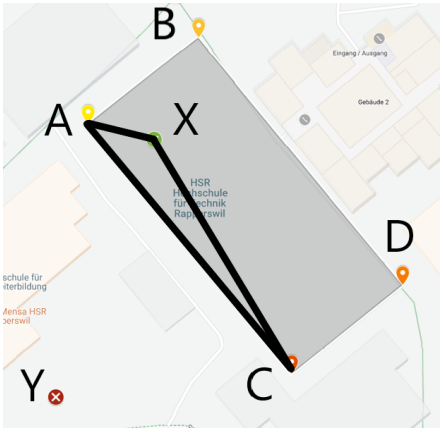


Abbildung 5.12: Fläche XAC

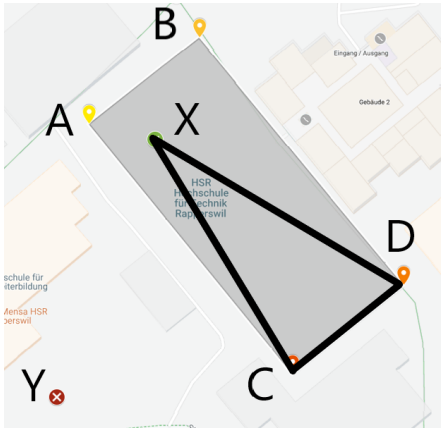


Abbildung 5.13: Fläche XCD

Sobald man alle diese Flächen hat, addiert man sie zusammen. Danach wird überprüft, ob die Fläche ABCD der Summe der einzelnen Dreiecksflächen entspricht. Wenn diese Flächen gleich sind, dann befindet sich der Punkt X genau in diesem abgesteckten Bereich.

Wenn wir allerdings den Punkt Y betrachten, sehen wir, dass er deutlich ausserhalb vom Bereich steht. Das Prinzip bleibt gleich wie oben, nur ändern sich hier natürlich die Flächen, wie in den folgenden Bildern 5.14 5.15 5.16 und 5.17 aufgezeigt.

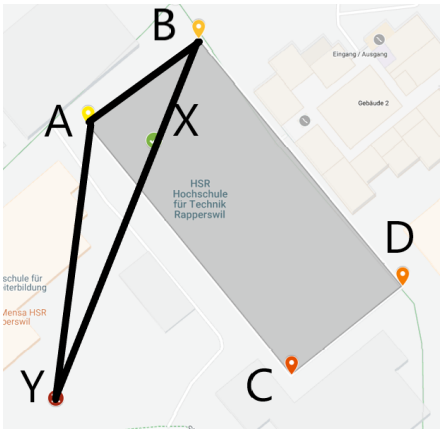


Abbildung 5.14: Fläche YAB

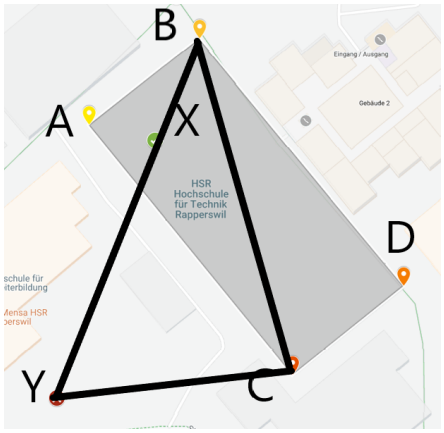


Abbildung 5.15: Fläche YBC

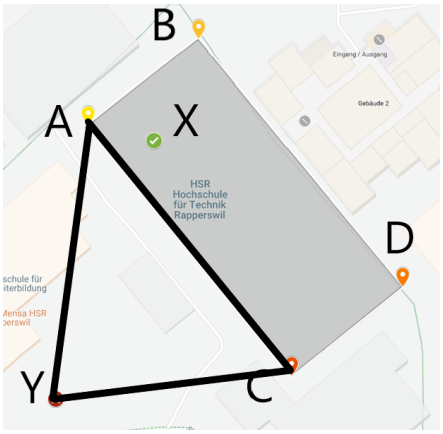


Abbildung 5.16: Fläche YAC

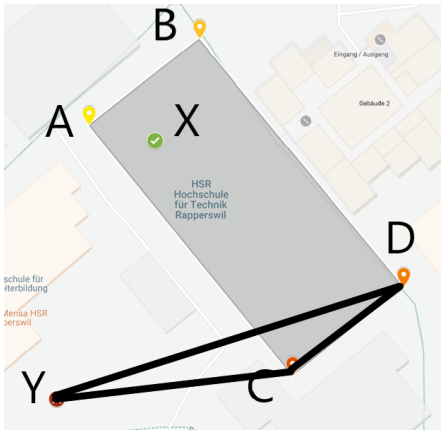


Abbildung 5.17: Fläche YCD

Wenn man nun diese Flächen miteinander vergleicht, merkt man ziemlich schnell, dass die Summe der einzelnen Dreiecksflächen deutlich grösser ist als die Fläche ABCD. Somit befindet sich der Punkt Y nicht im gewünschten Bereich. Hätte man also eine Regel mit genau dieser Bedingung erstellt, so würde bei diesem Beispiel keine Aktion ausgeführt werden.

Check Geofence Circle

Um einen Umkreis zu bilden, wird die gewünschte, zentrale Position (Latitude, Longitude) benötigt. Ab diesem Punkt wird dann der gegebene Radius geometrisch berechnet. Ausserdem werden die Koordinaten in Kilometern und umgekehrt berechnet, damit man eine Relation hat.

5.7 Installation

Zur allgemeinen Verwendung der Rule-Engine sind wir auf Redis sowie MQTT angewiesen. Die Installationsanleitung und Verwendung wird wiederum im Anhang unter 7.6 im Detail beschrieben.

6 Zusammenfassung und Ausblick

6.1 Zielerreichung

6.1.1 Use Cases

Nr	Use Case	Status
UC1	Daten senden	erfüllt
UC2	Regel CRUD	teilweise
UC3	Aktion beschreiben	erfüllt
UC4	Kondition beschreiben	erfüllt
UC5	Benachrichtigung senden	erfüllt
UC6	RegelSet abrufen	erfüllt
UC7	Gerätelisten abrufen	erfüllt
UC8	Daten abrufen	erfüllt
UC9	Kondition erstellen	erfüllt
UC10	Daten verteilen	erfüllt
UC11	Aktion erstellen	erfüllt
UC12	UI Regelbaukasten abrufen	erfüllt

UC2 Regel CRUD

Dieser Use Case ist grundsätzlich vollständig umgesetzt mit Ausnahme des Update-Prozesses einer Regel. Es ist zur Zeit nicht möglich einzelne Werte oder Parameter einer bereits gespeicherten Regel anzupassen.

6.1.2 User Stories

Nr	User Story	Status
US1	Regeln setzen und ausgeben	erfüllt
US2	Regeln setzen und abfragen	erfüllt
US3	Werte von Regeln anpassen	teilweise
US4	Positionsbestimmung GPS und Odometer	teilweise
US5	Einsatz Geofencing	erfüllt
US6	Temperaturschwellwert und Alarm	erfüllt

US7	Sensordaten filtern	teilweise
US8	Länderwechsel erkennen	offen
US9	Routenabweichung erkennen	offen
US10	Umleitungen anzeigen	offen
US11	Haltestellen ansagen	erfüllt
US12	Echtzeitdaten mittels App abfragen	offen
US13	Positions für rechnergesteuertes Leitsystem	offen
US14	Anbieten geobasierter Werbeinhalte	erfüllt
US15	Eco-Drive Daten	erfüllt
US16	Motorüberwachung	erfüllt
US17	Füllstandsanzeige	erfüllt
US18	Kontrolle Beschleunigungssensor	offen
US19	Pünktlichkeitsmessung	offen
US20	Verbrauchsgerechte Fahrtenaufteilung	offen
US21	Fahrgastzählungen	offen
US22	Datenkorrelation	erfüllt

US3 Werte von Regeln anpassen

Es ist nicht möglich bereits gespeicherte Regeln direkt anzupassen. Es besteht aber die Möglichkeit die Regel zu löschen, anzupassen und neu zu speichern.

US4 Positionsbestimmung GPS und Odometer

Es ist möglich die Position auszulesen und auszugeben. Theoretisch denkbar wäre auch eine Aktion die den Odometerzähler startet, da dies jedoch Tasks voraussetzt, welche im Hintergrund ausgeführt werden und wir diese noch nicht implementiert haben, ist diese User Story als nur teilweise umgesetzt zu betrachten.

US7 Sensordaten filtern

Es gibt einzelne Endpunkte um den Stream an Sensordaten gefiltert abzufragen. Beispielsweise ist es möglich nach Typ, Wert oder Beschreibung zu filtern. Es besteht jedoch noch keine Möglichkeit, Regeln zum Setzen von Filtern von Daten zu erstellen.

6.1.3 Nicht-funktionalle Anforderungen

Nr	**NFR**	**Status**
NFR1	Rahmenbedingungen	erfüllt
NFR2	Verbrauchsverhalten	erfüllt
NFR3	Skalierbarkeit	erfüllt
NFR4	Wiederherstellbarkeit	erfüllt
NFR5	Fehlertoleranz	erfüllt

NFR6	Überprüfbarkeit	erfüllt
NFR7	Erweiterbarkeit	erfüllt
NFR8	Verständlichkeit	erfüllt

6.1.4 Testprotokoll

Das Testprotokoll mit den einzelnen Testdurchführungen, Voraussetzungen und Bemerkungen kann im Anhang unter 7.9 nachgeschlagen werden.

6.2 Zusammenfassung

Für das Konzept einer Rule Engine haben wir nun insgesamt 16 Wochen benötigt. Während dieser ganzen Zeit sind 12 Use Cases, 22 User Stories und 8 nicht funktionale Anforderungen entstanden. Von den 12 Use Cases konnten wir alle bis auf einen (Use Case Regel CRUD) erledigen. Dieser offene Use Case beinhaltete allerdings mehr als eine Aktion, die zu realisieren war. Aus diesem Grund implementierten wir nur die, die auch für die weiteren Use Cases und User Stories von Bedeutung waren. Die User Stories konnten jedoch nicht alle fertiggestellt werden. Von insgesamt 22 wurde genau die Hälfte vollständig realisiert. Die anderen wurden aus diversen Gründen (im oberen Abschnitt genauer erklärt) ignoriert. Sehr erfreulich waren für uns die nicht funktionalen Anforderungen, denn diese konnten wir alle erfolgreich erfüllen.

Zu Beginn verwendeten wir einen Architektur-Ansatz, der uns aber im Verlauf des Projekts nicht wirklich gefiel. Die Struktur war nicht wie gewünscht und der Code war auch nicht so gegliedert, wie er sein sollte. Daraufhin haben wir einen etwas anderen Ansatz (Clean Architecture) ausprobiert. Da wir aber nicht zu 100 % sicher waren, ob diese Herangehensweise mit unserer bisherigen Arbeit kompatibel ist, haben wir diesen Schritt parallel ausgeführt. Die Arbeit wurde so aufgeteilt, dass eine Person an der neuen Architektur und die andere am bisherigen Stand arbeitete. So konnten wir glücklicherweise die neue Architektur mit unserem aktuellen Stand zusammenfügen und so gemeinsam daran weiterentwickeln. Wenn wir das Projekt neu starten könnten, würden wir aber von Anfang eine Python Clean Architecture für die saubere Trennung der einzelnen Komponenten verwenden.

Für die Fehlerbehandlung und allgemeine Überprüfungen der Services, müssten allerdings noch Validierungen und Domain-Modelle entwickelt werden. Diese waren jedoch nicht in unserem Projektumfang eingeplant. Die Datenstruktur der Sensor-Messages könnte unseres Erachtens verbessert werden oder man erstellt je nach Typ ein eigenes Domain-Modell (wie zum Beispiel für GSP). Für die Umsetzung des State-Managements war die Zeit etwas knapp - dies war nie von Anfang an eingeplant und entstand erst spät in der Construction-Phase.

Sollte der Connection Manager (ConMan) weiterentwickelt werden, so muss man noch

dessen Daten korrekt validieren. Mit unserem aktuellen Stand der Rule Engine werden diese Daten nicht auf die Korrektheit überprüft. Da vom Kunden kein User Interface (UI) verlangt wurde, haben wir auch praktisch keine Zeit dafür eingeplant. Trotzdem existiert eine UI, mit der man einfach Regeln erstellen und verwenden kann. Die UI ist jedoch nicht auf die neuen yoyowallet Business Rules angepasst worden - dies ist bei der Weiterentwicklung zu beachten.

6.3 Ausblick

Der intelligente Gateway ist gewiss noch nicht fertig. Wir haben einige Use Cases, die teilweise oder noch gar nicht erfüllt sind, wie im oberen Abschnitt ersichtlich ist. Zudem gibt es Konzepte, die ebenfalls noch nicht umgesetzt worden sind

Dazu gehört zum Beispiel das Redis-Repository. Die Regeln werden nicht auf die Inkonsistenz geprüft beziehungsweise die Regeln sind nicht einheitlich als JSON- oder String-Objekt abgelegt. Ausserdem fehlt noch ein **Filter** bei der Datensammlung. Damit ist gemeint, dass man beispielsweise einstellen kann, ob man überhaupt GPS-Daten (oder andere Sensordaten) oder in welchen Zeitabständen man diese Daten sammeln möchte. Aktuell werden alle Daten (egal von welchem Sensor) zwischengespeichert.

Des Weiteren bräuchte es noch ein **State-Manangement** (oben schon erwähnt) für Regeln, um zu wissen ob eine Regel gerade getriggert wurde oder nicht. Beispielsweise könnte man eine Geofence-Regel mit einem Umkreis von 10 km erstellen. Befindet man sich in diesem Bereich, so erhält der Benutzer eine Benachrichtigung auf seinem Smartphone. Ohne dieses State-Management (aktueller Stand) würde der Benutzer bei jeder Positionsänderung innerhalb des Bereichs eine Meldung erhalten. Das wäre unerfreulich und möchte man natürlich verhindern. Das State-Management sollte daher die erste Aktion ausführen und solange sich der Benutzer im entsprechenden Bereich befindet, soll nichts geschehen. Wie dieses State-Management genau funktionieren soll, ist im Abschnitt Implementation Geofence-Konzept 5.6 beschrieben.

Momentan gibt es in der Rule Engine nur einzelne Aktionen ohne Background Tasks. Die Idee ist, dass man beispielsweise eine Regel erstellen könnte, mit dem der Odometer im Hintergrund gestartet wird, wenn kein GPS-Signal vorhanden ist (wichtig für Haltestellen-Ansagen oder ähnliches). Dies könnte mit Celery umgesetzt werden.

Zu guter Letzt möchten wir noch einen offenen Punkt festhalten - den Filter für den Connection Manager. Um einen oder mehrere Filter für den Connection Manager zu setzen, könnte für jeden angeschlossenen Sensor ein Thread auf einen Channel gepusht werden, welcher dann später vom Connection Manager Sensor-Thread abgrufen werden kann. Die-

se Threads könnten beispielsweise mit der SensorID als ThreadID / Name ausgestattet werden, um darauf mittels einer Regel-Aktion über die Redis-Konfigurationswerte (Pub / Sub) zuzugreifen. Die Thread-Klasse würde mit grundlegenden Parametern voreingestellt werden, welche man mit dem soeben beschriebenen Pattern anpassen könnte. Wichtig wäre allerdings, dass man nur die neuen Daten zwischenspeichert. Es müssen nicht alle Daten festgehalten werden, wenn gegebenenfalls der Sensor X Mal die gleichen Daten sendet. Somit würde die Datenmenge im Redis stark minimiert werden und die Performance könnte gesteigert werden.

7 Verzeichnisse

7.1 Glossar und Abkürzungen

ACM

Association for Computing Machinery [56]

bsd-lizenz

Das BSD-Lizenzproblem [15]

Can-BUS

Can-BUS ist ein BUS Standard für Fahrzeuge [42]

choosealicense

Choose an open source license [57]

CleanArchitecture

The Clean Architecture Style [41] [40]

CoAP - Constrained Application Protocol

Das Constrained Application Protocol (CoAP) ist ein spezialisiertes Webübertragungsprotokoll zur Verwendung mit eingeschränkten Knoten und eingeschränkten Netzwerken im Internet der Dinge. Das Protokoll ist für Machine-to-Machine-Anwendungen (M2M) wie Smart Energy und Gebäudeautomation konzipiert. [30]

Confluence

Kolaborationssoftware für Teams [58]

Docker

Enterprise Container Platform zum sicheren erstellen von Applikationen und dessen Verteilung [1]

Gitlab

Webanwendung zur Versionsverwaltung für Softwareprojekte auf Basis von Git [59]

IEEE

The world's largest technical professional organization for the advancement of technology[60]

Jira

Software zur Vorgangs- und Projektverfolgung für agile Softwareentwicklungsprojekte [61]

LaTeX

LaTeX Typesetting System [62]

MQTT - Message Queuing Telemetry Transport

MQTT ist ein offenes Nachrichtenprotokoll für Machine-to-Machine-Kommunikation (M2M) [29]

OpenFog Consortium

OpenFog Konsortium mit dem Ziel zur Standardisierung und Förderung des Fog-Computing [6]

Pytest

Python Test Modul [53]

RClone

Eine rsync Lösung für Cloud-Speicher [63]

REST - Representational State Transfer

Programmierschnittstelle, die sich an den Paradigmen und Verhalten des World Wide Web (WWW) orientiert und einen Ansatz für die Kommunikation zwischen Client und Server in Netzwerken beschreibt [31]

Swagger

Swagger API Beschreibung [64]

VSCode

VSCode - Visual Studio Code Entwicklungsumgebung [44]

7.2 Abbildungsverzeichnis

7.3 Tabellenverzeichnis

7.4 Literatur

[1] *Docker CE*, [Online; aufgerufen am 2. Mai 2019]. Adresse: `https://www.docker.com/`.

[2] *Docker-Machine*, [Online; aufgerufen am 2. Mai 2019]. Adresse: `https://docs.docker.com/machine/`.

[3] *Docker-Swarm*, [Online; aufgerufen am 2. Mai 2019]. Adresse: `https://docs.docker.com/engine/swarm/`.

[4] *Docker-Compose*, [Online; aufgerufen am 2. Mai 2019]. Adresse: `https://docs.docker.com/compose/`.

[5] *LXC - Linux Containers*, [Online; aufgerufen am 11. Juni 2019]. Adresse: `https://linuxcontainers.org/`.

[6] *OpenFog - Building Open Architecture for Fog Computing*, [Online; aufgerufen am 2. März 2019]. Adresse: `https://www.openfogconsortium.org/`.

[7] B. Kevin Bachmann, *Design and Implementation of a Fog Computing Framework*, A-1040 Wien Karlsplatz 13, 2017.

[8] *TU-Wien - Technische Universität Wien*, [Online; aufgerufen am 2. März 2019]. Adresse: `https://www.tuwien.ac.at/`.

[9] *FogFrame*, [Online; aufgerufen am 2. März 2019]. Adresse: `https://github.com/keyban/fogframe`.

[10] *Apache License 2.0*, [Online; aufgerufen am 19. Februar 2019]. Adresse: `https://www.apache.org/licenses/LICENSE-2.0`.

[11] *FogLAMP*, [Online; aufgerufen am 2. März 2019]. Adresse: `https://github.com/foglamp/FogLAMP`.

[12] *FogLAMP Dokumentation*, [Online; aufgerufen am 2. März 2019]. Adresse: `https://foglamp.readthedocs.io`.

[13] A. Azzarà, D. Alessandrelli, S. Bocchino, M. Petracca und P. Pagano, „PyoT, a macro-programming framework for the Internet of Things“, in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, Juni 2014, S. 96–103. DOI: `10.1109/SIES.2014.6871193`.

[14] *PyOT - Webbasierte Makroprogrammierungsoberfläche*, [Online; aufgerufen am 2. März 2019]. Adresse: `https://github.com/tecip-nes/pyot`.

[15] *GNU - Das BSD-Lizenzproblem*, [Online; aufgerufen am 19. Februar 2019]. Adresse: `https://www.gnu.org/licenses/bsd.de.html`.

[16] E. Yigitoglu, M. Mohamed, L. Liu und H. Ludwig, „Foggy: A Framework for Continuous Automated IoT Application Deployment in Fog Computing“, in *2017 IEEE International Conference on AI Mobile Services (AIMS)*, Juni 2017, S. 38–45. DOI: 10.1109/AIMS.2017.14.

[17] P. Tsai, H. Hong, A. Cheng und C. Hsu, „Distributed analytics in fog computing platforms using tensorflow and kubernetes“, in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Sep. 2017, S. 145–150. DOI: 10.1109/APNOMS.2017.8094194.

[18] *Microservices Martin Fowler*, [Online; aufgerufen am 11. Juni 2019]. Adresse: https://martinfowler.com/articles/microservices.html.

[19] *Flask - Web Microservice Framework*, [Online; aufgerufen am 11. Juni 2019]. Adresse: http://flask.pocoo.org/.

[20] *Werkzeug - WSGI Web Application Library*, [Online; aufgerufen am 11. Juni 2019]. Adresse: http://werkzeug.pocoo.org/.

[21] *Jinja2 - Template Engine*, [Online; aufgerufen am 11. Juni 2019]. Adresse: http://jinja.pocoo.org/.

[22] *Nameko - Microservice-Framework*, [Online; aufgerufen am 11. Juni 2019]. Adresse: https://nameko.readthedocs.io/en/stable/.

[23] *Aiohttp - Asynchrones HTTP Client-Server-Framework*, [Online; aufgerufen am 11. Juni 2019]. Adresse: https://github.com/aio-libs/aiohttp.

[24] *Aiohttp Readthedocs*, [Online; aufgerufen am 11. Juni 2019]. Adresse: https://aiohttp.readthedocs.io/en/stable/glossary.html#term-asyncio.

[25] *Venmo Business-Rules Library*, [Online; aufgerufen am 10. Juni 2019]. Adresse: https://github.com/venmo/business-rules.

[26] *JSON-Rules-Engine*, [Online; aufgerufen am 11. Juni 2019]. Adresse: https://github.com/CacheControl/json-rules-engine.

[27] *Drools - Business-Rule-Management-System*, [Online; aufgerufen am 11. Juni 2019]. Adresse: https://www.drools.org/.

[28] *Durable-Rules*, [Online; aufgerufen am 11. Juni 2019]. Adresse: https://github.com/jruizgit/rules.

[29] *MQTT - Message Queuing Telementry Transport*, [Online; aufgerufen am 2. März 2019]. Adresse: https://mqtt.org/faq.

[30] *CoAP - Constrained Application Protocol*, [Online; aufgerufen am 2. März 2019]. Adresse: http://coap.technology/.

[31] *REST - Representational State Transfer*, [Online; aufgerufen am 2. März 2019]. Adresse: https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/.

[32] *SimpleIoTSimulator*, [Online; aufgerufen am 4. März 2019]. Adresse: https://www.simplesoft.com/SimpleIoTSimulator.html.

[33] *IBM Cloud Sensor Simulation*, [Online; aufgerufen am 4. März 2019]. Adresse: https://developer.ibm.com/recipes/tutorials/create-a-simulated-device-with-simulated-sensors/.

[34] *CARLA - Open-source simulator for autonomous driving research.* [Online; aufgerufen am 4. März 2019]. Adresse: http://carla.org/.

[35] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez und V. Koltun, „CARLA: An Open Urban Driving Simulator", in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, S. 1–16.

[36] *iFogSim*, [Online; aufgerufen am 4. März 2019]. Adresse: http://www.cloudbus.org/cloudsim/.

[37] *iFogSim Github Repository*, [Online; aufgerufen am 4. März 2019]. Adresse: https://github.com/Cloudslab/iFogSim.

[38] *SensorTrafficGenerator*, [Online; aufgerufen am 4. März 2019]. Adresse: https://github.com/vr000m/SensorTrafficGenerator.

[39] *MQTT Generator*, [Online; aufgerufen am 4. März 2019]. Adresse: https://gist.github.com/marianoguerra/be216a581ef7bc23673f501fdea0e15a.

[40] *Python Clean-Architecture with TDD*, [Online; aufgerufen am 2. Mai 2019]. Adresse: https://www.thedigitalcatonline.com/blog/2016/11/14/clean-architectures-in-python-a-step-by-step-example/.

[41] *The Clean-Architecture*, [Online; aufgerufen am 2. Mai 2019]. Adresse: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[42] *Can-BUS - Controller Area Network*, [Online; aufgerufen am 12. Juni 2019]. Adresse: https://en.wikipedia.org/wiki/CAN_bus.

[43] *SMART - Kriterien*, [Online; aufgerufen am 11. Juni 2019]. Adresse: https://en.wikipedia.org/wiki/SMART_criteria.

[44] *VSCode - Visual Studio Code*, [Online; aufgerufen am 2. Mai 2019]. Adresse: https://code.visualstudio.com/.

[45] *VSCode - Visual Studio Code*, [Online; aufgerufen am 2. Mai 2019]. Adresse: https://marketplace.visualstudio.com/VSCode.

[46] *GitLab Runner Installation*, [Online; aufgerufen am 19. Februar 2019]. Adresse: https://docs.gitlab.com/runner/install/.

[47] *GitLab Runner Linux Installation*, [Online; aufgerufen am 19. Februar 2019]. Adresse: https://docs.gitlab.com/runner/install/linux-manually.html.

[48] *Raspberry Pi 3 Model B*, [Online; aufgerufen am 2. Mai 2019]. Adresse: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[49] *netModule NB2800 MultiVehicle Router*, [Online; aufgerufen am 2. Mai 2019]. Adresse: http://netmodule.ch/en/products/vehicle-routers/NB2800-lte-advanced#_tabs_1.

[50] *netModule Wiki FAQ Flash und RAM*, [Online; aufgerufen am 2. Juni 2019]. Adresse: http://wiki.netmodule.com/faq/flash-and-ram.

[51] *netModule Lizenz Beschreibung*, [Online; aufgerufen am 2. Juni 2019]. Adresse: http://www.netmodule.com/products/accessories/licenses.html.

[52] *netModule LXC Container Setup Anleitung*, [Online; aufgerufen am 2. Juni 2019]. Adresse: http://wiki.netmodule.com/virtualisation/how-to-setup-a-container.

[53] *pytest: helps you write better programs*, [Online; aufgerufen am 3. Mai 2019]. Adresse: https://docs.pytest.org/en/latest/.

[54] *Yoyowallet Business-Rules Library*, [Online; aufgerufen am 10. Juni 2019]. Adresse: https://github.com/yoyowallet/business-rules.

[55] *Venmo Business-Rules UI*, [Online; aufgerufen am 10. Juni 2019]. Adresse: https://github.com/venmo/business-rules-ui.

[56] *ACM - Digital Library*, [Online; aufgerufen am 19. Februar 2019]. Adresse: https://www.acm.org/.

[57] *Choose an open source licence*, [Online; aufgerufen am 19. Februar 2019]. Adresse: https://choosealicense.com/.

[58] *Confluence - Team Collaboration Software*, [Online; aufgerufen am 19. Februar 2019]. Adresse: https://www.atlassian.com/software/confluence.

[59] *Gitlab Versionsverwaltung*, [Online; aufgerufen am 27. November 2018]. Adresse: https://gitlab.com/.

[60] *IEEE - Xplore Digital Library*, [Online; aufgerufen am 19. Februar 2019]. Adresse: https://www.ieee.org/.

[61] *Jira*, [Online; aufgerufen am 19. Februar 2019]. Adresse: https://www.atlassian.com/software/jira.

[62] *LaTeX - A document preparation system*, [Online; aufgerufen am 19. Februar 2019]. Adresse: https://www.latex-project.org/.

[63] *RClone - rsync for cloud storage*, [Online; aufgerufen am 2. März 2019]. Adresse: https://rclone.org/.

[64] *Swagger*, [Online; aufgerufen am 2. Mai 2019]. Adresse: https://swagger.io/.

[65] *Docker Ubuntu Installation*, [Online; aufgerufen am 2. Mai 2019]. Adresse: https://docs.docker.com/install/linux/docker-ce/ubuntu/.

[66] *Swagger Editor*, [Online; aufgerufen am 2. Mai 2019]. Adresse: http://editor.swagger.io/.

7.5 Entwicklungsumgebung

Der Code ist auf GitLab gehostet. Die Aufteilung der Repositories ist in Tabelle 7.1 aufgeführt.

Name	Zweck	URL
Gateway	REST API	`https://gitlab.com/dingouerinitx-ba/rule-engine-clean-arch`
Rule-Engine	Rule-Engine Services	`https://gitlab.com/dingouerinitx-ba/rule-engine`
Documentation	LaTeX Dokumentation	`https://gitlab.com/dingouerinitx-ba/ba-dokumentation`

Tabelle 7.1: Git Repositories

7.6 Entwicklungsanleitung

7.6.1 Installation Entwicklungsumgebung

Voraussetzung zur Ausführung der nachfolgenden Schritte ist eine vorhandene Linux Umgebung, welche entweder als Host oder virtuelle Machine läuft.

Zur Einrichtung von Docker [65] [1] auf Linux wurden folgende Schritte ausgeführt.

```
#Uninstall Old Docker Versions
sudo apt-get remove docker docker-engine docker.io containerd runc
#Setup Repository
sudo apt-get update
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
#Add Dockers Official PGP Key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
#Install Docker CE
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
#Test Docker
```

```
sudo docker run hello-world
# Use Docker as non-root user without sudo then do
sudo usermod -aG docker your-user
```

Zur Einrichtung von VSCode [44] wurden folgende Schritte ausgeführt.

```
#Add Repository and Key
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft
  ↪ .gpg
sudo install -o root -g root -m 644 microsoft.gpg /etc/apt/trusted.gpg.d/
sudo sh -c 'echo "deb [arch=amd64] https://packages.microsoft.com/repos/vscode
  ↪ stable main" > /etc/apt/sources.list.d/vscode.list'
#Update packages and install code
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install code # or code-insiders
```

7.6.2 Aufsetzen Rule-Engine Repository

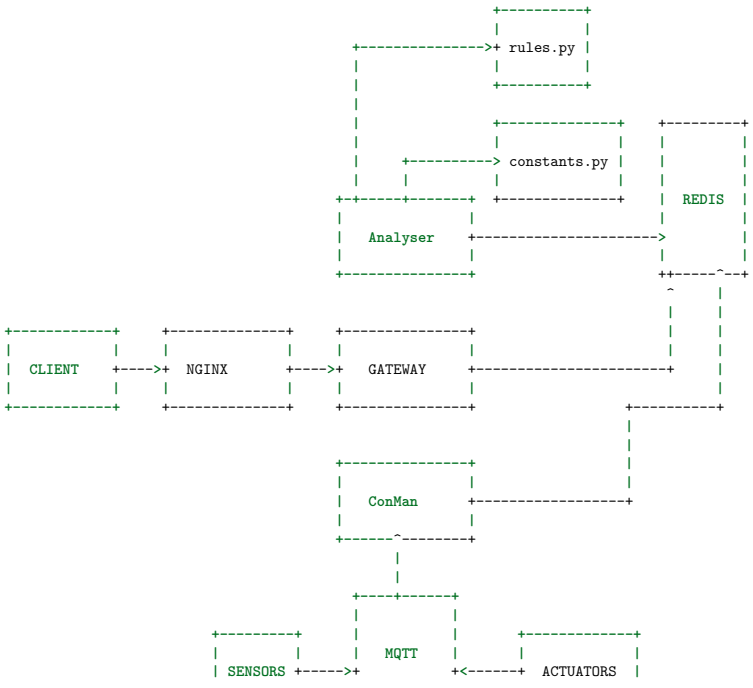
Das *README.md* des Repositories beschreibt, das Aufsetzen und die Benutzung und wurde daher mittels *pandoc* und *texlive* mittels folgenden Installationsbefehlen erstellt und als PDF eingefügt.

```
sudo apt-get install pandoc texlive
pandoc README.md -t latex -o README-rule-engine.pdf
```

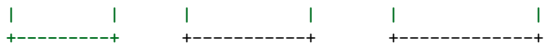
Backend Services Rule-Engine

The main backend services (*Analyser*, *ConMan*), *Mosquitto MQTT Broker* and *Redis Database* for development within a docker container. It is run by `docker-compose` with a **docker-compose.yml** file.

Overview



1



Usage

Start containers

```
docker-compose build --no-cache
docker-compose up
```

Restart scripts inside container

```
docker ps
docker exec -it ENGINE_CONTAINER_ID sh
$> supervisorctl restart COMPONENT_NAME_FROM_SUPERVISOR_INI
```

Development

The `app/` folder is mapped into the docker container and it is possible to work on the scripts on the fly, without restart of docker container. Only thing to do is to `supervisorctl restart ...` the script with the supervisor tool inside the container to make sure effects and changes are taking place.

Redis-cli

Start the redis comand line tool on your host or in the container

```
redis-cli
```

Some commands to check for containing data in redis.

Check StreamSensorMessage

```
$> xlen 'sensordata'
$> xrange 'sensordata' '-' '+'
```

Check latest SensorMessage per Sensor

```
$> get 'SENSOR_ID'
$> get 'd7e72836-6339-11e9-a8aa-27b2c181f989'
$> get '10b8301a-633a-11e9-a7cf-a373ae3dd760'
$> get '1057c680-633a-11e9-8abf-a7cb277ba226'
$> get '1132fac0-633a-11e9-9a30-23bf466785d0'
$> get 'd6d65a2a-6339-11e9-aa33-dbb318e8bbf7'
```

Check SensorList

```
$> smembers 'sensorlist'
```

Check ActauatorList

```
$>  smembers 'actuatorlist'
```

List all rules or a rule

```
$> hgetall 'rules'
$> hget 'rules' 'RULE_ID'
```

List ruledata (variables, actions, variable_type_operators)

```
$> get 'ruledata'
```

7.6.3 Aufsetzen Rule-Engine-Clean-Arch Repository

Das *README.md* des Repositories beschreibt das Aufsetzen und die Benutzung und wurde daher mittels *pandoc* und *texlive* mittels folgenden Installationsbefehlen erstellt und als PDF eingefügt.

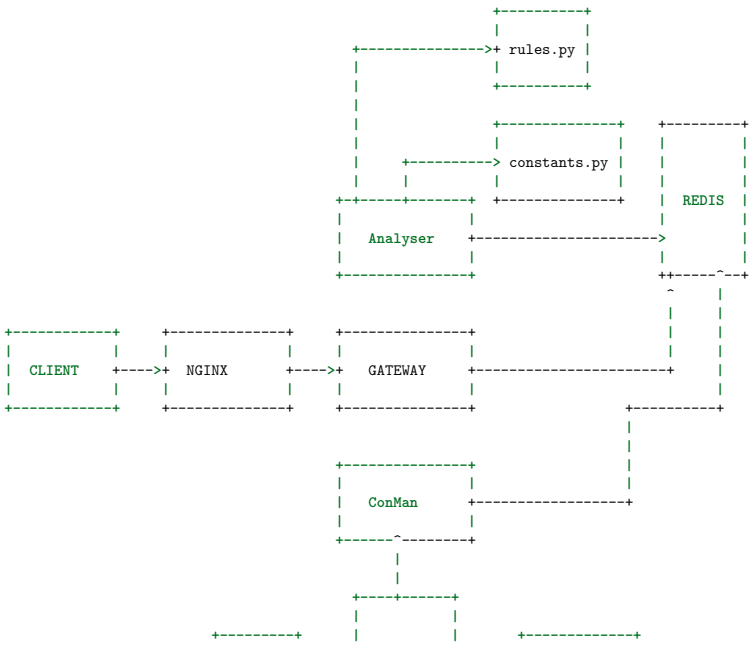
```
sudo apt-get install pandoc texlive
pandoc README.md -t latex -o README-rule-engine.pdf
```

Gateway Service Rule-Engine (clean architecture style)

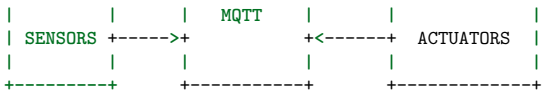
A TDD approach with a clean architecture from http://www.thedigitalcatonline.com/blog/2016/11/14/clean-architectures-in-python-a-step-by-step-example/

In this Repo we host the **Gateway** with a python *flask* webserver and *swagger*.

Overview



1



Usage

First of all lets create a virtual python environment **venv** in the `gateway` folder so that we can activate it.

```
python3 -m venv venv
source venv/bin/activate
```

Hint: to just deactivate the virtual environment just type `deactivate` anywhere in the terminal.

After this step all global `pip` and `python` commands are mapped into the activated virtual environment we created, so that we can use this commands as we would be in this folder at any place.

As a next step we now want to install all necessary dependencies to be able to run the gateway.

```
cd ..
pip install --upgrade pip
pip install -r requirements/dev.txt
```

As a special case we need to install the yoyowallet business-rules via our `*.tar.gz` file.

```
tar -xzvf business-rules-1.5.0.tar.gz
cd business-rules-1.5.0
../venv/bin/python3 setup.py install
cd ..
rm -rf business-rules-1.5.0
```

Next of all make sure you have a mapping for `redis` and `mqtt` to `127.0.0.1` in `/etc/hosts`

```
# Add container hostnames in /etc/hosts
sudo cp /etc/hosts /etc/hosts.bak
echo "127.0.0.1 redis" | sudo tee -a /etc/hosts
echo "127.0.0.1 mqtt" | sudo tee -a /etc/hosts
echo "[+] redis and mqtt localhost mapping added"
```

Hint: If we dont need the mapping anymore, we can restore the `hosts` file with the created `hosts.bak` file.

After this step we can run the flask application. We can set the `FLASK_ENV` environment variable to let flask know that we are under development. This

makes it possible to have more debugging output and when working on the python code of the application, to restart automatically on changes, which is very handy.

```
# export FLASK_ENV=development
flask run
# unset FLASK_ENV
```

Tests

To run the **style guide enforcement** test use

```
flake8
```

To run the **pytest** test use

```
py.test --sv
```

Or with coverage

```
py.test --cov-report term-missing --cov=gateway
```

Or just

```
pytest
```

Info

Generate timestamp in milliseconds with bash

```
echo $(($(date +%s%N)/1000000))
```

Setup a NEW Project

Important: This setup is just necessary when creating such a project from the ground up. Which means you can scip this part normally.

Create a virtual environment and activate it to use for the current installation of cookiecutter and the template.

```
python3 -m venv venv3
source venv3/bin/activate
pip install cookiecutter
cookiecutter https://github.com/audreyr/cookiecutter-pypackage
```

Next, deactivate and delete the virtaulenv and go into the created gateway folder to create a virtual environment for development

```
deactivate
rm -fR venv3
cd gateway
python3 -m venv venv3
source venv3/bin/activate
```

Go back into the root folder of your project and delete the `requirements_dev.txt` file and create a `requirements` folder and its files.

```
cd ..
mkdir requirements
touch requirements/prod.txt
touch requirements/dev.txt
touch requirements/test.txt
```

The `test.txt` file has the following content:

```
-r prod.txt

pytest
tox
coverage
pytest-cov
```

The `dev.txt` file has the following content:

```
-r test.txt

pip
wheel
flake8
Sphinx
```

The `requirements.txt` file has the following content:

```
-r requirements/prod.txt
```

Now its time to install the development requirements with the following command:

```
pip install -r requirements/dev.txt
```

The rest of the setup in **Step1** is exactly the same as in the step-by-step instruction from http://www.thedigitalcatonline.com/blog/2016/11/14/clean-architectures-in-python-a-step-by-step-example/

7.6.4 Installation GitLab Runner

Zur späteren Registrierung eines Runners muss als erstes die GitLab.com Seite aufgerufen werden. Unter **Settings - CI/CD - Runners** kann ein Token zur Registrierung bezogen werden. Folgende Abbildung 7.1 zeigt mögliche Werte.

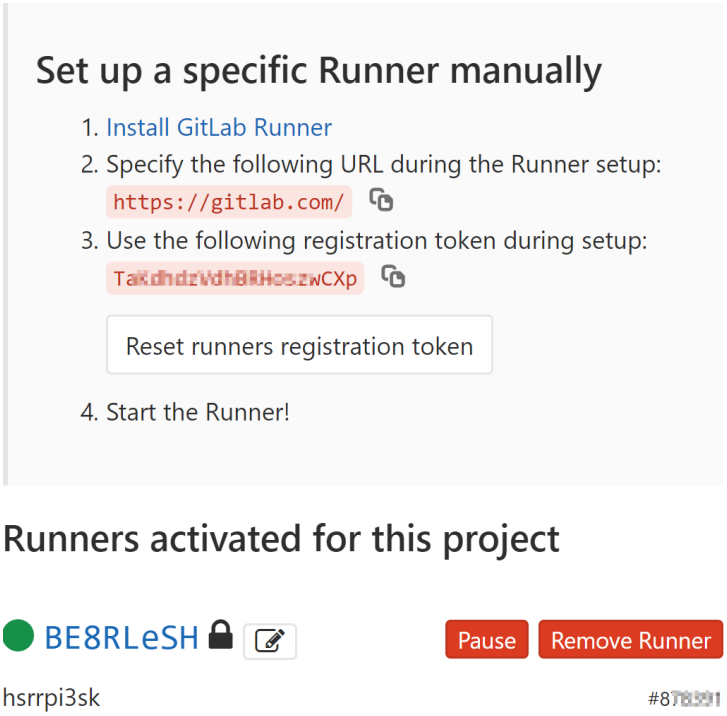


Abbildung 7.1: GitLab Settings Runner Token

Auf dem Gerät selbst werden folgende Befehle ausgeführt, welche in Listing 7.1 ersichtlich sind. Die genaue Dokumentation kann bei Bedarf auch auf der GitLab Runner Installationsseite nachlesen werden [47]. Zur Einfachheit und für den Vollzugriff, werden hier alle Befehle als Root ausgeführt, womit auch die CI/CD Befehle standardmässig als Root ausgeführt werden. Dies birgt zwar ein gewisses Risiko, ist aber für unsere Testumgebung nicht von Belang.

```
1  # Linux arm Version
2  wget -O /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.
     ↪ com/latest/binaries/gitlab-runner-linux-arm
3
4  # Make executable
5  chmod +x /usr/local/bin/gitlab-runner
6
7  # Runner config
8  cat /etc/gitlab-runner/config.toml
9
10 # Register a Runner with Token
11 gitlab-runner register
12
```

```
# Start Runner
#gitlab-runner start
gitlab-runner run &
```

Listing 7.1: GitLab Runner Linux Installation

7.6.5 GitLab CI/CD Konfiguration

GitLab CI/CD Konfiguration *.gitlab-ci.yml* für die Rule-Engine Services **Analyser**, **ConMan**, **Simulatoren** sowie **MQTT** und **Redis** DB.

```
stages:
- test
- deploy

test job:
image: python:latest
stage: test
tags:
    - hsr-server
script:
    - export PYTHONPATH=.
    - export FLASK_APP=gateway/app/web
    - apt-get update -qy
    - apt-get install -y python3-dev python3-pip curl
    - pip3 install -r requirements.txt
    - tar -xzvf business-rules-1.5.0.tar.gz
    - cd business-rules-1.5.0
    - python setup.py install
    - cd ..
    - rm -rf business-rules-1.5.0
    - pytest

lint job:
image: python:latest
stage: test
tags:
    - hsr-server
script:
    - pip3 install -r requirements.txt
    - flake8

deploy job:
stage: deploy
tags:
    - hsr-rpi3
script:
    - whoami
    - echo "$SHELL"
    - ps aux | grep python3
```

```
    - chmod +x netmodule_setup.sh netmodule_kill.sh
    - ./netmodule_setup.sh
    - ps aux | grep python3
when: manual
only:
    - master

kill job:
stage: deploy
tags:
    - hsr-rpi3
script:
    - whoami
    - echo "$SHELL"
    - ps aux | grep python3
    - chmod +x netmodule_setup.sh netmodule_kill.sh
    - ./netmodule_kill.sh
    - ps aux | grep python3
allow_failure: true
when: manual
only:
    - master
```

Das *netmodule_setup.sh* Skript.

```
#!/bin/bash

# If backup of hosts exists restore bakup hosts file first
if [ -f /etc/hosts.bak ]
then
    sudo cp /etc/hosts.bak /etc/hosts
    echo "[+] /etc/hosts backup restored"
fi

# Kill running python scripts before rerun
kill -9 $(ps aux | grep 'analyser.py' | grep -v grep | awk '{print $2}') 2>/dev/
 ↪ null
kill -9 $(ps aux | grep 'actuator.py' | grep -v grep | awk '{print $2}') 2>/dev/
 ↪ null
kill -9 $(ps aux | grep 'conman.py' | grep -v grep | awk '{print $2}') 2>/dev/null
kill -9 $(ps aux | grep 'sensors.py' | grep -v grep | awk '{print $2}') 2>/dev/
 ↪ null

# Update
sudo apt-get update

# Check and Install Curl
sudo apt-get install -y curl

# Check and Install Python3 and Pip3
sudo apt-get install -y python python3 python3-dev python3-pip supervisor
```

```

# Install Python3 Module dependencies
pip3 install --upgrade pip
pip3 install gunicorn paho-mqtt flasgger redis requests business-rules

# Check and Install Redis
sudo apt-get install -y redis-server
sudo systemctl enable redis-server.service

# Check and Install Broker (Mosquitto)
sudo apt-get install -y mosquitto

# Add container hostnames in /etc/hosts
sudo cp /etc/hosts /etc/hosts.bak
echo "127.0.0.1 redis" | sudo tee -a /etc/hosts
echo "127.0.0.1 mqtt" | sudo tee -a /etc/hosts
echo "[+] redis and mqtt localhost mapping added"

# Start Services (sensors, actuators, message manager, gateway, rule manager)
# TODO replace with supervisor commands
setsid nohup python3 engine/app/services/conman.py &>/dev/null &
setsid nohup python3 engine/app/services/analyser.py &>/dev/null &
setsid nohup python3 engine/app/services/ruleman.py &>/dev/null &
setsid nohup python3 engine/app/simulators/sensors.py engine/app/simulators/config
 ↪ .json &>/dev/null &
setsid nohup python3 engine/app/simulators/gps.py engine/app/simulators/gps_config
 ↪ .json &>/dev/null &
setsid nohup python3 engine/app/simulators/actuator.py &>/dev/null &
setsid nohup python3 engine/app/web/gateway.py &>/dev/null &

echo "[+] services started."
exit 0
```

Das *netmodule_kill.sh* Skript. Dieses Skript ermöglicht das Beenden von Diensten im Fall eines Problems ohne physikalischen Zugriff auf das System und mittels GitLab CI Pipeline.

```
# Kill running python scripts before rerun
kill -9 $(ps aux | grep 'analyser.py' | grep -v grep | awk '{print $2}') 2>/dev/
 ↪ null
kill -9 $(ps aux | grep 'actuator.py' | grep -v grep | awk '{print $2}') 2>/dev/
 ↪ null
kill -9 $(ps aux | grep 'conman.py' | grep -v grep | awk '{print $2}') 2>/dev/null
kill -9 $(ps aux | grep 'sensors.py' | grep -v grep | awk '{print $2}') 2>/dev/
 ↪ null
kill -9 $(ps aux | grep 'gps.py' | grep -v grep | awk '{print $2}') 2>/dev/null
exit 0
```

GitLab CI/CD Konfiguration *.gitlab-ci.yml* für den Rule-Engine-Clean-Arch **Gateway** Service.

```
# This file is a template, and might need editing before it works on your project.
# Official language image. Look for the different tagged releases at:
# https://hub.docker.com/r/library/python/tags/
image: python:latest

# Change pip's cache directory to be inside the project directory since we can
# only cache local items.
variables:
    PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

# Pip's cache doesn't store the python packages
# https://pip.pypa.io/en/stable/reference/pip_install/#caching
#
# If you want to also cache the installed packages, you have to install
# them in a virtualenv and cache it as well.
cache:
    paths:
    - .cache/pip
    - venv/

stages:
    - test
    - build
    - deploy

before_script:
    - python -V # Print out python version for debugging
    - pip install virtualenv
    - virtualenv venv
    - source venv/bin/activate
    - pip install -r requirements/dev.txt
    - tar -xzvf business-rules-1.5.0.tar.gz
    - cd business-rules-1.5.0
    - python setup.py install
    - cd ..
    - rm -rf business-rules-1.5.0

lint_job:
    stage: test
    tags:
    - hsr-server
    script:
    - python setup.py test
    - pip install flake8
    - flake8

test_job:
    stage: test
    tags:
    - hsr-server
    script:
    - pip install pytest
```

```
    - pytest

run_job:
    stage: build
    tags:
    - hsr-server
    script:
    - python setup.py bdist_wheel
    # an alternative approach is to install and run:
    - pip install dist/*
    # run the command here
    artifacts:
    paths:
    - dist/*.whl

pages_job:
    stage: build
    tags:
    - hsr-server
    script:
    - pip install sphinx sphinx-rtd-theme
    - cd docs ; make html
    - mv _build/html/ ../public/
    artifacts:
    paths:
    - public
    only:
    - master

deploy_job:
    stage: deploy
    tags:
    - hsr-rpi3
    script:
    - whoami
    - echo "$SHELL"
    - ps aux | grep python3
    - chmod +x netmodule_setup.sh netmodule_kill.sh
    - ./netmodule_setup.sh
    - ps aux | grep python3
    when: manual
    only:
    - master

kill job:
    stage: deploy
    tags:
    - hsr-rpi3
    script:
    - whoami
    - echo "$SHELL"
    - ps aux | grep python3
```

```
    - chmod +x netmodule_setup.sh netmodule_kill.sh
    - ./netmodule_kill.sh
    - ps aux | grep python3
    allow_failure: true
    when: manual
    only:
    - master
```

Das *netmodule_setup.sh* Skript.

```
#!/bin/bash


# Kill running python flask before rerun
kill -9 $(ps aux | grep 'flask run' | grep -v grep | awk '{print $2}') 2>/dev/null

# Update
sudo apt-get update

# Check and Install Curl
sudo apt-get install -y curl nginx

# Copy nginx config and restart nginx
sudo cp nginx_flask.conf /etc/nginx/conf.d/flask.conf
sudo nginx -t
sudo service nginx restart

# Check and Install Python3 and Pip3
sudo apt-get install -y python python3 python3-dev python3-pip supervisor

# Install Python3 Module dependencies
pip3 install --upgrade pip
pip3 install -r requirements/dev.txt


# Start flask service
setsid nohup flask run &>/dev/null &

echo "[+] flask started."
exit 0
```

Das *netmodule_kill.sh* Skript. Dieses Skript ermöglicht das Beenden von Diensten im Fall eines Problems ohne physikalischen Zugriff auf das System und mittels GitLab CI Pipeline.

```
# Kill running python flask before rerun
kill -9 $(ps aux | grep 'flask run' | grep -v grep | awk '{print $2}') 2>/dev/null
exit 0
```

7.7 Benutzeranleitung

Zur Erweiterung des Regelsatzes ist es nötig, weitere Variablen oder Aktionen je nach gewünschten Bedürfnissen zu formulieren und ins Projekt einzubetten. Dazu sei auf folgendes Repository `https://github.com/yoyowallet/business-rules` verwiesen oder die angehängte Anleitung weiter unten, in welchem das Erstellen dieser Variablen und Aktionen Schritt für Schritt erklärt wird.

business-rules

As a software system grows in complexity and usage, it can become burdensome if every change to the logic/behavior of the system also requires you to write and deploy new code. The goal of this business rules engine is to provide a simple interface allowing anyone to capture new rules and logic defining the behavior of a system, and a way to then process those rules on the backend.

You might, for example, find this is a useful way for analysts to define marketing logic around when certain customers or items are eligible for a discount or to automate emails after users enter a certain state or go through a particular sequence of events.

Usage

1. Define Your set of variables

Variables represent values in your system, usually the value of some particular object. You create rules by setting threshold conditions such that when a variable is computed that triggers the condition some action is taken.

You define all the available variables for a certain kind of object in your code, and then later dynamically set the conditions and thresholds for those.

For example:

```
class ProductVariables(BaseVariables):

    def __init__(self, product):
        self.product = product

    @numeric_rule_variable
    def current_inventory(self):
        return self.product.current_inventory

    @numeric_rule_variable(label='Days until expiration')
    def expiration_days(self)
        last_order = self.product.orders[-1]
        return (last_order.expiration_date - datetime.date.today()).days

    @string_rule_variable()
    def current_month(self):
        return datetime.datetime.now().strftime("%B")

    @select_rule_variable(options=Products.top_holiday_items())
    def goes_well_with(self):
```

1

```
        return products.related_products

    @numeric_rule_variable(params=[{
        'field_type': FIELD_NUMERIC,
        'name': 'days',
        'label': 'Days'
    }])
    def orders_sold_in_last_x_days(self, days):
        count = 0
        for order in self.product.orders:
            if (datetime.date.today() - order.date_sold).days < days:
                count += 1
        return count
```

2. Define your set of actions

These are the actions that are available to be taken when a condition is triggered.

For example:

```
class ProductActions(BaseActions):

    def __init__(self, product):
        self.product = product

    @rule_action(params={"sale_percentage": FIELD_NUMERIC})
    def put_on_sale(self, sale_percentage):
        self.product.price = (1.0 - sale_percentage) * self.product.price
        self.product.save()

    @rule_action(params={"number_to_order": FIELD_NUMERIC})
    def order_more(self, number_to_order):
        ProductOrder.objects.create(product_id=self.product.id,
                                    quantity=number_to_order)
```

If you need a select field for an action parameter, another -more verbose- syntax is available:

```
class ProductActions(BaseActions):

    def __init__(self, product):
        self.product = product

    @rule_action(params=[{'fieldType': FIELD_SELECT,
                          'name': 'stock_state',
                          'label': 'Stock state',
                          'options': [
```

```
                            {'label': 'Available', 'name': 'available'},
                            {'label': 'Last items', 'name': 'last_items'},
                            {'label': 'Out of stock', 'name': 'out_of_stock'}
                        ]}])
    def change_stock_state(self, stock_state):
        self.product.stock_state = stock_state
        self.product.save()
```

If you introduced a new action parameter but already have an older set of active Rules, which will all require to be updated with this new parameter, you can use `ActionParam` class which contains default value for parameter along with field type:

```
from business_rules.actions import ActionParam, BaseActions, rule_action
from business_rules import fields

class ProductActions(BaseActions):

    def __init__(self, product):
        self.product = product

    @rule_action(params={"sale_percentage": fields.FIELD_NUMERIC,
                         "on_sale": ActionParam(field_type=fields.FIELD_BOOLEAN,
                                                default_value=False}
                                                )
    def put_on_sale(self, sale_percentage, on_sale):
        self.product.price = (1.0 - sale_percentage) * self.product.price
        self.on_sale = on_sale
        self.product.save()
```

3. Build the rules

A rule is just a JSON object that gets interpreted by the business-rules engine.

Note that the JSON is expected to be auto-generated by a UI, which makes it simple for anyone to set and tweak business rules without knowing anything about the code. The javascript library used for generating these on the web can be found here.

An example of the resulting python lists/dicts is:

```
rules = [
# expiration_days < 5 AND current_inventory > 20
{ "conditions": { "all": [
      { "name": "expiration_days",
        "operator": "less_than",
        "value": 5,
```

```
      },
      { "name": "current_inventory",
        "operator": "greater_than",
        "value": 20,
      },
  ]},
  "actions": [
      { "name": "put_on_sale",
        "params": {"sale_percentage": 0.25},
      },
  ],
},

# current_inventory < 5 OR (current_month = "December" AND current_inventory < 20)
{ "conditions": { "any": [
            {
                "name": "current_inventory",
                "operator": "less_than",
                "value": 5,
            },
            {
                "all": [
                    {
                        "name": "current_month",
                        "operator": "equal_to",
                        "value": "December",
                    },
                    {
                        "name": "current_inventory",
                        "operator": "less_than",
                        "value": 20,
                    }
                ]
            }
        ]
  },
  "actions": [
    { "name": "order_more",
      "params":{"number_to_order": 40},
    },
  ],
},
# orders_sold_in_last_x_days(5) > 10
{
    "conditions": { "all": [
        {
```

```
            "name": "orders_sold_in_last_x_days",
            "operator": "greater_than",
            "value": 10,
            "params": {"days": 5},
        }
    ]},
    "actions": [
        {
            "name": "order_more",
            "fields": [{"name": "number_to_order", "value": 40}]
        }
    ]
}]
```

Export the available variables, operators and actions

To e.g. send to your client so it knows how to build rules

```
from business_rules import export_rule_data
export_rule_data(ProductVariables, ProductActions)
```

that returns

```
{
  "variables": [
    {
      "name": "expiration_days",
      "label": "Days until expiration",
      "field_type": "numeric",
      "options": [],
      "params": []
    },
    {
      "name": "current_month",
      "label": "Current Month",
      "field_type": "string",
      "options": [],
      "params": []
    },
    {
      "name": "goes_well_with",
      "label": "Goes Well With",
      "field_type": "select",
      "options": [
        "Eggnog",
        "Cookies",
```

```
      "Beef Jerkey"
    ],
    "params": []
  },
  {
    "name": "orders_sold_in_last_x_days",
    "label": "Orders Sold In Last X Days",
    "field_type": "numeric",
    "options": [],
    "params": [
      {
        "field_type": "numeric",
        "name": "days",
        "label": "Days"
      }
    ]
  }
],
"actions": [
  {
    "name": "put_on_sale",
    "label": "Put On Sale",
    "params": {
      "sale_percentage": "numeric"
    }
  },
  {
    "name": "order_more",
    "label": "Order More",
    "params": {
      "number_to_order": "numeric"
    }
  }
],
"variable_type_operators": {
  "numeric": [
    {
      "name": "equal_to",
      "label": "Equal To",
      "input_type": "numeric"
    },
    {
      "name": "less_than",
      "label": "Less Than",
      "input_type": "numeric"
    },
```

```
      {
        "name": "greater_than",
        "label": "Greater Than",
        "input_type": "numeric"
      }
    ],
    "string": [
      {
        "name": "equal_to",
        "label": "Equal To",
        "input_type": "text"
      },
      {
        "name": "non_empty",
        "label": "Non Empty",
        "input_type": "none"
      }
    ]
  }
}
```

To validate rule data:

```
from business_rules import validate_rule_data
is_valid = validate_rule_data(ProductVariables,
                              ProductActions,
                              {'conditions':[], 'actions':[]}
                              )
```

Run your rules

```
from business_rules import run_all

rules = _some_function_to_receive_from_client()

for product in Products.objects.all():
    run_all(rule_list=rules,
            defined_variables=ProductVariables(product),
            defined_actions=ProductActions(product),
            stop_on_first_trigger=True,
           )
```

API

Variable Types and Decorators:

The type represents the type of the value that will be returned for the variable and is necessary since there are different available comparison operators for different types, and the front-end that's generating the rules needs to know which operators are available.

All decorators can optionally take a label: - `label` - A human-readable label to show on the frontend. By default we just split the variable name on underscores and capitalize the words. - `params` - A list of parameters that will be passed to the variable when its value is calculated. The list elements should be dictionaries with a `field_type` to specify the type and `name` that corresponds to an argument of the variable function.

The available types and decorators are:

`numeric` - an integer, float, or python Decimal.

`@numeric_rule_variable` operators:

- `equal_to`
- `greater_than`
- `less_than`
- `greater_than_or_equal_to`
- `less_than_or_equal_to`

Note: to compare floating point equality we just check that the difference is less than some small epsilon

`string` - a python bytestring or unicode string.

`@string_rule_variable` operators:

- `equal_to`
- `starts_with`
- `ends_with`
- `contains`
- `matches_regex`
- `non_empty`

`boolean` - a True or False value.

`@boolean_rule_variable` operators:

- `is_true`
- `is_false`

`select` - a set of values, where the threshold will be a single item.

`@select_rule_variable` operators:

- `contains`
- `does_not_contain`

`select_multiple` - a set of values, where the threshold will be a set of items.

`@select_multiple_rule_variable` operators:

- `contains_all`
- `is_contained_by`
- `shares_at_least_one_element_with`
- `shares_exactly_one_element_with`
- `shares_no_elements_with`

`datetime` - a Timestamp value

A rule variable accepts the following types of values:

- int
- string with format `%Y-%m-%dT%H:%M:%S`
- string with format `%Y-%m-%d`

A variable can return the following types of values:

- int
- datetime
- date
- string with format `%Y-%m-%dT%H:%M:%S`
- string with format `%Y-%m-%d`

`@datetime_rule_variable` operators:

- `equal_to`
- `before_than`
- `before_than_or_equal_to`
- `after_than`
- `after_than_or_equal_to`

`time` - a Time value

A rule variable accepts the following types of values:

- string with format `%H:%M:%S`
- string with format `%H:%M`

A variable can return the following types of values:

- datetime
- time
- string with format `%H:%M:%S`
- string with format `%H:%M`

`@time_rule_variable` operators:

- `equal_to`
- `before_than`
- `before_than_or_equal_to`
- `after_than`
- `after_than_or_equal_to`

Contributing

Open up a pull request, making sure to add tests for any new functionality. To set up the dev environment (assuming you're using virtualenvwrapper):

```
$ mkvirtualenv business-rules
$ pip install -r dev-requirements.txt
$ nosetests
```

7.8 API Dokumentation

Mittels Swagger [64] und dem Swagger Editor [66] ist es uns möglich, unsere Swagger API Beschreibung im JSON Format in HTML umzuwandeln.

Anschliessend wird das HTML mittels *wkhtmltopdf* in PDF umgewandelt, um es schliesslich hier einzufügen.

```
1 wkhtmltopdf index.html swagger-api.pdf
```

Im nachfolgenden ist die Swagger API Beschreibung unserer Applikation ersichtlich.

200
Returns the ruledata to set rules
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /rules

Up

Get all rules from the memrepo (**rulesGet**)

Responses
200
Returns a json formated set of added rules
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

POST /rules

Up

Create a new rule from a json string (**rulesPost**)

Request body

body body (required)
Body Parameter — Send a Rule in JSON format

Return type
inline_response_200

Example data
Content-Type: application/json

```
{
  "rule_id" : "rule_id",
  "conditions" : "conditions",
  "actions" : "actions"
}
```

Responses
200
Return of rule with rule_id when successful inline_response_200
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

DELETE /rules/{rule_id}

Up

Delete a single rule by rule_id (**rulesRuleIdDelete**)

Path parameters

rule_id (required)
Path Parameter —

Return type
inline_response_200

Example data
Content-Type: application/json

```
{
```

```
  "rule_id" : "rule_id",
  "conditions" : "conditions",
  "actions" : "actions"
}
```

Responses
200
Return of rule with rule_id when successful inline_response_200
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /rules/{rule_id}

Up

Get a single rule by rule_id (**rulesRuleIdGet**)

Path parameters

rule_id (required)
Path Parameter —

Return type
inline_response_200

Example data
Content-Type: application/json

```
{
  "rule_id" : "rule_id",
  "conditions" : "conditions",
  "actions" : "actions"
}
```

Responses
200
Return of rule with rule_id when successful inline_response_200
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /sensor/{sensor_id}

Up

Get latest sensor value and info from a sensor registered to the gateway (**sensorSensorIdGet**)

Path parameters

sensor_id (required)
Path Parameter —

Responses
200
Immediate response of single device info
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /sensorlist

Up

Get all sensor_ids as list from the memrepo (**sensorlistGet**)

Responses
200

Returns a json formatted sensorlist
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /streams/filter/description/{filter_by_description}

Up

Get all stream SensorMessages from the redis stream by description
(**streamsFilterDescriptionFilterByDescriptionGet**)

Path parameters

filter_by_description (required)
Path Parameter —

Responses
200
Returns a json formated stream of all sensordata by type
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /streams/filter/id/{filter_by_id}

Up

Get all stream SensorMessages from the redis stream by id (**streamsFilterIdFilterByIdGet**)

Path parameters

filter_by_id (required)
Path Parameter —

Responses
200
Returns a json formated stream of all sensordata by id
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /streams/filter/type/{filter_by_type}

Up

Get all stream SensorMessages from the redis stream by type (**streamsFilterTypeFilterByTypeGet**)

Path parameters

filter_by_type (required)
Path Parameter —

Responses
200
Returns a json formated stream of all sensordata by type
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /streams/filter/unit/{filter_by_unit}

Up

Get all stream SensorMessages from the redis stream by unit (**streamsFilterUnitFilterByUnitGet**)

Path parameters

filter_by_unit (required)
Path Parameter —

Responses
200
Returns a json formated stream of all sensordata by type
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /streams/filter/value/{filter_by_value}

Up

Get all stream SensorMessages from the redis stream by value (**streamsFilterValueFilterByValueGet**)

Path parameters

filter_by_value (required)
Path Parameter —

Responses
200
Returns a json formated stream of all sensordata by type
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /streams

Up

Get all stream SensorMessages from the memrepo (**streamsGet**)

Responses
200
Returns a json formated set of all sensordata
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

Models

[Jump to Methods]

Table of Contents

body - **Up**

actions (optional)
String

conditions (optional)
String

inline_response_200 - **Up**

actions (optional)

String

conditions (optional)
String

rule_id (optional)
String

A swagger API

powered by Flasgger
More information: https://helloreverb.com
Contact Info: hello@helloreverb.com
Version: 0.0.1
BasePath:

http://apache.org/licenses/LICENSE-2.0.html

Access

Methods

[Jump to Models]

Table of Contents

Default

GET /actuatorlist

Up

Get all actuator_id's as a list from the repo (**actuatorlistGet**)

Responses
200
Returns a json formatted actuatorlist
400
ParameterError Failure
404
ResourceError Failure
500
SystemError Failure

GET /

Up

The main page with endpoint descriptions (**rootGet**)

Responses
200
Main page with its endpoint descriptions
400
An error occured
500
SystemError Failure

GET /ruledata

Up

Get all the variables, actions and variable_type_operators, which can be used for setting rules (**ruledataGet**)

Responses

7.9 Testprotokoll VM lokal 13.06.2019

Voraussetzungen

Als Ausgangslage der Testdurchführung wurde eine virtuelle Machine in VMWare Workstation 15.1.0 mit Ubuntu 16.04 LTS als Host Betriebssystem, Firefox Quantum in der Version 67.0.1 als Browser sowie Docker in der Version 18.09.6 verwendet. Das Repository *rule-engine* mit den Services wurde mittels *docker-compose up* hochgefahren und anschliessend im Repository *rule-engine-clean-arch* der Gateway mittels *source venv/bin/activate* und *flask run* gestartet.

Vorbemerkungen

- Die Tests leiten sich mehrheitlich aus den Anforderungen ab, insbesondere den Use Cases sowie nicht funktionalen Anforderungen.
- Für nicht implementierte Funktionalitäten im Rahmen der Projektarbeit sind keine Testfälle abgebildet, da diese nicht getestet werden können.
- Aufgrund der Optionalität des User-Interfaces gibt es keine expliziten Anforderungen und es wird darauf geachtet möglichst User-Interface neutral zu berichten.

Tests

Test	**Zustand / Bemerkungen**
T1: Initialisierung	Aufruf der URL: http://127.0.0.1:5000/. Die Startseite erscheint.
T2: Abrufen Sensorliste	Aufruf der URL: GET http://127.0.0.1:5000/sensorlist. Die Sensorliste als JSON und mit Statuscode 200 erscheint.
T3: Abrufen Aktuatorliste	Aufruf der URL: GET http://127.0.0.1:5000/actuatorlist. Die Aktuatorliste als JSON und mit Statuscode 200 erscheint.
T4: Abrufen RuleSet	Aufruf der URL: GET http://127.0.0.1:5000/ruledata. Die RuleSet Daten als JSON und mit Statuscode 200 erscheinen.
T5: Aufrufen des RuleBuilder UI	Aufruf der URL: GET http://127.0.0.1:5000/ui. Das RuleBuilder UI als HTML und mit Statuscode 200 erscheint. Die gefetchten Daten werden abgeholt und es ist möglich Regeln zusammenzustellen

T6: Regel erstellen (gültig)	Aufruf der URL: POST http://127.0.0.1:5000/rules. Body-Content: Gültige Regel als JSON. Antwort Statuscode 200 mit gesendeter Regel als JSON.
T7: Regel erstellen (ungültig)	Aufruf der URL: POST http://127.0.0.1:5000/rules. Body-Content: Ungültige Regel als JSON. Antwort Statuscode 500 mit einer Nachricht als JSON und dem Inhalt: Exception Invalid Rule.
T8: Regeln anzeigen	Aufruf der URL: GET http://127.0.0.1:5000/rules. Die gespeicherte gültige Regel wird als JSON mit Statuscode 200 zurückgegeben.
T9: Regel löschen	Aufruf der URL: DELETE http://127.0.0.1:5000/rules/3b05ae9e-dbfb-49fe-b210-f138c1aa6f08 mit der Rule-ID der gespeicherten gültigen Regel wird abgeschickt. Antwort ist die gelöschte Regel mit Statuscode 200.
T10: Sensordaten anzeigen	Aufruf der URL: GET http://127.0.0.1:5000/streams. Eine Liste von Sensordaten als JSON mit Statuscode 200 wird zurückgegeben.
T11: Sensordaten Filter Description	Aufruf der URL: GET http://127.0.0.1:5000/streams/filter/desc/Motor. Eine Liste von Sensordaten als JSON mit Statuscode 200 wird zurückgegeben.
T12: Sensordaten Filter ID	Aufruf der URL: GET http://127.0.0.1:5000/streams/filter/id/d6d65a2a-6339-11e9-aa33-dbb318e8bbf7. Eine Liste von Sensordaten als JSON mit Statuscode 200 wird zurückgegeben.
T13: Sensordaten Filter Type	Aufruf der URL: GET http://127.0.0.1:5000/streams/filter/type/temp. Eine Liste von Sensordaten als JSON mit Statuscode 200 wird zurückgegeben.
T14: Sensordaten Filter Unit	Aufruf der URL: GET http://127.0.0.1:5000/streams/filter/unit/C. Eine Liste von Sensordaten als JSON mit Statuscode 200 wird zurückgegeben.

T15: Sensordaten Filter Value	Aufruf der URL: GET http://127.0.0.1:5000/streams/filter/value/50. Eine Liste von Sensordaten als JSON mit Statuscode 200 wird zurückgegeben.

Schlussbemerkungen

Die exakt gleichen Tests wurden auf unserem Raspberry Pi Deployment-System durchgeführt und verliefen bis auf das Holen der SensorDaten Streams exakt gleich ab. Beim abholen der Streams wurde der Fehler *redis ERR unknown command XRANGE* geworfen. Nach einer ganzen Reihe Debugging via Logs, Logfiles und Redis-Cli Tests, konnte das Problem eingegrenzt jedoch nicht behoben werden. Grund scheint das nicht Vorhandensein des *XRANGE* Befehls und damit der Verwendung von Streams auf dem Raspberry Pi Redis-Servers zu sein. Demnach ist dies kein Fehler unserer Applikation selbst sondern des Infrastruktur-Setups auf dem Raspberry Pi und der installierten *Redis-Server* Umgebung.