ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

STL SYSTEMTECHNIKLABOR

Konzeption und Implementierung eines Microservices zur Metadatenverwaltung von Sensoren im Kontext einer Smart City

Philipp Gölter

Technical Report – STL-TR-2022-02 – ISSN 2364-7167

stl.htwsaar.de

Technische Berichte des Systemtechniklabors (STL) der htw saar
Technical Reports of the System Technology Lab (STL) at htw saar
ISSN 2364-7167

Philipp Gölter: Konzeption und Implementierung eines Microservices zur Metadatenverwaltung von Sensoren im Kontext einer Smart City
Technical report id: STL-TR-2022-02

First published: May 2022
Last revision: November 2022
Internal review: Markus Esch, Klaus Berberich

For the most recent version of this report see: https://stl.htwsaar.de/

Title image source: Colourbox, https://www.colourbox.de/bild/internet-skyline-iot-bild-25213100

htw saar – Hochschule für Technik und Wirtschaft des Saarlandes (University of Applied Sciences)
Fakultät für Ingenieurwissenschaften (School of Engineering)
STL – Systemtechniklabor (System Technology Lab)
Prof. Dr.-Ing. André Miede (andre.miede@htwsaar.de)
Goebenstraße 40
66117 Saarbrücken, Germany
https://stl.htwsaar.de

ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Master-Thesis

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Konzeption und Implementierung eines Microservices zur Metadatenverwaltung von Sensoren im Kontext einer Smart City

vorgelegt von
Philipp Gölter

betreut und begutachtet von
Prof. Dr. Markus Esch
Prof. Dr.-Ing. Klaus Berberich

Saarbrücken, 14.05.2022

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 14.05.2022

Philipp Gölter

Zusammenfassung

Das Konzept Smart City beschäftigt sich damit, städtische Prozesse zu automatisieren und selbstorganisiert zu steuern. Durch die Integration und Vernetzung einer Vielzahl von Sensoren und Aktoren soll dieses Ziel erreicht werden. Das dadurch resultierende komplexe Netzwerk erfordert einen hohen Verwaltungsaufwand. Wie können verschiedene Sensoren auf einfache Weise in ein solches System eingebunden werden? Wie verwaltet man Metadaten in einem solchen System? Wie geht man mit heterogenen Daten um? Mit diesen und weiteren Fragen beschäftigt sich diese Arbeit, indem ein Konzept erarbeitet und evaluiert wird, um eine geeignete Lösung zu finden, diesen Problemen zu begegnen. Dazu wird im Rahmen des CiTe-Testfelds zur Erforschung von Smart City-Themen, ein Microservice zur Verwaltung von Sensor-Metadaten konzipiert und umgesetzt. Zur Verwaltung wird eine universelle Schnittstelle bereitgestellt, über welche Nutzer mit der Anwendung interagieren können. Um Daten zu persistieren, wird evaluiert, welche Anforderungen ein Datenhaltungssystem im gegebenen Kontext erfüllen muss und ein geeignetes System ausgewählt.

Die Anwendung zur Verwaltung, sowie das Datenbanksystem werden in einer Cluster-Umgebung bereitgestellt und können beliebig skaliert werden. Damit ein fehlerfreier Betrieb gewährleistet werden kann, werden zusätzlich Monitoring-Anwendungen bereitgestellt, welche eine Überwachung aller auf dem Cluster verfügbaren Anwendungen ermöglichen.

Weiterhin wird eine Anwendung konzipiert und entwickelt um Sensoren über ein Gateway mit dem Netzwerk zu verbinden. Dabei sollen die Metadaten der Sensoren automatisch bereitgestellt werden. Zu Kommunikation wird eine einheitliche Kommunikationsstruktur auf Basis der Message Queuing Telemetry Transport (MQTT)-Spezifikation konzipiert. Zuletzt erfolgt eine Evaluation der umgesetzten Lösung in Hinblick auf Qualitative und Quantitative Merkmale.

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahrzehnten hat die Urbanisierung von Lebensraum stetig zugenommen. Laut Prognosen werden voraussichtlich bis zum Jahr 2050 rund 68% der Weltbevölkerung in städtischen Gebieten leben. [66] Dieser rapide Zuwachs bringt diverse Herausforderungen mit sich. Resssourcen- und Energieknappheit, Verkehrsüberlastung aber auch eine nachhaltige Lebensweise, sowie Sicherheit im urbanen Lebensraum stellen komplexe zu bewältigende Probleme dar. Um diese Probleme zu lösen, wird immer mehr auf den stetigen Fortschritt im Bereich der Informations- und Kommunikationstechnologie zurückgegriffen. [69] Insbesondere *Internet of Things (IoT)* bezogene Technologien erhalten zunehmend Einzug in die urbane Infrastruktur. Miteinander vernetzte physische Geräte [1] bieten so zuvor nicht vorhandene Steuerungsmöglichkeiten und erlauben es, eine Vielzahl von Daten zu erfassen und zu verarbeiten. Mit den dadurch gewonnenen Informationen sollen urbane Prozesse effizienter gestaltet, bedarfsgerechter ausgeführt und automatisiert gesteuert werden. Eine *Smart City* stellt unter Zuhilfenahme von modernen technologischen Konzepten Dienste und Infrastrukturen der neuen Generation bereit. [69]

Ein solches weitläufiges Informationsnetz aus verschiedenen Geräten bringt ganz eigene Herausforderungen unter anderem in Bezug auf Verwaltung, Interoperabilität, Verfügbarkeit, Sicherheit, Stabilität und Skalierbarkeit mit sich. Mit diesen Problemen beschäftigt sich das Distributed Systems Lab (DSL) der Hochschule für Technik und Wirtschaft des Saarlandes. Im Kontext der Erforschung und Konzeption von Smart Cities wurde das Forschungsprojekt *CiTe* ins Leben gerufen. Das CiTe-Testfeld simuliert ein modernes städtisches Umfeld und dient somit als Grundlage für Forschungsprojekte, welche sich mit verschiedensten Anwendungszenarien beschäftigen.

1.1 Motivation

Im Rahmen des CiTe-Testfeldes beschäftigt sich diese Arbeit mit der Konzeption und Prototypisierung eines Systems zu Verwaltung von Metadaten und der Registrierung von Sensoren und Aktoren. Zentraler Punkt ist die Anbindung von verschiedenen Sensoren und Aktoren, sowie deren Verwaltung. Diese sollen auf möglichst einfache Weise (Plug & Play) an das System angebunden werden. Damit neue, in das System eingebundene Services mit den Sensoren/Aktoren interagieren können, ist eine Verwaltungs-Instanz erforderlich, welche die Aufgabe übernimmt, Informationen über angeschlossene Sensoren und Aktoren zu verwalten. Weiterhin sollte dieser Service eine REST-Schnittstelle anbieten um Sensoren entsprechend zu konfigurieren. Die Konfiguration soll in Form von Metadaten im Datenhaltungssystem persistiert werden.

Hierbei gilt es eine geeignete Datenstruktur zu entwickeln, welche alle Konfigurationsmöglichkeiten und grundlegende Informationen zu Sensoren/Aktoren vollumfänglich abbildet. Die benötigten Metadaten umfassen unter anderem Informationen zu Status, Typ, Hersteller, Standort von Sensoren oder auch Verweise zu entsprechenden Topics des

[1] Sensoren und Aktoren

Message-Brokers auf welchen die Sensoren und Aktoren ihre Daten kommunizieren. Initial soll für einen neu angeschlossenen Sensor möglichst automatisch ein neuer Eintrag im Verwaltungssystem angelegt werden. Weiterhin sollte nachträglich die Möglichkeit bestehen Sensoren, welche weiteren Konfigurationsbedarf benötigen über eine entsprechend vom Verwaltungs-Service bereitgestellte Schnittstelle, zu konfigurieren.

Diese Schnittstelle soll zum einen eine nachträgliche Konfiguration der Metadaten ermöglichen und zum anderen Funktionalität zum Abrufen von Metadaten bieten. Mögliche Anwendungsfälle erfordern somit das Abrufen von Metadaten von einzelnen oder mehreren Sensoren. Des Weiteren soll das System einem möglichst modularen Aufbau entsprechen, um eine einfache Erweiterbarkeit zu gewährleisten. Die Datenhaltung ist ebenfalls ein weiterer wichtiger Punkt. Neu angeschlossene Sensoren müssen möglichst ohne Verzögerung im System registriert werden können, sowie auch bei horizontaler Skalierung konsistent und stets erreichbar über alle Instanzen der genutzten Datenhaltungssysteme sein. Dementsprechend muss eine Strategie entwickelt werden um dies bestmöglich zu gewährleisten.

1.2 Aufbau der Arbeit

Die Arbeit dokumentiert die wesentlichen Schritte der Analyse, Konzeption, Implementierung und Evaluation des zu entwickelnden Systems. In Kapitel 2 werden zunächst wesentliche Grundlagen, welche zum Verständnis der Arbeit benötigt werden, erläutert. Darunter relevante theoretische Konzepte, verwendete Software und Programmbibliotheken, sowie die zu verwendende Hardware und Infrastruktur.

Kapitel 3 beschäftigt sich mit verwandten Arbeiten zur gegebenen Problemstellung. Es werden ausgewählte Arbeiten vorgestellt die ebenfalls im Rahmen des CiTe-Testfeldes durchgeführt wurden. Gemäß dem Stand der aktuellen Entwicklung bei Systemen zur Verwaltung von IoT-Geräten wird eine Softwarelösung betrachtet. Im Anschluss werden zwei bestehende Vorschläge zu einer standardisierten Kommunikationsstruktur über das Transportprotokoll MQTT vorgestellt. Im letzten Abschnitt werden Probleme bei der Entwicklung von einheitlich nutzbaren Standards im IoT-Bereich dargestellt.

Anschließend werden in Kapitel 4 Anwendungsfälle für die vorliegende Thematik betrachtet und entsprechende Anforderungen abgeleitet, die das Endergebnis erfüllen soll.

Unabhängig von Technologien erfolgt in Kapitel 5 die Konzeption einer Kommunikationsstruktur via MQTT-Kommunikation, sowie die Entwicklung eines Konzepts zur Anbindung der Sensoren, als auch eines Microservice zu Verwaltung der Metadaten von angebundenen Geräten. Weiterhin wird eine Lösung zur Überwachung der Clusteranwendungen konzipiert und diskutiert.

In Kapitel 6 erfolgt eine konkrete Umsetzung des im vorigen Kapitel erstellten Konzepts. Dabei ist die Implementierung anhand der zu entwickelten Komponenten in separate Abschnitte unterteilt.

Die Evaluation der umgesetzten Lösung erfolgt in Kapitel 7. Zunächst wird ein Fertigstellungsgrad anhand der erfüllten Anforderungen ermittelt. Anschließend erfolgt eine detaillierte Betrachtung der umgesetzten Komponenten.

Zuletzt erfolgt in Kapitel 8 eine Zusammenfassung der Arbeit und eine Ausführung auf zukünftige Entwicklungen im Rahmen dieser Arbeit und des CiTe-Testfeldes.

2 Grundlagen

Konzeption und Umsetzung einer verteilten Softwarearchitektur erfordern ein umfassendes Verständnis von theoretischen Konzepten und Technologien. In diesem Kontext werden alle erforderlichen Grundlagen im Folgenden erläutert.

2.1 Konzepte

2.1.1 Microservices

Um größere monolithische Systeme wartbar und verständlich zu gestalten, findet häufig das Konzept der Modularisierung Anwendung. Dabei werden große Systeme in kleinere Module aufgeteilt, welche jeweils eine der Geschäftsdomäne zu Grunde liegende Funktionalität implementieren. Das Konzept Microservices beschreibt ein weiter gedachtes Paradigma der Modularisierung, damit eine größere Unabhängigkeit zwischen einzelnen Modulen erreicht werden kann. Es unterscheidet sich zur klassischen Vorgehensweise darin, dass ein einzelner Service ein einzelnes Modul implementiert. Einzelne Services werden in eigenen Prozessen ausgeführt. Wolff ordnet dem Konzept keine genaue Definition zu, jedoch verschiedene Eigenschaften und Kriterien die einen Microservice von einem klassischen Deployment Monolithen abgrenzen. Im Vordergrund stehen die drei Eigenschaften, dass ein Microservice stets nur eine Aufgabe erfüllt, mit anderen Microservices kommuniziert und eine universelle Schnittstelle zur Verfügung stellt, um eine sprachunabhängige Kommunikation zu ermöglichen. [81, p. 3f]

Abbildung 2.1 beschreibt die wichtigsten Eigenschaften des Konzepts Microservices. Insbesondere bei großen Systemen die sich entsprechend gut aufteilen lassen, hat eine solche Architektur erheblichen Einfluss auf die Organisationsstruktur innerhalb eines Entwicklungsprojektes.

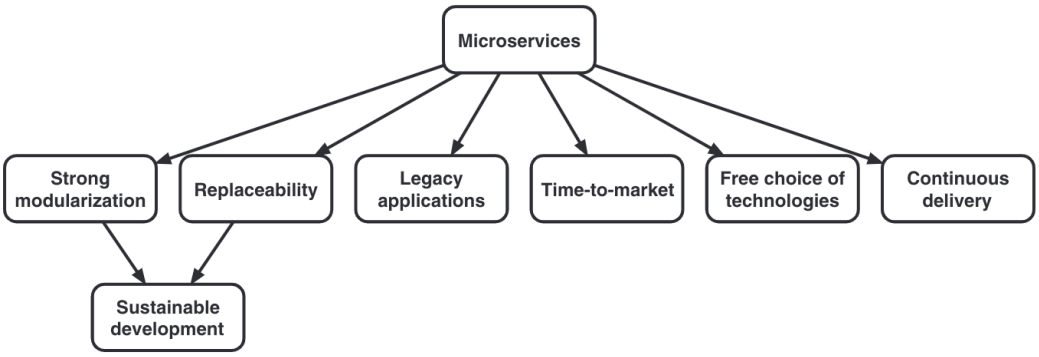


Abbildung 2.1: Eigenschaften von Microservices [81, p. 5]

Starke Modularisierung Klassische monolithische Architekturen bringen oft ungewollte Abhängigkeiten mit sich. Durch die Aufteilung in kleinere Elemente, repräsentiert durch Microservices und deren Kommunikation über explizite Interfaces [1] besteht eine größere technische Hürde zur Kommunikation mit einem Service. Dadurch besteht eine wesentlich geringere Wahrscheinlichkeit ungewollte Abhängigkeiten einzubringen. [81, p. 5]

Ersetzbarkeit Ein entscheidender Vorteil ist, dass einzelne Microservices sich leicht ersetzen lassen. Ein neuer Service muss lediglich das Kommunikationsinterface beibehalten. Eine bestehende Codebasis kann so komplett ersetzt werden, oder es kann sogar eine völlig andere Technologie verwendet werden. Im Hinblick auf mögliche Fehlentscheidungen in der Implementierung ist die dadurch entstehende Technical Debt vergleichsweise gering. Bei Bedarf kann ein Service mit geringeren Kosten als in einem modularisierten Monolithen neu entwickelt werden. [81, p. 5f]

Nachhaltige Entwicklung Modularisierung und Ersetzbarkeit ermöglichen eine nachhaltige Entwicklung. Je größer und länger ein monolithisches System entwickelt wird desto mehr *erodiert* eine Architektur und wird mit zunehmender Zeit unübersichtlich. Dem wird mit einzelnen Services entgegengewirkt, so dass diese in der Regel eine starke Abgrenzung zueinander besitzen. Neuere, besser geeignete Technologien können so für einzelne Services eher adaptiert werden als für einzelne Module in einem monolithischen System. [81, p. 6]

Legacy-Applikationen Legacy Applikation nutzen oft veraltete Technologie, die Erweiterung um neue Funktionalität ist meist kostspielig und aufwändig. Microservices können neue Funktionalität bereitstellen, ohne dass aufwändige Anpassungen am Legacy-System vorgenommen werden müssen. [81, p. 6]

Time-to-Market Durch die lose Kopplung zwischen Microservices können diese durch unabhängige Deployments in ein bestehendes System eingeführt werden. Teams können unabhängig voneinander parallel an Funktionalitäten arbeiten, so dass Overhead durch Koordination zwischen Teams entfällt und eine schnellere Entwicklung möglich ist. Gerade im agilen Projektmanagement ist dieses Vorgehen von Vorteil. Große Teams werden in kleinere voneinander unabhängige Teams aufgespalten. [81, p. 6f]

Technologieunabhängigkeit Die Adaption von neuen Technologien oder Versionen bereits verwendeter Technologie bringt in der Regel Risiken mit sich. Diese können durch Verwendung von Microservices wesentlich reduziert werden, da diese in einer geschlossenen Umgebung getestet werden können, ohne Einfluss auf andere Services zu nehmen. [81, p. 7]

Continous Delivery Microservices erlauben aufgrund ihrer Größe und Unabhängigkeit eine einfache Integration in bestehende *Continous Delivery*-Pipelines. Im Gegensatz zu einer monolithischen Applikation wird weniger Overhead in Ressourcenverbrauch und Konfiguration verursacht.[81, p. 7f]

[1] Wie beispielsweise bei REST-Schnittstellen oder Message-Brokern

Je nach Komplexität und Größe einer Microservicearchitektur, bringt diese durchaus Herausforderungen mit sich, die zu bewältigen sind. Eine solche Architektur definiert sich im Wesentlichen durch Abhängigkeiten zwischen einzelnen Services. Somit ist nicht inhärent ersichtlich wie die Kommunikationsstruktur der Services untereinander umgesetzt wurde. Um mit solchen *versteckten Abhängigkeiten* umzugehen ist eine genaue Absprache in einem Entwicklungsteam, sowie eine ausführliche Dokumentation essentieller Bestandteil. In modernen Anwendungen ändern sich Anforderungen, was gegebenenfalls ein *Refactoring* einer Anwendung erfordert. Aus diesem Grund sollte beim Entwurf einer Microservice Architektur, schon zu Beginn der Entwicklungsphase die Granularität der Modularisierung mit Bedacht gewählt werden. Ist diese zu fein gewählt ist es möglicherweise schwierig diese zu ändern. Dem kann jedoch durch kontinuierliche Anpassung der Granularität von grob zu fein, entgegengewirkt werden. [81, p. 8]

Im Unternehmenskontext spiegelt die Modularisierung in den meisten Fällen eine Aufteilung in verschiedene Geschäftsbereiche wider und dementsprechend auch die Aufteilung von zuständigen Teams. Entstehen hier Probleme ist es unter Umständen schwer diesen entgegenzuwirken, da Entscheidungen sich direkt auf das Unternehmen auswirken.

Das *Deployment* von Microservices geht mit einem hohen Grad an Automatisierung einher. Zur Ausführung ist eine entsprechende Infrastruktur nötig, welche wiederum den Komplexitätsgrad erhöht solche Services zu deployen, auszuführen und zu überwachen.

Mit zunehmender Zahl an beteiligten Microservices einer Architektur steigt auch die *Komplexität* eines solchen verteilten Systems. Es müssen viele potentielle Fehlerquellen beachtet werden, die in einer monolithischen Applikation nicht existieren. Services kommunizieren über Netzwerkprotokolle. Diese Kommunikation kann fehlschlagen und ist ebenso deutlich langsamer als die Kommunikation zwischen Prozessen innerhalb eines einzelnen System. [81, p. 8]

2.1.2 CAP Theorem

Das von Dr. Eric Brewer im Rahmen des *Symposium on Principles of Distributed Systems* im Jahr 1999 vorgestellte CAP-Theorem besagt, dass verteilte Systeme mit gemeinsam genutzten Daten, wie in Abbildung 2.2 dargestellt, höchstens zwei der drei Eigenschaften, Konsistenz (Consistency), Verfügbarkeit (Availability) und Partitionstoleranz (Partition Tolerance), erfüllen können.[4]

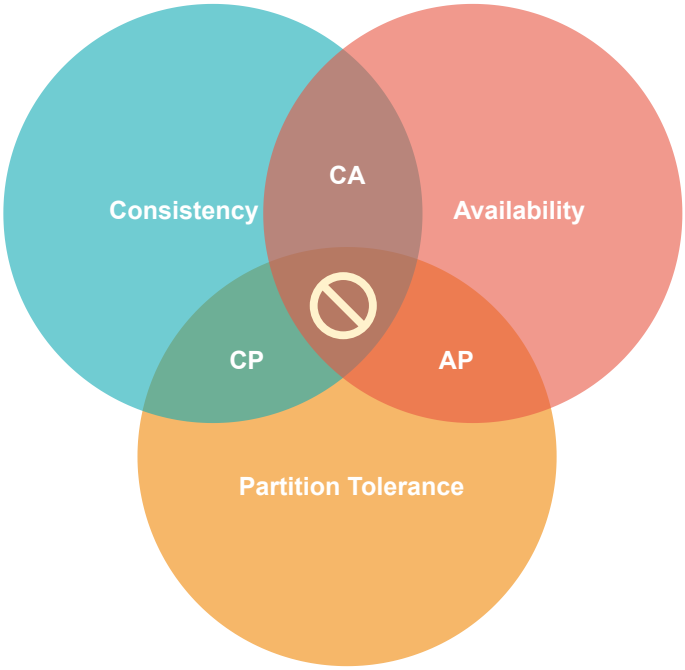


Abbildung 2.2: Visuelle Darstellung des CAP-Theorems

Konsistenz bedeutet in diesem Kontext, dass in einem verteilten System das Daten über mehrere Knoten repliziert, es zu jedem beliebigen Zeitpunkt eine Version eines gespeicherten Datums gibt, die dem aktuellsten Zustand im ganzen System entspricht. Ein externer Betrachter sieht also zu jedem Zeitpunkt, unabhängig des Knotens auf den zugegriffen wird, die aktuellste Version eines Datums.

Weiterhin gibt es für den Begriff der Konsistenz Abstufungen. Angefangen bei der *strikten* oder *starken Konsistenz*, welche sich mit dem bereits beschriebenen Sachverhalt deckt, dass zu jedem Zeitpunkt immer der aktuellste Wert gelesen wird, bis hin zur *schwachen Konsistenz*, welche keine Garantie gibt, dass Anfragen den aktuellsten Wert zurückgeben. Eine Sonderform der schwachen Konsistenz ist die *Eventual Consistency*. Diese Form findet hauptsächlich in verteilten Systemen Anwendung. Hierbei wird aus Performancegründen bei Schreiboperationen darauf verzichtet Updates unmittelbar auf alle Repliken zu verteilen. Nach Beendigung von Schreiboperationen werden Algorithmen verwendet, die ohne Zusicherung eines Zeitfensters dafür zuständig sind, die manipulierten Daten über alle Repliken zu synchronisieren. Der Zustand aller Knoten konvergiert in einem unbestimmten Zeitraum zu einem konsistenten Zustand.

Ebenso gilt zu beachten, dass sich der Begriff Konsistenz im Sinne des CAP-Theorems von dem Begriff Konsistenz im Kontext des ACID-Prinzips bei relationalen Datenbanken unterscheidet. ACID-Konsistenz bezieht sich stets auf Daten eines einzelnen Knotens. Alle Datenbankverbindungen beziehen stets die aktuellste Version eines Datums von einem einzelnen Knoten. [4]

Verfügbarkeit bezieht sich auf die Fähigkeit einer Anwendung in einem angemessenen Zeitraum eine Antwort zu liefern. Dabei sollen mehrere Anfragen unabhängig voneinander beantwortet werden, so dass ein entsprechendes Datenhaltungssystem während der Bearbeitung einer einzelnen Anfrage möglichst nicht blockiert.[4]

Ausfalltoleranz legt die Eigenschaft eines verteilten Datenhaltungssystem fest mit dem Ausfall einzelner oder mehrerer Knoten in einem Cluster umzugehen. In diesem Szenario soll ein System weiterhin stabil arbeiten und eingehende Anfragen weiterhin beantworten.[4]

2.1.2.1 Kritik am CAP-Theorem

Brewer äußert mittlerweile selbst Kritik an der Interpretation des Theorems. Die Einteilung in Systeme, die jeweils immer zwei der genannten Eigenschaften erfüllen[2], ist nicht unbedingt sinnvoll, da in der klassischen Interpretation des CAP-Theorems die Latenz einer Anwendung im Umgang mit einer Partitionierung nicht berücksichtigt wird. Im Falle einer Partition ist ein Teil eines Systems nicht mehr verfügbar, also nicht mehr ansprechbar. Entscheidend ist an dieser Stelle, wie ein System mit einer solchen Partitionierung umgeht. An diesem Punkt muss eine zeitgebundene Entscheidung getroffen werden und zwar ob eine Operation auf Kosten der Konsistenz durchgeführt wird oder auf Kosten der Verfügbarkeit abgebrochen wird. Da es sich bei der Partitionierung eines Systems in der Praxis um einen Fall handelt der eher selten Auftritt, definieren sich Systeme eher dadurch, wie sie mit einer Partitionierung umgehen, um entweder Konsistenz oder Verfügbarkeit zu gewährleisten. [4]

2.1.3 Leader Election

Leader-Election bezeichnet im Kontext von verteilten Systemen einen Algorithmus, um unter mehreren eindeutig identifizierbaren Prozessen $p_1, ..., p_n$ einen Prozess p_i zu bestimmen, der kritische Progammabschnitte, im Kontext einer verteilten Anwendung betreten darf. Dabei muss ein allgemeiner Konsens unter den Prozessen bezüglich des Gewinners einer Wahl bestehen. Generell gilt, dass ein einzelner Prozess nicht mehr als eine Wahl initiieren darf. Jedoch sind mehrere gleichzeitig ablaufende, von mehreren Prozessen gestartete Wahlen möglich. Im Rahmen eines Netzwerks, in welchem die Bestimmung einer Leader-Rolle notwendig ist, hat ein Prozess p_i abhängig davon, ob dieser zu einem beliebigen Zeitpunkt an einer Wahl beteiligt oder nicht beteiligt ist, entweder den Status *Teilnehmer (T)* oder *Nicht-Teilnehmer (NT)* inne. Eine häufig verwendete Prämisse, ist die Bestimmung eines Gewinners anhand eines größten Identifikationsmerkmals, unter der Voraussetzung, dass dieses einer totalen Ordnung unterliegt. Abhängig von der Topologie eines vorliegenden Netzwerks existieren unterschiedliche Ansätze zur Bestimmung eines Gewinners. Diese unterscheiden sich im Wesentlichen dadurch, ob ein Knoten in einem Netzwerk Kenntnis über andere Knoten oder benachbarte Knoten besitzt und mit diesen kommunizieren kann.[7, p. 657f]

[2] Solche Systemen betiteln sich meist selbst als CA, CP oder AP Systeme.

2.1.3.1 Ringbasierter Election-Algorithmus nach Chang und Roberts

Algorithm 1 Ringbasierter Election-Algorithmus

for $i = 1, 2, \ldots, N$ **do**
  $state_i = NT$ Markiere Prozess p_i als NT
end for

Wahl einleiten durch senden einer *election*-Nachricht:
$state_i = T$ Markiere Prozess als T
Sende Election-Nachricht mit Identifier $Id(i)$ an nächsten Nachbar $p_{(i+1) mod N}$

Empfangen einer Election-Nachricht:
$state_i = T$ Markiere Prozess p_i als T
if $Id(i) < Id(i-1)$ **then**
  Leite Nachricht an Nachbar weiter
else if $Id(i) \geq Id(i-1)$ AND p_i *is* T **then**
  Sende eigenen Identifier mit *election*-Nachricht an nächsten Nachbarn
else if $Id(i) = Id(i-1)$ **then**
  p_i hat die Wahl gewonnen.
  $state_i = NT$ Markiere Gewinner p_i als NT
  Sende Identifier mit *elected*-Nachricht an nächsten Nachbarn.
end if

Empfangen einer Elected-Nachricht:
$state_i = NT$ Markiere Prozess p_i als NT
Vermerke Identifier des Gewinners in $elected_i$
if p_i is not Winner **then**
  Leite Nachricht mit Identifier des Gewinners an Nachbar weiter
end if

Der in Listing 1 dargestellte Algorithmus von Chang und Roberts setzt eine logische, ringförmige Anordnung von Knoten eines Netzwerkes voraus. Existieren in einem solchen Netzwerk N Knoten, ist jeder Knoten k_i mit seinem nächsten Nachbarn $k_{(i+1) mod N}$ durch einen Kommunikationskanal verbunden. Nachrichten werden nach einem asynchronen Kommunikationsschema im Uhrzeigersinn versendet. Zur Kommunikation existieren lediglich die zwei Nachrichtentypen *elected* und *election*. Im Kern basiert der Algorithmus darauf, den Knoten, unter der Annahme einer totalen Ordnung, mit dem größten Identifizierungsmerkmal als Gewinner zu bestimmen. Eine Wahl wird durch eine *election*-Nachricht eingeleitet und führt dazu, dass alle Knoten die eine solche Nachricht erhalten das eigene Identifizierungsmerkmal mit dem Empfangenen vergleichen. Ist das empfangene Merkmal größer, wird die Nachricht mit diesem an den nächsten Knoten weitergeleitet. Im Fall, dass das empfangene Merkmal kleiner ist, wird das Merkmal in der erhaltenen Nachricht mit dem Eigenen ersetzt und weitergeleitet.

Erhält ein Knoten sein eigenes Identifizierungsmerkmal in einer *election*-Nachricht ist dieser der Gewinner der Wahl und kommuniziert dies an alle anderen Knoten mittels einer *elected*-Nachricht, welche das Identifizierungsmerkmal des Knotens enthält. Alle Knoten die eine *election*-Nachricht erhalten, speichern das Identifizierungsmerkmals des Gewinners, leiten die Nachricht weiter und wechseln in den Status *Nicht-Teilnehmer*. Zu-

sammenfassend ist der Algorithmus nur verwendbar, wenn bestimmte Voraussetzungen erfüllt sind. Es besteht eine Abhängigkeit zur vorliegenden Netzwerktopologie, sowie der Annahme, dass Nachrichten zuverlässig übertragen werden. In der Praxis eignet sich der Algorithmus auch nur bedingt, da er keine Ausfälle von Knoten toleriert. [7, p. 658f]

2.1.3.2 Bully-Algorithmus nach Garcia-Molina

Algorithm 2 Bully Algorithmus

while coordinatorIsAlive == true **do**
  if Ist der Koordinator Prozess erreichbar? **then**
    Continue
  else
    coordinatorIsAlive = false
  end if
end while

Sende *election*-Nachricht an alle Prozesse mit größerem Identifier
Warte Zeitraum T
if *answer*-Nachricht erhalten **then**
  Warte Zeitraum T'
  if *coordinator*-Nachricht erhalten **then**
    Setze $elected_i$ auf Identifier des Koordinators
  else
    Starte neue Wahl
  end if
else
  Prozess p_i wird Koordinator; Sendet Identifier an Prozesse mit niedrigerem Identifier
end if
if Keine Antwort in Zeitraum T **then**
  if Keine Antwort in Zeitraum T' **then**
  end if
end if

Erhalten einer *coordinator*-Nachricht:
Setze $elected_i$ auf Identifier des Koordinators

Erhalten einer *election*-Nachricht:
Prozess sendet eine *answer*-Nachricht und starte eine neue Wahl, insofern er noch keine begonnen hat.

Im Gegensatz zu dem vorgestellten ringbasierten Algorithmus, setzt der in Listing 2 dargestellten Bully-Algorithmus eine Topologie voraus in der jeder Knoten Kenntnis über alle unter- und übergeordneten Knoten besitzt. Eine Ordnung wird hier ebenfalls durch ein eindeutiges Identifizierungsmerkmal eines Knotens erzwungen. Die Kommunikation erfolgt auf Basis von *answer*-Nachrichten, *election*-Nachrichten und *coordinator*-Nachrichten. Erhält ein Knoten eine *coordinator*-Nachricht, vermerkt diese durch setzen einer Variable

das Identifizierungsmerkmal des Knotens und behandelt diesen als neuen Koordinator. Beim Erhalt einer *election*-Nachricht sendet ein Knoten eine *answer*-Nachricht zurück und startet eine neue Wahl.

Initial stellt sich der Knoten mit dem größten Identifizierungsmerkmal als Koordinator, durch Senden einer *coordinator*-Nachricht vor. Alle anderen Knoten prüfen anhand eines festgelegten Zeitraums T, ob ein Koordinator noch erreichbar ist. Ist dieser nicht mehr erreichbar beginnt ein Knoten eine neue Wahl, indem er eine *election*-Nachricht an alle Knoten mit kleineren Identifizierungsmerkmalen sendet. Erhält ein Knoten keine Antwort, wird dieser zum neuen Koordinator und übermittelt diese durch Senden einer *coordinator*-Nachricht. Wird hingegen eine *answer*-Nachricht empfangen, wird ein weiterer Zeitraum T' auf eine Koordinatornachricht gewartet. Empfängt der Knoten keine Information über einen neuen Koordinator startet er erneut eine Wahl. Grundlegend lässt sich das Prinzip des Bully-Algorithmus dadurch zusammenfassen, dass Knoten, welche das größte Identifizierungsmerkmal besitzen, alle untergeordneten Knoten bei der Wahl zum Koordinator verdrängen. Der Algorithmus kann durch Verwendung von Timeouts mit dem Ausfall von einzelnen Knoten umgehen, ist jedoch auf Vorwissen bezüglich der Ordnung und Kommunikation, sowie eine synchrone Kommunikation angewiesen. [7, p. 660ff]

2.1.4 Virtualisierung

Unter Virtualisierung von Hard- oder Software versteht man die Nachbildung von Computersystemen. Ein solches virtuelles Computersystem wird gemeinhin als virtuelle Maschine bezeichnet. Diese virtuelle Umgebung nutzt Ressourcen wie CPU, RAM, Netzwerkschnittstelle und Speicher vom Host-System auf dem sie erstellt wurde. Dabei separiert eine Software, der Hypervisor, die Hardwareressourcen vom Hostsystem und erstellt einen Ressourcenpool aus dem diese dynamisch an virtuelle Maschinen verteilt werden können. Durch diese dynamische Provision von Ressourcen können beispielsweise verschiedene Betriebsysteme in mehreren virtuellen Maschinen gleichzeitig auf einem physischen Hostsystem ausgeführt werden. [46]

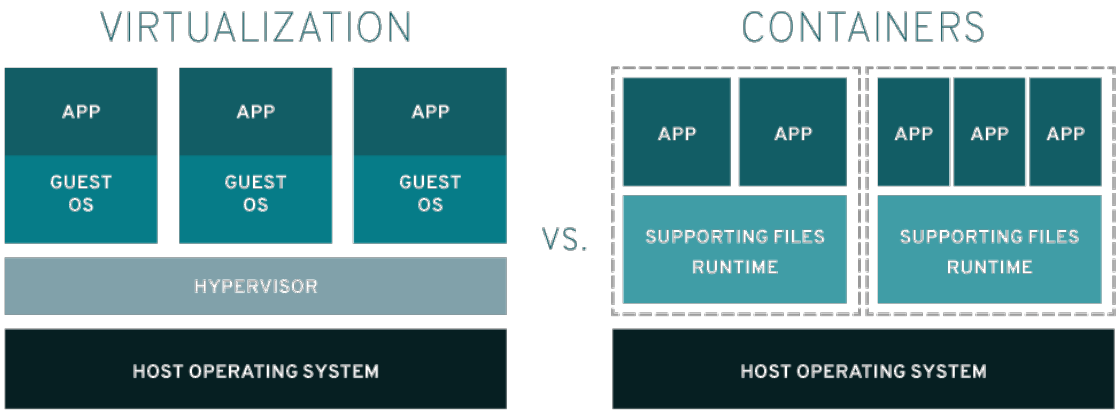


Abbildung 2.3: Differenzierung - Virtuelle Maschine und Container [46]

2.1.4.1 Container

Eine leichtgewichtigere Alternative zu virtuellen Maschinen bietet das Container Prinzip. In ihren Grundzügen sind Container mit virtuellen Maschinen vergleichbar, indem sie von der unterliegenden physischen Hardware abstrahieren und eine isolierte Ausführung einer Anwendung ermöglichen. Im Gegensatz zu einer klassischen virtuellen Maschine teilen sich mehrere Container den Kernel des unterliegenden Systems, es wird kein Hypervisor zur Provisionierung von Ressourcen benötigt. Sie kapseln lediglich die Anwendung und ihre Abhängigkeiten (siehe Abbildung 2.3), können dennoch unterschiedliche Betriebssysteme ausführen, insofern diese kompatibel mit dem unterliegenden Kernel sind. Durch die geringe Größe von Containern sind diese in ihrer Ausführung und Startzeit bedeutend schneller als eine virtuelle Maschine. Um Container auszuführen wird eine entsprechende Laufzeitumgebung wie Docker benötigt. [61, p.241f]

2.1.4.2 Container Image

Ein Container-Image stellt gewissermaßen den Bauplan eines Containers dar. Es definiert die Anwendung und ihre Abhängigkeiten die in einem Container ausgeführt werden sollen. Für jeden Container existiert ein Image als Grundlage. Container sind unveränderliche Objekte. Soll der Code eines Containers geändert werden muss ein neues Image mit den enstprechenden Änderungen gebaut werden. [19]

2.1.4.3 Docker

Bei Docker handelt es sich um eine Laufzeitumgebung zur Ausführung von Containern. Docker ist auf nahezu allen gängigen Betriebssystemen verfügbar. Für einen einfachen Einstieg wurde die Anwendung Docker-Desktop entwickelt, welche die Docker-Laufzeitumgebung mit einem nutzerfreundlichen User-Interface ausstattet, dass die einfache Verwaltung, von Images und Containern ermöglicht. Weiterhin ist es möglich erstellte Images über das öffentliche *Docker Hub-Repository* öffentlich verfügbar zu machen. [17]

2.1.5 Representational State Transfer

Erstmals vorgestellt von Roy Fielding im Rahmen seiner Dissertation beschreibt Representational State Transfer (REST) einen Architekturstil für Netzwerk basierte Systeme. Fielding definiert das Representational State Transfer (REST)-Paradigma anhand verschiedener Restriktionen die einem verteilten System auferlegt werden. Durch Anwendung des Client-Server-Architekturstils wird das User-Interface von der Server-Komponente entkoppelt. Die so in ihrer Komplexität reduzierte Server-Anwendung lässt sich leichter skalieren und Client und Server können unabhängig voneinander weiterentwickelt werden. Einzelne Interaktionen zwischen Client und Server sollen komplett unabhängig voneinander und die Kommunikation zustandslos erfolgen. Um eine solche Kommunikation zu realisieren, muss eine Nachricht alle notwendigen Informationen enthalten, damit die Server-Anwendung diese interpretieren und verarbeiten kann. Server-Zustand und Client-Zustand sind entsprechend komplett voneinander entkoppelt, eine Nachricht darf sich auf keinen auf Serverseite gespeicherten Kontext beziehen. Durch diese Entkopplung und die zustandslose Natur der Interaktion liegt der Session-State komplett auf Client-Seite.

Zur Steigerung der Effizienz und der von Benutzern wahrgenommenen Leistung führt Fielding die Nutzung von Caches ein. Antworten des Servers stellen hierbei einen Indikator zur Verfügung ob eine Antwort gecached werden kann. Da REST über einfache HTTP-Requests kommuniziert, können auf Client-Seite entsprechend einfache Web-Caches integriert werden, um den Caching-Mechanismus zu implementieren. Bei der Entwicklung eines, aus mehreren Komponenten bestehenden Systems ist es von Vorteil, wenn diese unabhängig voneinander entwickelt werden können. Aus diesem Grund stellt Fielding diverse Anforderungen an ein solches Interface. Informationen werden mittels eines Uniform Resource Identifier (URI) als Ressourcen identifiziert. Diese können wiederum manipuliert werden. Nachrichten sind selbst beschreibend im Sinne, dass diese Relationen zu anderen Ressourcen beinhalten. Diese Relationen werden durch Hyperlinks in einer Antwort dargestellt. Diese Modellierung von Relationen beschreibt das Prinzip Hypermedia as the Engine of Application State (HATEOAS). Die Darstellung dieser Links wird wiederum durch die *Hypertext Application Language (HAL)* definiert. Zur weiteren Förderung der Unabhängigkeit von Systemkomponenten, ist eine REST-Architektur mehrschichtig aufgebaut. Einzelne Komponenten sind in sich geschlossen und kommunizieren über die beschriebenen universellen Schnittstellen miteinander. [40]

Dieses von Fielding ausführlich beschriebene Architektur Paradigma ist in der modernen Entwicklung von Microservices weit verbreitet. Applikationen bedienen sich jedoch meist nur zu Teilen dieser Definition. Um den Grad der Adaption des REST-Prinzips zu bestimmen, führt Richardson das *Richardson Maturity Modell* ein. Das Modell definiert drei Stufen um den Reifegrad einer Applikation zu bestimmen.[42]

Stufe 1 - Einführung von Ressourcen
Die erste Stufe führt das Konzept von Ressourcen ein. Eine Ressource repräsentiert eine Information, die eindeutig über einen URI identifiziert werden kann. Dabei handelt es sich lediglich um Links die auf eine Ressource zeigen.

Stufe 2 - Einführung von HTTP Verben
Um die Ausführungen von Operationen, die eine Ressource manipulieren oder anfordern zu vereinheitlichen, wird in der zweiten Stufe ein fixes Vokabular eingeführt, mit dessen Hilfe der Zweck einer Anfrage identifiziert werden kann. Tabelle 2.1 listet alle verfügbaren HTTP Verben und deren Funktion auf.

HTTP Verb	Funktion
GET	Die Ressource wird angefordert. Es findet keine Manipulation der Ressource statt.
POST	Eine neue Ressource wird angelegt.
PUT	Eine bestehende Ressource wird ersetzt.
PATCH	Einzelne Attribute einer bestehenden Ressource werden manipuliert.
DELETE	Löscht eine bestehende Ressource.

Tabelle 2.1: Funktion der verschiedenen HTTP Verben

Stufe 3 - Einführung von HATEOAS
Mit der letzten Stufe, der Einführung von HATEOAS, entspricht eine REST-Architektur der von Fielding beschriebenen Definition. Eine Antwort enthält Informationen zu verwandten Ressourcen in Form von Links zu diesen. Durch dieses Konzept erhält eine Schnittstelle einen selbst dokumentierenden Charakter, die API wird erforschbar. Ein

Client benötigt lediglich einen Eintrittspunkt. Ist ein Client umfänglich HATEOAS kompatibel, kann das URI-Schema einer Schnittstelle geändert werden ohne, dass die Kompatibilität zu zuvor kompatiblen Clients beeinflusst wird.

2.1.6 GraphQL

Bei GraphQL handelt es sich um eine von Facebook entwickelte Abfragesprache, sowie eine serverseitige Laufzeitumgebung. Diese ermöglicht eine Entwicklung von plattformunabhängigen, universellen Schnittstellen. Das primäre Ziel der GraphQL-Abfragesprache ist die Bereitstellung von ausschließlich relevanten Informationen und damit einhergehende Steigerung der Performanz einer Anwendung. Dies wird durch eine genaue Spezifikation der benötigten Informationen erzielt, so dass eine Serverantwort möglichst keine redundanten Informationen enthält. Im Gegensatz zu klassischen REST-Anwendungen werden GraphQL-Anfragen über einen einzelnen zentralen Endpunkt auf Serverseite abgehandelt. citeredhat:graphql

2.1.6.1 Kernkomponenten[72, ch. 2]

Das *Document* bildetet den Kern der GraphQL-Spezifikation. Es beschreibt den Inhalt einer Anfrage an einen GraphQL-Server und definiert eine oder mehrere Operationen und Fragmente. *Operationen* sind wesentlicher Bestandteil einer GraphQL-Abfrage. Unterschieden wird zwischen drei verschiedenen Typen. *Queries* sind einfache Leseoperationen um Daten abzufragen. *Mutations* sind Schreiboperationen um neue Daten oder Änderungen an bestehenden Daten zu persistieren. Eine Besonderheit dabei ist, dass der Schreiboperation eine Leseoperation folgt. Dementsprechend enthält die Antwort einer Anfrage stets die von einer Mutation aktualisierten Daten. Der dritte Operationstyp wird als *Subscription* bezeichnet. Hierbei wird das klassische *Publish/Subscribe-Pattern* umgesetzt. Ein GraphQL-Client, in diesem Szenario der *Subscriber* , spezifiziert über ein Document an welchen Daten er interessiert ist. Analog fungiert ein GraphQL-Server als *Publisher*, der aktualisierte Daten unmittelbar an einen Subscriber weiterleitet.

```
{
    getPersonById(id: 42) { # Basic query operation
        id
        name
        age
        height(unit: CENTIMETER) # Field argument
        weight
        address { # Nested selection set
            streetname
            housenumber
            city
            zipcode
        }
    }
}
```

Listing 2.1: Beispiel einer GraphQL-Operation

Listing 2.1 beschreibt eine einfache Query-Operation um Daten über eine Person abzurufen. Eine solche Abfrage besteht aus Feldern (Fields), die alle gewünschten Attribute einer Entität spezifizieren und Argumenten (Arguments), die sich serverseitig darauf auswirken wie ein Feld aufgelöst wird. Die Menge an Feldern die in einer Operation die gewünschten Informationen spezifizieren, wird als *SelectionSet* bezeichnet. *SelectionSets* erlauben eine beliebige Schachtelung in einer Anfrage. So können beispielsweise Relationen in einer Anfrage angegeben werden.

```
{
    query FetchPersonsWithName($name: String) { // Query operation
        getPersonByName(name: $name){
            ...basicPersonFields    // Fragment usage
        }
    }
    mutation CreatePerson($newPerson: Person) { // Mutation operation
        createPerson(person: $person){
            ...basicPersonFields    // Fragment usage
        }
    }
    fragment basicPersonFields on Person {  // Fragment definition
        id
        name
        age
    }
}
```

Listing 2.2: Beispiel einer komplexeren GraphQL-Anfrage

Eine einfache Query-Operation wie in Listing 2.1 dargestellt, benötigt keine Angabe welche Art von Operation ausgeführt werden soll. Wird jedoch eine andere oder mehrere Operationen in einer Anfrage ausgeführt, müssen diese für die GraphQL-Server-Applikation eindeutig identifizierbar sein. Dies geschieht durch die Angabe des Operationstyps, welcher den Werten *query*, *mutation* oder *subscription* entsprechen kann. Auf den Operationstyp folgt der Name einer Operation, um diese eindeutig zu identifizieren. Eine solche Operation kann noch weitere optionale *Variablen-Definitionen* enthalten, welche zur Parametrisierung einer Operation dienen. Gerade bei komplexeren Operationen dienen *Fragmente (Fragments)* zur Vermeidung von redundanten Angaben in einer GraphQL-Anfrage

Ein Fragment definiert eine Menge von Feldern einer Entität, die in einer Abfrage häufig benötigt werden. Ein Fragment kann mithilfe des *Spread-Operator*, wie in Listing 2.2 zu sehen verwendet werden um die benötigten Felder in einer Abfrage anzugeben.

```
{
    query DirectiveExample($skipAge: Boolean!, $includeName: Boolean!) {
        person{
            id
            age @skip (if: $skipAge)
            name @include (if: $includeName)
        }
    }
}
```

Listing 2.3: Verwendung von Direktiven

Manche Operationen erfordern zusätzliche Funktionalität, um zu steuern welche Ergebnisse eine Serverantwort zurückliefern soll. Ein mögliches Szenario würde erfordern, dass Ergebnisse, die in einem bestimmten Feld einen bestimmten Wert annehmen, ausgelassen oder inkludiert werden. Diese Funktionalität ermöglichen *Direktiven*. Dabei handelt es sich um Annotationen die mit einzelnen Feldern verknüpft werden um eine gewisse Funktionalität, gemäß gewisser Bedingungen zu erzwingen. In Listing 2.3 werden die Direktiven *skip* und *include* eingeführt. Die *Skip-Direktive* exkludiert ein Ergebnis, insofern das annotierte Feld einen bestimmten Wert annimmt. Die gegensätzliche Funktionalität bietet die *Include-Direktive*. Hierbei ist ein Ergebnis nur in der Serverantwort enthalten, insofern das annotierte Feld einen bestimmten Wert annimmt. Die beiden im Beispiel vorgestellten Direktiven sind lediglich exemplarisch zu betrachten. Verfügbare Direktiven sind von der konkreten Implementierung einer Server Anwendung abhängig. Es existieren keine vorgegebenen Direktiven, die eine GraphQL-Server-Applikation implementieren muss.

2.1.6.2 Typensystem[72, ch. 3]

```
type Person {
    id: ID!
    name: String!
    age: Int
    height: Float
    weight: Float
    address: Address
}

type Address {
    streetname: String!
    housenumber: Int!
    city: String!
    zipcode: String!
}

type Query {
    getAllPersons(): [Person]
    getPersonByName(name: String!): [Person]
    getPersonById(id: ID!): Person
}

type Mutation {
    createPerson(person: Person!): Person
    deletePerson(id: ID!): Person
}
```

Listing 2.4: Exemplarische Darstellung eines GraphQL-Schemas

Der Kern einer GraphQL-Anwendung ist, wie in Listing 2.4 dargestellt, das durch ein Schema spezifizierte Typsystem. Ein GraphQL-Schema stellt im Grunde ein Mapping der Felder von Entitäten auf entsprechende Typen dar, sowie Definitionen bezüglich verfügbarer Operationen dar. Eine Server-Anwendung gleicht bei jeder Abfrage die Eingaben und bei jeder Antwort die Ausgaben mit dem Schema ab, um zu gewährleisten, dass nur die spezifizierten Typen akzeptiert werden. Das Schema dient als eine Art Vertrag zwischen einer GraphQL-Server-Applikation und allen Clients der Anwendung. Damit ein Client in der Lage ist mit einer GraphQL-Applikation zu kommunizieren, muss ein Server festgelegte introspektive Queries zur Verfügung stellen. Mit diesen ist ein Client in der Lage alle verfügbaren Operationen, sowie alle definierten Typen abzufragen. In der Regel bieten Server-Anwendungen einen Endpunkt an, um das gesamte Schema abzufragen.

2.1.6.3 Architektur einer GraphQL-Anwendung[72, ch. 6]

Zur Implementierung einer GraphQL-Server-Anwendung schreibt die GraphQL-Spezifikation nicht vor, wie eine konkrete Anwendungsarchitektur umgesetzt werden muss. Eine Anwendung muss in der Lage sein ein Schema zu prozessieren, welches in der Regel separat in einer eigenen Datei angegeben wird. Alternativ kann eine Anwendung ein Schema auch programmatisch definieren. In der GraphQL-Spezifikation beschreibt ein Resolver eine Funktion, welche eingehende Operationen verarbeiten und anhand des Schemas validieren kann. Dementsprechend stellt die Implementierung der Resolver-Funktion die zweite Komponente in einer GraphQL-Server Architektur dar. Die Verarbeitungsschritte einer GraphQL-Anfrage durch einen GraphQL-Server lassen sich durch die folgenden Schritte zusammenfassen:

1. *Document* wird eingelesen und in der Anfrage spezifizierte *Operationen* werden identifiziert.
2. Validierung der Anfrage gemäß des Schemas. Kann eine GraphQL-Anfrage nicht validiert werden antwortet der Server mit einer entsprechenden Fehlermeldung.
3. Wurde eine Anfrage erfolgreich validiert, können alle spezifizierten Operationen ausgeführt werden und eine Antwort mit allen geforderten Daten an den Client versendet werden.

Zusammenfassend benötigt eine Anwendung lediglich ein an einen Server gekoppeltes Schema, welches die Gestalt einer GraphQL-API festlegt und ein oder mehrere Resolver, die spezifizieren, wie eine bestimmte Operation auszuführen ist. GraphQL erhält über die letzten Jahre immer mehr Einzug in moderne Software. Bekannte Frameworks wie das Spring Framework bieten bereits Erweiterungen an, um eine GraphQL-Anwendung im Spring-Ökosystem zu entwickeln. Neben der Spring-Erweiterung existieren auch eigenständige, rein auf GraphQL spezialisierte Lösungen wie Apollo oder Hasura. Dabei handelt es sich um eigenständige GraphQL-Server, welche die GraphQL-Spezifikation vollumfänglich implementieren und um nützliche Funktionalitäten wie beispielsweise Caching erweitern.

2.1.7 MQTT - Protokoll

Das Message Queuing Telemetry Transport Protokoll ist ein in IoT-Anwendungen weit verbreitetes, leichtgewichtiges Nachrichtenprotokoll. Auf dem TCP/IP-Stack aufbauend

ist MQTT auf der Anwendungsschicht des OSI-Modells einzuordnen. Es existieren verschiedene Versionen der MQTT-Spezifikation. Unter anderem die zu diesem Zeitpunkt aktuellste Version 5.0, sowie die am weitesten verbreitete Version 3.1.1. Kompatible Client-Anwendungen können über MOM-Broker kommunizieren. MQTT ist speziell auf leichtgewichtige asynchrone *Machine to Machine (M2M)*-Kommunikation ausgelegt und verursacht dementsprechend einen sehr geringen Overhead in der Netzwerkkommunikation. [57, p. 5f] Nachrichten werden über Kanäle, sogenannte *Topics* verschickt. Die Benennung der Topics ist von kompatiblen Clients frei wählbar. Diese sind in der Lage über Topics Nachrichten zu senden (Publisher) oder Nachrichten zu empfangen (Subscriber). Generell werden Nachrichten per *Broadcast-Routing* an alle interessierten Subscriber gesendet.[57, p.19f] Eine Besonderheit der MQTT-Spezifikation sind die drei *Quality of Service (QoS)*-Stufen. Dabei handelt es sich um eine Vereinbarung zwischen Sender und Empfänger einer Nachricht, die die Garantie der Zustellung einer Nachricht definiert. [57, p .22]

2.1.7.1 Quality of Service Level 0 - At most once delivery

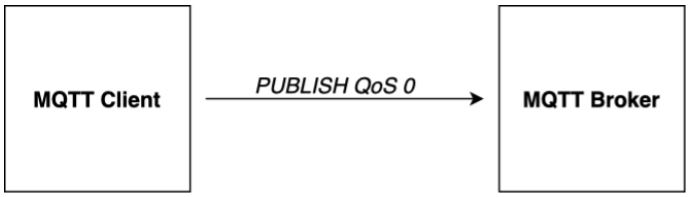


Abbildung 2.4: Versenden einer Nachricht mit QoS-Level 0

Die erste QoS-Stufe garantiert die selbe Zustellgarantie wie das unterliegende TCP-Protokoll. Eine Nachricht wird in Fire and Forget-Manier gesendet. Weder Empfänger noch Sender bestätigen die Zustellung der Nachricht. QoS 0 weist den geringsten Overhead auf, da keine Bestätigungsnachrichten ausgetauscht werden. [57, p. 22]

2.1.7.2 Quality of Service Level 1 - At least once delivery

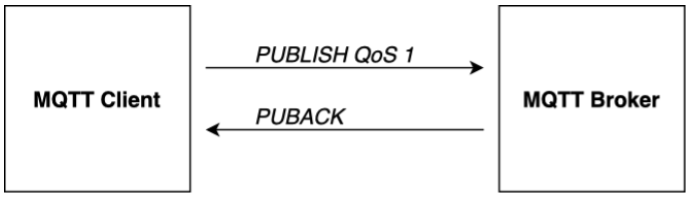


Abbildung 2.5: Versenden einer Nachricht mit QoS Level 1

Diese QoS-Stufe fordert von dem Ziel, welches die Nachricht erhalten soll, eine Bestätigung in Form eines PUBACK-Pakets. So kann sichergestellt werden, dass eine Nachricht mindestens einmal an einen Empfänger zugestellt wurde. Der Sender der Nachricht speichert diese temporär ab, bis er eine Bestätigung über den Erhalt der Nachricht empfängt. Erhält der Absender die Nachricht nicht innerhalb eines festgelegten Zeitraums, wird die gespeicherte Nachricht über ein entsprechendes Flag als Duplikat markiert und erneut gesendet. Der Nachteil dieser QoS-Stufe besteht darin, dass Nachrichten doppelt bei einem Empfänger ankommen können. In diesem Fall liegt es in der Verantwortung des Empfängers eine entsprechende Logik für den Umgang mit Duplikaten zu implementieren.[57, p. 23]

2.1.7.3 Quality of Service Level 2 - Exactly once delivery

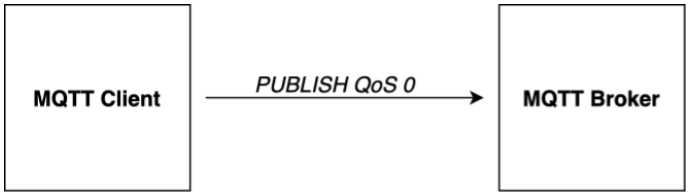


Abbildung 2.6: Versenden einer Nachricht mit QoS-Level 2

Die letzte QoS-Stufe garantiert, dass eine Nachricht genau einmal ankommt. Dazu findet ein Vier-Wege-Handshake statt. Zunächst wird die Nachricht an den Empfänger gesendet. Analog zur vorigen Stufe sendet der Absender eine Nachricht erneut, insofern er keine Bestätigung in einem gewissen Zeitraum erhält. Der Empfänger antwortet bei Erhalt der Nachricht mit einer Bestätigung in Form eines *PUBREC-Pakets*. Nach Erhalt des *PUBREC-Pakets* verwirft der Sender die ursprüngliche Nachricht und sendet ein *PUBREL-Paket* an den Empfänger. Jede Nachricht besitzt einen eindeutigen *Identifier (ID)*. Diese ist solange der Handshake läuft belegt, um zu vermeiden, dass eine Nachricht ein weiteres Mal prozessiert wird. Nach Erhalt des *PUBREL-Pakets* antwortet der Empfänger wiederum mit einem *PUBCOMP-Paket* um den Vorgang abzuschließen und gibt die zuvor gesperrte Nachrichten-ID wieder frei.[57, p. 23f]

Wichtig ist es an dieser Stelle anzumerken, dass die verschiedenen Quality of Service-Stufen immer nur eine Garantie zwischen Client und Broker darstellt. Zum einen zwischen sendendem Client und Broker, der die Nachricht empfängt und zum anderen zwischen Broker als Sender und Client als Empfänger. Wird beispielsweise vom Client QoS 2 gewählt, wird eine Nachricht vom Broker mit dieser Stufe empfangen. Hält ein Subscriber eine Subscription mit QoS 1 auf dem selben Topic, passt der Broker bei Weiterleitung die QoS-Stufe an die des Subscribers an.[57, p. 25f]

2.1.7.4 Retained Messages

Ein typisches Szenario das mehrere über MQTT kommunizierende Clients beinhaltet erfordert, dass neue Subscriber auf einem Topic Statusinformationen oder Konfigurationsinformationen bezüglich etwaigen sendenden Clients erhalten. Für diesen Anwendungsfall führt die MQTT-Spezifikation das Konzept von *Retained Messages* ein. Eine jede Nachricht bietet die Möglichkeit ein *retained*-Flag zu setzen. Ist dieses gesetzt, speichert der Broker die letzte auf einem Topic gesendete Nachricht ab und leitet diese an alle neu angemeldeten Subscriber weiter. Diese Nachricht stellt einen *last known good value* dar, was nicht bedeutet, dass es der letzte auf dem Topic gesendete Wert ist, sondern nur die letzte Nachricht auf diesem Topic deren Retained-Flag gesetzt wurde. [57, p. 29f]

2.1.7.5 Last Will and Testament

Das MQTT-Protokoll wird oft in unzuverlässigen Netzwerken genutzt. In diesen kann es zu unregelmäßigen Ausfällen von Client-Verbindungen kommen. Man spricht dann von ungewollten Verbindungsausfällen. Im Kontext der MQTT-Spezifikation wird eine beabsichtigte Unterbrechung oder Schließung einer Verbindung von Client-Seite mit dem Senden einer *DISCONNECT-Nachricht* bestätigt. Um zu unterscheiden, ob eine Client-Verbindung beabsichtigt oder unbeabsichtigt unterbrochen wurde, führt MQTT das Kon-

zept *Last Will and Testament (LWT)* ein. Für den Fall einer *ungewollten Unterbrechung* legt der Client bei Initialisierung einer Session eine *LWT-Nachricht* für einen Topic fest, die im Falle einer solchen Unterbrechung des Clients automatisch vom Broker auf diesem Topic veröffentlicht wird. [57, p. 31f]

2.1.7.6 Clean und Persistent Session

Verliert ein Client die Verbindung zu einem Broker, gehen bestehende Subscriptions verloren. Verbindet sich der Client erneut, muss dieser erneut alle Topics abonnieren. In diesem Fall bezeichnet man eine solche Session als *Clean Session*, der Broker speichert keine Informationen ab um dem Client wieder in seinen vorigen Zustand zu versetzen. Im Falle einer persistenten Session speichert der Broker alle Subscriptions, noch nicht bestätigte Nachrichten mit QoS 1 und QoS 2, sowie verpasste Nachrichten mit QoS 1 und QoS 2. Bei Wiederaufnahme der Verbindung kann der Client durch die vom Broker gespeicherten Informationen wieder in seinen ursprünglichen Zustand versetzt werden. Für jede Client-Verbindung kann ein entsprechendes Flag, das *Clean Session-Flag* gesetzt werden. Durch setzen dieses Flags wird eine Sesssion als nicht persistente Session gehandhabt. Andernfalls ist die Session persistent. [57, p. 27f]

2.1.7.7 Wildcard Funktionalität

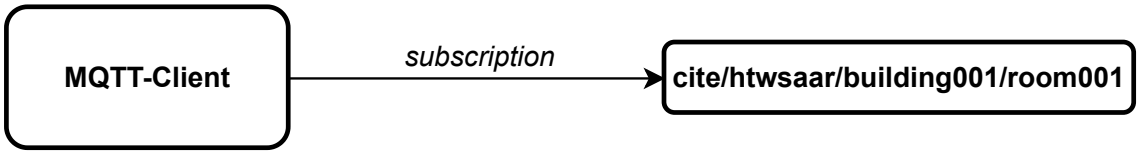


Abbildung 2.7: Subscription auf einen einzelnen Topic

Standardmäßig wird bei der Benennung von MQTT-Topics eine hierarchische Benennung verfolgt. Im Beispiel (siehe Abbildung 2.7) bildet die Topic-Struktur Gebäude mit ihren Räumen ab. Sowohl Gebäude als auch Räume sind aufsteigend nummeriert. Abbildung 2.7 zeigt eine Subscription auf einem einzelnen Topic. Eingehende Nachrichten werden nur auf diesem Topic empfangen. Ein häufiger Anwendungsfall bedingt jedoch mehrere Subscriptions auf mehrere Topics zugleich. [57, p. 19]

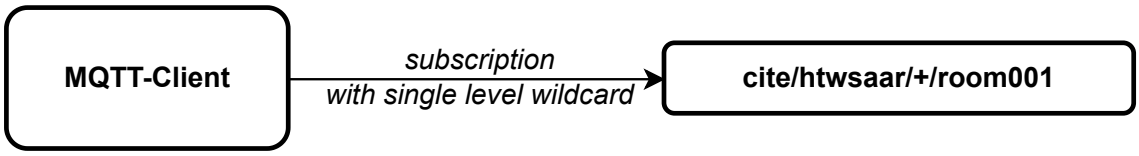


Abbildung 2.8: Subscription mit Single Level Wildcard

Mithilfe des *Single Level Wildcard* + kann ein einzelnes Level in der Topichierarchie ersetzt werden. Durch Angabe des *Single Level Wildcards* signalisiert ein Client, dass er an allen existierenden Topics, die diesem Pattern entsprechen, interessiert ist. Gemäß der oben genannten Analogie ist ein Client der eine Subscription wie in Abbildung 2.8 hält an allen Daten aus dem Raum 001 eines jeden Gebäudes interessiert. [57, p. 19f]

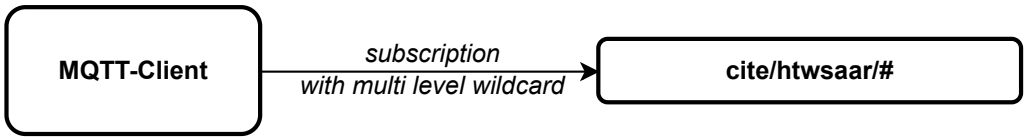


Abbildung 2.9: Subscription mit Multi Level Wildcard

Analog können durch Angabe von # im Topicnamen mehrere Hierachielevel ersetzt werden. Abbildung 2.9 zeigt die Subscription eines Clients der an Daten aus allen Räumen von allen Gebäuden interessiert ist. [57, p. 20]

2.2 Toolstack

2.2.1 Kubernetes

Kubernetes ist ein von Google entwickeltes System zur Verwaltung von Clustern und der Orchestrierung von Containern. Dabei bietet es für containerisierte Anwendungen umfassende Möglichkeiten zur Konfiguration und Automatisierung von Deployments. Im Kubernetes-Ökosystem können alle beteiligten Elemente als Objekte betrachtet werden. Diese werden verwendet um den Zustand des Clusters zu repräsentieren. Sie beschreiben, welche Anwendungen im Cluster ausgeführt werden, welche verfügbaren Ressourcen diese nutzen und geben Richtlinien, beispielsweise zur Skalierung oder zum Umgang mit Netzwerkpartitionierungen vor. Ein Objekt definiert den Workload und den gewünschten Zustand. Kubernetes versucht diesen entsprechend dauerhaft beizubehalten oder gegebenenfalls anzupassen. [27]

Pods

Ein Pod stellt die kleinste zu verwaltende Einheit eines Workloads in einem Kubernetes Cluster dar. Im Kontext eines Pods teilen sich ein oder mehrere Container die gleichen Speicher- und Netzwerkressourcen, sowie eine Konfiguration die definiert wie Container ausgeführt werden. [22]

Replica Set

Bei der Skalierung einer Anwendung in einem Cluster werden in der Regel mehrere Repliken eines Pods, in dem eine Anwendung läuft, erstellt. Um sicherzustellen, dass zu jedem Zeitpunkt immer eine feste Anzahl an Pods verfügbar ist, kann ein ReplicaSet verwendet werden. [23]

Deployment

Ein Deployment verwaltet ein oder mehrere Objekte des Typs ReplicaSet und somit den gesamten Lebenszyklus einer Anwendung. Im Gegensatz zum ReplicaSet ermöglicht ein Deployment ein automatisiertes und kontrolliertes Rollout von neuen Anwendungsversionen. [18]

Namespace

Namespaces bieten eine Möglichkeit zur Isolierung von Anwendungen innerhalb eines Clusters. Ressourcen innerhalb eines Namespaces müssen eine eindeutige Benennung aufweisen. Grundsätzlich kann der Scope für Objekte wie Deployments, ReplicaSet, Service durch Verwendung eines Namespaces begrenzt werden. Namespaces werden beispielsweise genutzt um eine bestehende Organisationsstruktur abzubilden und so die Entwicklung zu erleichtern, indem jedem Team ein eigener Namespace zugewiesen wird. [20]

Networking

Um eine Kommunikation innerhalb und außerhalb eines Clusters zu ermöglichen, verwaltet Kubernetes die komplette Netzwerkinfrastruktur. Einem jedem Pod wird eine eigene IP-Adresse zugeordnet, wobei entsprechende Ports von Containern und Pods getrennt voneinander zu betrachten sind. Das ermöglicht auch die Kommunikation zwischen einzelnen Containern innerhalb eines Pods. [21]

Service

Netzwerkanwendungen die in einem Cluster ausgeführt werden bieten oftmals Schnittstellen, über die beispielsweise eine Frontend Applikation kommunizieren kann. Das erfordert letztendlich jedoch, dass ein solcher Service nach außen hin verfügbar ist. Kubernetes teilt jedem Pod eine eigene IP-Adresse zu und fasst mehrere Pods der gleichen Anwendung unter einem DNS-Namen zusammen. Dadurch kann ein Objekt des Typs Service eingehende Anfragen an einen entsprechenden Service weiterleiten. Weiterhin ist ein Service-Objekt in der Lage eine Anwendung bereitzustellen. Kubernetes unterscheidet zwischen vier verschiedenen Typen von Service Objekten. Der Typ *ClusterIP* stellt den Service nur innerhalb des Clusters zur Verfügung. Ein Service kann außerhalb des Clusters bereitgestellt werden, indem der Typ *NodePort* genutzt wird. Dieser ist dann über jede Node auf einem konfigurierten statischen Port erreichbar. Eine weitere Möglichkeit für die externe Bereitstellung bietet der Typ *LoadBalancer*. Hierbei wird der Service über eine externe Anwendung zur Lastenverteilung bereitgestellt. Zuletzt kann durch den Typ *ExternalName* dem Service ein DNS Name zugeordnet werden. [25]

ConfigMap und Secret

Zur externalisierten Konfiguration einer Anwendung stellt Kubernetes die Objekte *ConfigMap* und *Secret* bereit. In diesen können Informationen Key-Value-Paare abgespeichert werden. Beide Möglichkeiten bieten den Vorteil, dass von Services benötigte Informationen über zentral im Cluster verfügbare Objekte bereitgestellt werden können. Durch die ausgelagerte Konfiguration muss insbesondere im Kontext einer Microservice Architektur nicht jede Instanz einer Applikation individuell konfiguriert werden. Eine Anwendung kann diese direkt über entsprechende ConfigMaps oder Secrets beziehen. Anzumerken ist jedoch, bei Bedarf der Speicherung von sensiblen Daten sind Secrets zu bevorzugen, da diese im Gegensatz zu ConfigMaps Informationen veschlüsselt im Cluster hinterlegen. [15][24]

Volumes

In containerisierten Anwendungen ist Speicher in der Regel flüchtig. Das bedeutet, so lange ein Pod existiert, hat er Zugriff auf den internen Speicher des Pods. Wird der Pod zerstört, wird auch der zugehörige Speicher aus dem System entfernt und somit auch alle von der Anwendung gespeicherten Daten. Da manche nicht triviale Applikationen persistenten, über die Lebensdauer eines Pods hinausreichenden Speicher benötigen, beispielsweise für Log-Dateien und Backup-Dateien, bietet Kubernetes die Möglichkeit, persistenten Speicher im gesamten Cluster bereitzustellen. Ein persistenter Speicher kann entweder statisch durch einen Administrator erstellt oder dynamisch über ein weiteres Kubernetes-Objekt, der *Storage Class*, bereitgestellt werden. Damit ein Pod ein Persistent Volum (PV) nutzen kann, stellt dieser eine Anfrage auf persistenten Speicher mittels eines Persistent Volume Claims. Bei einem Persistent Volume Claim werden zunächst alle statisch erstellten Persistent Volumes durchsucht. Entspricht kein PV den im Persistent Volume Claim (PVC) spezifizierten Anforderungen kann eine Storage Class Dynamisch neue PVs über einen Provisioner erstellen und diese dann zur Verfügung stellen. Provisioner sind für die Verteilung von Speicher zuständig ist. Das können Anwendungen wie beispielsweise GlusterFS sein, ein Netzwerk-Dateisystem das Dateisysteme von mehreren physischen Knoten zu einem logischen Gesamtsystem zusammenfasst und verwaltet. Weitere Möglichkeiten bieten die gängigen Cloud-Dienstleister wie Amazon und Microsoft mit beispielsweise AWSBlockstore, AzureFile oder AzureDisk. [26]

2.2.2 Spring Framework

Bei Spring handelt es sich um ein weitverbreitetes Java-Framework mit dem Ziel, die Entwicklung von Javaanwendungen zu erleichtern, indem es diese auf infrastruktureller Ebene unterstützt. Im Kern bedient sich Spring des Prinzips der Dependency-Injection (auch Inversion of Control). Von Objekten benötigte Abhängigkeiten werden nicht von den Objekten selbst, sondern vom Spring-Framework, dem Inversion of Control Container initialisiert, konfiguriert und zugewiesen. Alle von Spring verwalteten Objekte werden als Beans bezeichnet. [36]

Erweiterung	Beschreibung
Spring - Data JPA	Implementierung einer auf der Java Persistence API basierendenden Datenzugriffsschicht.
Spring - Data REST	Unterstützung bei der Implementierung von Hypermedia-gesteuerten REST-Webservices auf Basis von Spring Data-Repositories.
Spring - Security	Nahtlose Integration von Authentifizierung und Autorisierung in bestehende Spring Anwendungen.
Spring - Cloud	Senden von Steuerungsbefehlen an eine Node.

Tabelle 2.2: Beispiele von existierenden Spring-Erweiterungen

Tabelle 2.2 listet einige populäre Erweiterungen auf die das Spring Ökosystem zur Verfügung stellt. Mittels dieser ist es möglich, alle wichtigen Aspekte einer Enterprise-Applikation vollumfänglich abzudecken und nahtlos in das Spring Ökosystem zu integrieren. [37]

2.2.2.1 Spring Boot

Spring Boot erweitert das Spring-Ökosystem, indem es Templates zur Projekterstellung und vorkonfigurierte Bibliotheken von Drittanbietern zur Verfügung stellt, um eine schnellere Entwicklung von Spring-Anwendungen mit minimalem Konfigurationsaufwand zu ermöglichen. Dabei handelt es sich unter anderem auch um nicht-funktionale Funktionalitäten die oft in verteilten Systemen Anwendung finden. Darunter die Bereitstellung von Metriken über verbrauchte Ressourcen oder eine ausgelagerte (externalisierte) Konfiguration. Diese können mithilfe der Build-Tools Gradle oder Maven auf einfache Weise in ein bestehendes Projekt integriert werden und werden von Spring als Sammlung von allen notwendigen Abhängigkeiten unter der Bezeichnung *Spring Starters* angeboten. Für verschiedene Funktionalitäten existieren unterschiedliche *Spring Starter* Abhängigkeiten. [35]

2.2.3 Apache Maven

Apache Maven ist ein im Java-Ökosystem weit verbreitetes Tool zur Projektverwaltung. Es automatisiert unter anderem den Build-Prozess und verwaltet alle Abhängigkeiten eines Projektes. Alle Abhängigkeiten und Build-Spezifikationen werden zentral in einer Datei gebündelt, welche nach Maven-Spezifikation als *Project Object Model (POM)-Datei* bezeichnet wird. Die POM-Datei definiert in der Auszeichnungssprache *XML* allgemeine Informationen über ein Projekt, wie das verantwortliche Unternehmen, Versionsnummer, Autoren und Name des Projekts. [30]

Neben diesen allgemeinen Informationen definiert das *Project Object Model* alle Abhängigkeiten, welche von der Applikation verwendet werden. Diese werden zentral über das *Central Maven Repository* bezogen. Dabei handelt es sich um ein zentrales Repository in dem alle mit Maven kompatiblen Abhängigkeiten als Artifakte hinterlegt sind. Mit allen in der POM-Datei hinterlegten Informationen ist es Maven möglich den Build-Prozess für ein Projekt auszuführen. Dieser teilt sich in mehrere Phasen auf.

Die erste Phase des Lebenszyklus ist die *Validierung*, in welcher das Projekt auf Korrektheit und Vorhandensein aller nötigen Informationen geprüft wird. Die *Compile-Phase* kompiliert im zweiten Schritt den Code. Alle im Projekt definierten Unit-Tests werden in der *Test-Phase* mit einem entsprechenden Testing-Framework ausgeführt. Der kompilierte und getestete Code wird in der vierten, der *Package-Phase*, in ein Format gepackt das zur Auslieferung bereit ist. Anschließend wird in der *Verify-Phase* die Applikation im gepackten Format ausgeführt und alle Integration-Tests ausgeführt. Mit Hilfe der *Install-Phase* wird die Applikation in einem lokalen Repository installiert. So kann sie beispielsweise als Abhängigkeit für weitere lokal entwickelte Maven-Projekte verwendet werden. In der letzten, der *Deploy-Phase*, verlässt die Applikation die lokale Build-Umgebung und wird in ein Remote-Repository geladen um das Projekt für andere Entwickler und Projekte verfügbar zu machen. [31]

Der zuvor beschriebene Lebenszyklus ist der *Default-Lifecycle*, einer von drei verfügbaren Lebenszyklen. Die weiteren Lebenszyklen *Clean-Lifecycle* und *Site-Lifecycle* sind dafür zuständig, alle durch den Build-Prozess entstandenen Artefakte zu entfernen und die Projekt-Dokumentation zu kompilieren und zu deployen. [31]

2.2.4 RabbitMQ

RabbitMQ ist ein in der funktionalen und für verteilte Systeme optimierten Programmiersprache Erlang implementierter Open-Source-Message-Broker. Als Message Oriented Middleware (MOM) unterstützt RabbitMQ die Kommunikation durch das Advanced Message Queing Protocol (AMQP). Ein Message-Broker ist ein Software-System, dass eine asynchrone Kommunikation zwischen Anwendungen ermöglicht. Zugehörige Anwendungen werden gemeinhin als *Producer* und *Consumer* bezeichnet. Ein Producer oder auch Publisher generiert Nachrichten und sendet diese über den Message-Broker auf einem festgelegten Nachrichtenkanal. Den Anwendungspart, der in der Lage ist Nachrichten über den Broker zu empfangen, bezeichnet man als Consumer oder Subscriber. Dieser wartet auf dem zuvor spezifizierten Nachrichtenkanal auf einkommende Nachrichten, um diese dann weiter zu verarbeiten. Durch die asynchrone Natur dieser Kommunikation bietet RabbitMQ eine ideale Grundlage für den Entwurf von komplexen, verteilten Systemen gegeben. Nativ unterstützt RabbitMQ das AMQP-Protokoll über verschiedene Exchange-Typen. Exchanges definieren wie Nachrichten weitergeleitet werden. [33]

2.2.4.1 MQTT Support

RabbitMQ unterstützt die Kommunikation via MQTT gemäß Version 3.1.1 der Spezifikation über eine entsprechende Erweiterung, die über Konfigurationseinstellungen aktiviert werden kann. Hierbei ist anzumerken, dass der MQTT-Support einigen wenigen Limitationen unterliegt und die nachfolgenden Features unterstützt.

RabbitMQ unterstützt lediglich QoS 0 und QoS 1. Alle mit QoS 2 veröffentlichten Nachrichten werden automatisch vom Broker auf QoS 1 heruntergesetzt. Dies ist auf die Implementierung des MQTT-Plugins zurückzuführen. Das Plugin nutzt auf unterster Ebene immer noch AMQP um Nachrichten weiterzuleiten. AMQP unterstützt laut Spezifikation kein Quality of Service Level 2. Unterstützt werden die MQTT-Funktionalitäten *Last Will and Testament (LWT)*, *Retained Messages*, *Clean Sessions* und Support für *TLS*-Verbindungen. [34]

2.2.5 CockroachDB

CockroachDB ist eine relationale SQL-Datenbank mit Fokus auf den Einsatz in Cloud Umgebungen. Dabei implementiert CockroachDB den ANSI-SQL-Standard und ist mit der weit verbreiteten Open-Source-Datenbank *PostgreSQL* kompatibel. Das bedeutet, dass die überwiegende Mehrheit der PostgreSQL kompatiblen Datenbanktreiber und Frameworks, ebenso kompatibel zu CockroachDB sind. Viele PostgreSQL-Anwendungen können auf CockroachDB portiert werden, ohne dass wesentliche Änderungen am Code erforderlich sind. So ist beispielsweise ein Großteil der bekannten PostgreSQL-Erweiterung PostGis, eine Erweiterung zur Verwaltung von geographischen Daten, bereits in CokroachDB integriert. [12]

Im Kern handelt es sich bei CockroachDB um einen *Key-Value-Store*, der nach außen eine SQL-API bereitstellt, die SQL-Syntax in Operationen auf dem Key-Value-Store übersetzt. Serialisierbare Transaktionen, das höchste Isolationslevel für Transaktionen definiert durch den ANSI-SQL-Standard, werden realisiert durch Verwendung des *Raft Consensus Algorithmus* für Schreiboperationen und einem individuell angepassten, auf Timestamps basierenden Synchronisationsalgorithmus für Leseoperationen. Die Mehrheit aller Repliken eines Clusters muss einer Schreiboperation zustimmen, damit diese persistiert wird.

Gespeicherte Daten werden versioniert, so dass Leseoperationen nur die Daten sehen, die zum Zeitpunkt des Starts Operation verfügbar sind. Grundsätzlich sind alle Knoten in einem CockroachDB Cluster miteinander vernetzt und kommunizieren mit Hilfe des *Gossip Protokolls*. Dadurch existiert kein Single Point of Failure (SOP) und der Clusterbetrieb wird bei Ausfall einer Node nicht eingeschränkt und es kann eine stark konsistente Sicht der Daten garantiert werden. Weiterhin ermöglicht CockroachDB eine nahtlose horizontale Skalierung indem auf einfache Weise dem Cluster neue Knoten hinzugefügt werden können. Eine weitere Besonderheit stellt die Möglichkeit zur Geo-Partitionierung eines Cockroach-Clusters dar. Ist ein Cluster über mehrere Regionen verteilt, ist es möglich Daten physisch an bestimmten Orten zu speichern, um eine bessere Leistung für lokalisierte Anwendungen zu erzielen. Jedoch kann nicht gänzlich auf Kompromisse verzichtet werden um Garantien wie eine starke Konsistenz zu gewährleisten. Um Konsistenz unter allen Umständen zu gewährleisten, müssen Abstriche in Hinblick auf Verfügbarkeit gemacht werden. Es kann der seltene Fall eintreten, dass eine Node keine Daten zurückgibt, wenn sie in Folge einer Netzwerkpartition von anderen Nodes getrennt wurde. Im Falle einer Partition ist dennoch in jedem Fall die Partition mit der Mehrheit der lauffähigen Knoten weiterhin verfügbar. Ebenso werden transaktionale Workloads stets priorisiert, weshalb andere Lösungen gegebenenfalls CockroachDB zu bevorzugen sind, wenn analytische Workloads Priorität haben. [12]

2.2.6 Eclipse Paho Client

Die Client-Implementierung der MQTT-Spezifikation steht für verschiedene Versionen und Programmiersprachen zur Verfügung. Bereitgestellt wird die Implementierung kostenlos von der *Eclipse-Foundation*. Aufgrund der weiten Verbreitung wird der Eclipse-Paho-Client ständig weiterentwickelt und für eine immer weiter wachsende Zahl an Programmiersprachen weiterentwickelt. Abbildung 2.10 stellt eine Übersicht der aktuell unterstützten Versionen dar. [38]

Client	MQTT 3.1	MQTT 3.1.1	MQTT 5.0	LWT	SSL / TLS	Automatic Reconnect	Offline Buffering	Message Persistence	WebSocket Support	Standard MQTT Support	Blocking API	Non-Blocking API	High Availability
Java	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Python	✔	✔	✔	✔	✔	✔	✔	✖	✔	✔	✔	✔	✖
JavaScript	✔	✔	✖	✔	✔	✔	✔	✔	✔	✖	✖	✔	✔
GoLang	✔	✔	✖	✔	✔	✔	✔	✔	✔	✔	✖	✔	✔
C	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
C++	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Rust	✔	✔	✖	✔	✔	✔	✔	✔	✖	✔	✔	✔	✔
.Net (C#)	✔	✔	✖	✔	✔	✖	✖	✖	✖	✔	✖	✔	✖
Android Service	✔	✔	✖	✔	✔	✔	✔	✔	✔	✔	✖	✔	✔
Embedded C/C++	✔	✔	✖	✔	✔	✖	✖	✖	✖	✔	✔	✔	✖

Abbildung 2.10: Übersicht der verfügbaren Client-Implementierungen

2.2.7 Grafana Stack

2.2.7.1 Grafana

Grafana ist eine Open Source Software zur Datenanalyse und deren Visualisierung über individuell konfigurierbare Dashboards. Der Zugriff der Daten erfolgt unabhängig der Datenhaltung. Grafana ist kompatibel mit einer heterogenen Palette an Datenquellen. Oft wird Grafana im Zusammenspiel mit der Monitoring-Anwendung *Prometheus* genutzt, um in verteilten Systemen Metriken von Systemkomponenenten zu beobachten. Grafana bietet hier umfassendere Möglichkeiten zur Visualisierung als Prometheus selbst. [43]

2.2.7.2 Grafana Loki

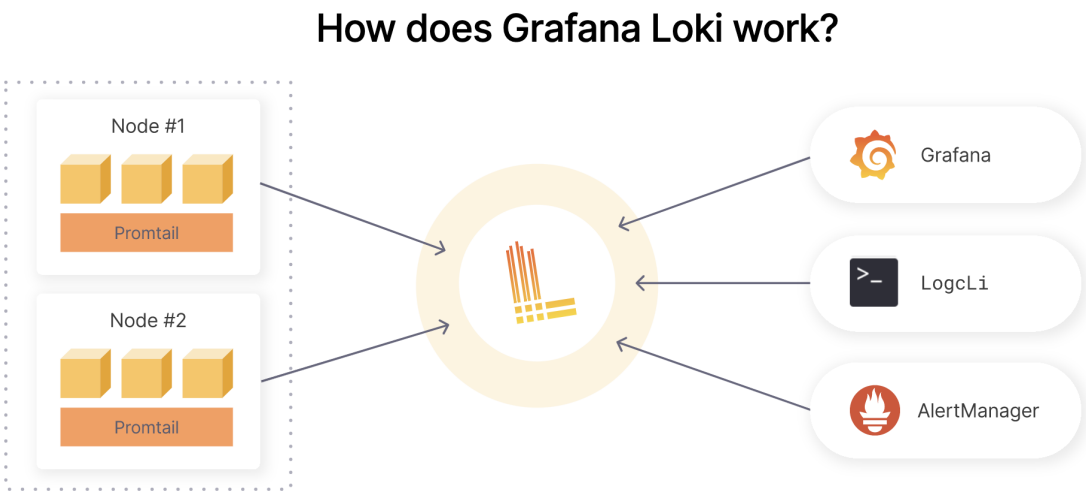


Abbildung 2.11: Funktionsweise von Grafana Loki [56]

Loki ist ein horizontal skalierbares Log-Aggregation System, optimiert für den Betrieb in durch Kubernetes verwalteten Cluster. Die zu Grunde liegende Architektur ist in Abbildung 2.11 skizziert. Es garantiert eine nahtlose Integration in einen bestehenden Grafana/Prometheus-Stack. Logs werden üblicherweise, insofern nicht in einem Logfile gespeichert, über die Streams *stdout* und *stderr* ausgegeben. Diese stellen jeweils einen Log-Stream dar. In einem Loki Cluster-Deployment findet sich auf jeder Node eine Instanz des Log-Collectors Promtail. Dieser sammelt über die Log Streams *stdout* und *stderr* alle generierten Logs, welche dann anschließend in einem *Object Storage* persistiert werden. Visualisierungstools wie Grafana können nun alle so gewonnenen Logs abfragen und über Dasboards visualisieren. Für die die Abfrage der Logs wird die eigens entwickelte Abfragesprache LogQL verwendet. [56]

2.2.7.3 Prometheus

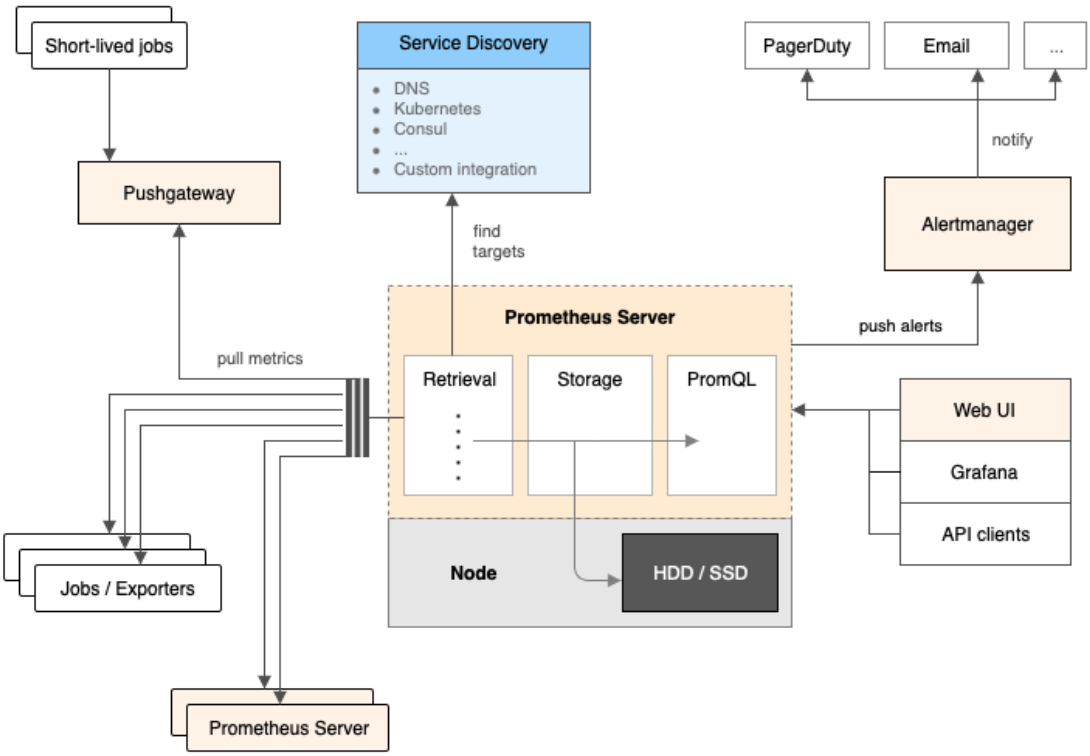


Abbildung 2.12: Funktionsweise von Prometheus [65]

Prometheus ist ein Cloud natives Monitoring-Tool zur Aggregation von Metriken in verteilten System. Wie Abbildung 2.12 zu entnehmen, arbeitet Prometheus, um Metriken zu sammeln, nach dem Pull-Prinzip. Zu beobachtende Systeme stellen einen entsprechende HTTP-Ressource zur Verfügung über die entsprechende Metriken bereitgestellt werden. In regelmäßigen Intervallen bezieht Prometheus Metriken von registrierten Services. Alle Services die entsprechende Endpunkte zur Veröffentlichung von Metriken zur Verfügung stellen, werden über einen Service Discovery-Mechanismus automatisch erfasst. So kann eine Prometheus-Instanz in einem Kubernetes Cluster automatisch Metriken von neu deployten Services aggregieren, insofern diese einen entsprechenden Endpunkt zur Verfügung stellen. Von Prometheus persistierte Metriken können mit der speziell für Prometheus entwickelten Query Language Prometheus Query Language abgefragt werden. [65]

2.3 Hardware und Infrastruktur

Zur Erforschung von Smart City- und IoT-Themen hat das Labor für verteilte Systeme der HTW-Saarland eine entsprechende Infrastruktur erarbeitet um Forschungsprojekte zu Testen und zu Evaluieren.

2.3.1 CiTe-Testfeld

Das durch das DSL verwaltete CiTe-Testfeld stellt den Kern der Forschung im Bereich *Smart City* und *IoT* an der HTW-Saar dar. Im Zentrum des Testfeldes steht ein Cluster, welcher mittels des Clustermanagement-System Kubernetes verwaltet wird. Innerhalb

des Clusters existieren für die Infrastruktur relevante Services.

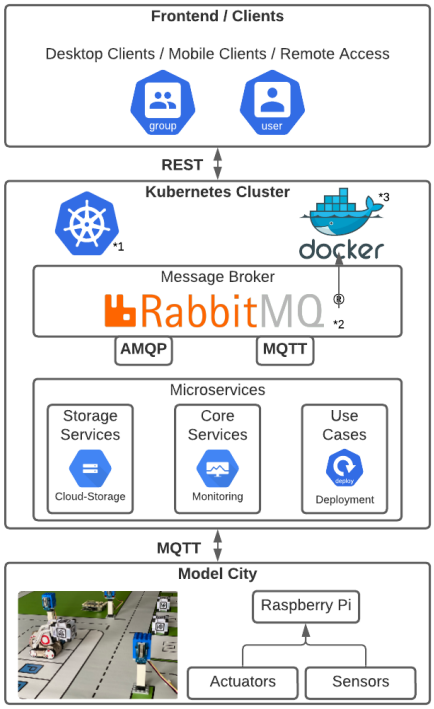
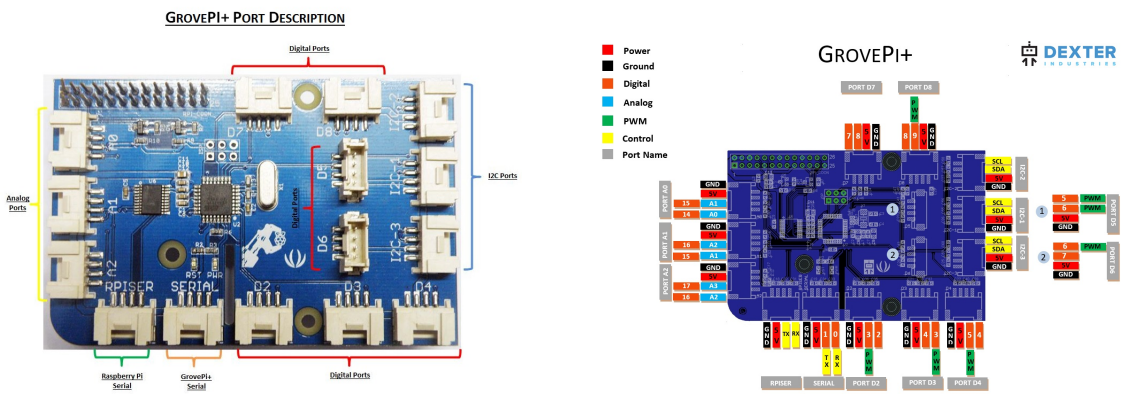


Abbildung 2.13: Architektur des CiTe-Testfeldes [1]

Ein Cluster-Deployment des Message-Brokers RabbitMQ ermöglicht eine asynchrone Kommunikation, entweder über das Nachrichtenprotokoll MQTT oder AMQP. Die Komponente *Model City* umfasst Sensoren und Aktoren, welche Daten erzeugen und versenden oder Steuerungsdaten empfangen können. Der Austausch von Daten erfolgt via MQTT-Protokoll über den Message-Broker. Zur Datenhaltung stehen unter anderem ein Deployment der NoSQL-Datenbank *Cassandra*, sowie wie *MinIO* als *Object Store* zur Verfügung. Abbildung 2.13 skizziert die Gesamtarchitektur und teilt diese in die drei Komponenten *Frontendanwendungen & Clients*, *Kubernetes Cluster*, sowie *Model City* auf.

2.3.2 Raspberry Pi & GrovePi+

Zur Anbindung von Sensoren und Aktoren wird ein *Raspberry Pi* verwendet. Dieser fungiert zusammen mit dem Erweiterungsboard *GrovePi+* als Gateway um Sensoren mit dem Message-Broker zu verbinden. Als MQTT-Client wird hauptsächlich die *Eclipse Paho* Implementierung verwendet, da diese in verschiedenen Programmiersprachen unterstützt wird. Aufgrund der Verwendung des GrovePi+-Erweiterungsboard müssen entsprechende Sensoren der Marke *Seeed Studio* verwendet werden, welche explizit für das Grovepi+-Board entwickelt wurden.



(a) Übersicht Ports (b) Übersicht Pins

Abbildung 2.14: Übersicht der verwendbaren Ports und Pins

Sensoren und Aktoren unterteilen sich in drei verschiedene Typen von Sensoren. *Analoge* Geräte verarbeiten lediglich analoge Signale und müssen an einem entsprechenden analogen Port angeschlossen werden. *Digitale* Sensoren und Aktoren verarbeiten digitale Signale und können ebenfalls nur an den dafür ausgewiesenen Anschlüssen betrieben werden. Der letzte Typ von Geräten umfasst Inter Integrated Circuit (I2C) kompatible Geräte. [44] I2C ist eine von Philips entwickelte Datenbus-Schnittstelle zur einfachen Integration von Peripheriegeräten in eingebettete Systeme. [80]

3 Verwandte Arbeiten und Stand der Technik

3.1 CiTe - Verwandte Arbeiten

Mit dem CiTe-Testfeld als Grundlage zur Erforschung von IoT-Themen, wurden bereits in der Vergangenheit Arbeiten verfasst, deren Inhalt sich mit Teilen dieser Arbeit überschneidet. Im Folgenden werden diese Arbeiten erläutert und in den Kontext dieser Arbeit eingeordnet.

3.1.1 Dynamische Anbindung von Sensoren und Aktoren an ein Cloud-Backend

Die Arbeit von Wagner beschäftigt sich ebenfalls mit der Dynamischen Anbindung von Sensoren und Aktoren im CiTe-Testfeld. In diesem Kontext wird hardwareseitig mit verschiedenen Geräten gearbeitet. Zur Anbindung wird zum einen ein Raspberry Pi mit dem Erweiterungsboard GrovePi+ als auch ein Arduino mit dem Erweiterungsboard Grove-Shield verwendet. Entsprechende Sensoren und Aktoren von Seeed Studio, welche mit dem gegebenen Ökosystem kompatibel sind, werden als Geräte zur Erfassung und Empfang von Daten verwendet.

Im Fokus der Arbeit steht die automatische Erfassung von an den Grove-Erweiterungsboards angeschlossenen Peripheriegeräten. Durch mehrere Experimente hat Wagner versucht eine Möglichkeit zur automatischen Anbindung zu finden. Diese unterteilen sich nach dem Grad der Informationen die automatisch bei Anschluss eines Gerätes erfasst werden sollen. Sowohl Arduino als auch Raspberry-Pi bieten beide verschiedene Typen von Hardware-Pins an. Diese differenzieren wie eingehende Signale verarbeitet werden. Dabei wird zwischen analogen-, digitalen- und I2C-Pins unterschieden. Weiterhin ist für eine Plug'n Play Anbindung die Erfassung des Typs eines Gerätes erforderlich. Hierbei wird lediglich zwischen Sensor und Aktor unterschieden. Eine detaillierterer Informationsgrad schließt in letzter Ebene die konkrete Art eines Peripheriegerätes mit ein. Dieser umfasst konkrete Metadaten wie beispielsweise die SI-Einheit, den Datentyp der erfassten Daten sowie, ob es sich beispielsweise um einen Temperatursensor oder Lichtsensor handelt.

Als Resultat der durchgeführten Experimente konnte Wagner feststellen, dass lediglich eine rudimentäre Erfassung auf den digitalen Pins möglich ist, die es ermöglicht festzustellen, dass ein Gerät angeschlossen wurde, jedoch keine Aussage über die Art des Gerätes erlaubt. Dementsprechend erfordert die Konfiguration von Peripheriegeräten in diesem Kontext Kenntnis über angeschlossene Geräte. [79]

3.1.2 Konzeption und Implementierung einer Microservices-Architektur zur Sensordatenverarbeitung und -aggregation für IoT-Anwendungen

Studer konzipiert und implementiert in seiner Arbeit eine geeignete Architektur um Peripheriegeräte dynamisch anzubinden, deren Daten zu erfassen und zu verarbeiten. Damit eine problemlose Integration in das CiTe-Umfeld gewährleistet werden kann, basiert das von Studer konzipierte System auf einer Microservice-Architektur. Die grundlegende Funktionalität des Gesamtsystems umfasst die Netzwerkkommunikation von Sensordaten über einen Message Broker mittels des MQTT-Protokolls, die grundlegende Verwaltung von Peripheriegeräten im Gesamtsystem gekoppelt an ein entsprechendes Datenhaltungssystem, sowie die Erhebung und Aggregation von Sensordaten. Durch Gruppierung von Sensoren können Aggregate gebildet werden um komplexere Datenpunkte zu erstellen, wobei in deren Berechnung die Rohdaten mehrerer Messwerte unterschiedlicher Sensoren einfließen. Dabei handelt es sich um die Berechnung von Durchschnittswerten, Minima und Maxima, aber auch spezifischeren komplexeren Berechnungen wie beispielsweise von Taupunkten oder Kondensationswerten.

Die Implementierung wurde in der Programmiersprache Java umgesetzt und stützt sich auf das Spring Framework. Dabei wurden Microservices zur Rohdatenerfassung, Aggregatbildung und historischen Aggregation entwickelt. Diese Services sind in der Lage entsprechende Daten zu erheben, verarbeiten und in ein Datenhaltungssystem zu persistieren. Weiterhin wurde, da die Erhebung und Prozessierung der Daten im Fokus der Arbeit standen, ein einfacher Prototyp zur Verwaltung von Metadaten entwickelt, welcher durch eine einfache *REST-Schnittstelle* angesprochen werden kann. Um eine erweiterbare Code Basis bereitzustellen wurde in der Entwicklung darauf geachtet Abläufe nach Möglichkeit zu generalisieren und ein entsprechendes Template über das Maven-Plugin *Maven Archetypes* bereitzustellen. Auf diese Weise kann mit geringem Aufwand ein Template-Projekt auf Basis der Archetypes erstellt werden.

Damit alle Services auf einfache Weise bereitgestellt werden können, erfolgt das Deployment über entsprechende Deploymentkonfigurationen auf dem Kubernetes Cluster des CiTe-Testfelds. Alle Services können beliebig skaliert werden und werden auch im aktiven Betrieb je nach Auslastung durch Verwendung des *HorizontalPodAutoscaler* skaliert. Durch Angabe der Konfiguration über *ConfigMaps* wird die redundante Angabe von Konfiguration beim Deployment von mehreren Instanzen eines Service vermieden. Zusätzlich zu den Deployments der Microservices stellt Studer eine im Cluster bereitgestellte Grafana und Prometheus Instanz zur Verfügung. Mithilfe von Prometheus werden entsprechende Metriken der Services gesammelt. Diese werden an Grafana weitergeleitet und über entsprechend erstellte Dashboards visualisiert. [76]

3.1.3 Konzeption und Implementierung einer Architektur zur Verteilung von Steuerungs- und Navigationsaufgaben an Roboter in einem Smart-City-Umfeld

Ein weiteres Anwendungsszenario wurde von Altmeyer umgesetzt. Ziel der Arbeit war die Anbindung von Robotern und deren Steuerung, über die durch das CiTe-Testfeld bereitgestellte Infrastruktur. Roboter fungieren in dieser Arbeit als verwaltende Instanz von entsprechenden Sensoren und Aktoren. Dementsprechend gilt es in diesem Kontext diese zu verwalten und an das System anzubinden. Die Netzwerkkommunikation erfolgt über einen Message Broker mittels des MQTT-Protokolls. Altmeyer konzipiert ebenfalls eine auf dem Microservice Paradigma basierende Architektur, damit kompatible Roboter

nahtlos in das Testfeld integriert werden können. Das entworfene Gesamtsystem lässt sich in drei Komponenten aufteilen. Das Backend umfasst alle im Cluster bereitgestellten Microservices und stellt die Kernkomponenten der Architektur bereit. Es werden Microservices zur Verwaltung von Gerätedaten, zur Kalkulation von Wegfindungsalgorithmen und ein Service zur Steuerung der Roboter, welcher Steuerungsbefehle weiterleitetet und Sensordaten aggregiert, bereitgestellt.

Ergänzend zur Backendarchitektur wurde eine Webanwendung entworfen, welche ein grafisches User Interface zur Verfügung stellt. Die Abfrage der erforderlichen Daten erfolgt über entsprechende REST-Schnittstellen der einzelnen Microservices.

Die letzte Komponente des Gesamtsystems stellt die Software auf Seite der Roboter dar. Diese fungiert als Client für den Message-Broker und ermöglicht somit das Empfangen und Senden von relevanten Daten. So können Steuerungsbefehle prozessiert und Sensordaten publiziert werden. Weiterhin wird über diesen Kommunikationskanal die Registrierung von neuen Geräten vorgenommen. Die Registrierung erfordert in diesem Fall, dass eine Roboterkonfiguration bereits Kenntnis über alle benötigten Daten besitzt. Verwaltungsrelevante Informationen werden über entsprechend dafür entworfene Kommunikationskanäle kommuniziert. [3]

3.2 Thingsboard.io - Enterprise Lösungen für IoT Anwendungen

Die Forschung im Bereich IoT hat bereits viele Softwarelösungen zur Verwaltung von IoT-Geräten hervorgebracht. Ein Beispiel dafür ist die Open Source-Plattform Thingsboard.io. Dabei handelt es sich um eine umfassende Lösung zur Verwaltung von IoT kompatiblen Geräten. Thingsboard unterstützt aufgrund des breiten heterogenen Angebots an Geräte die am weitesten verbreiteten Transportprotokolle, wie MQTT, HTTP, CoAP, SNMP und LwM2M.

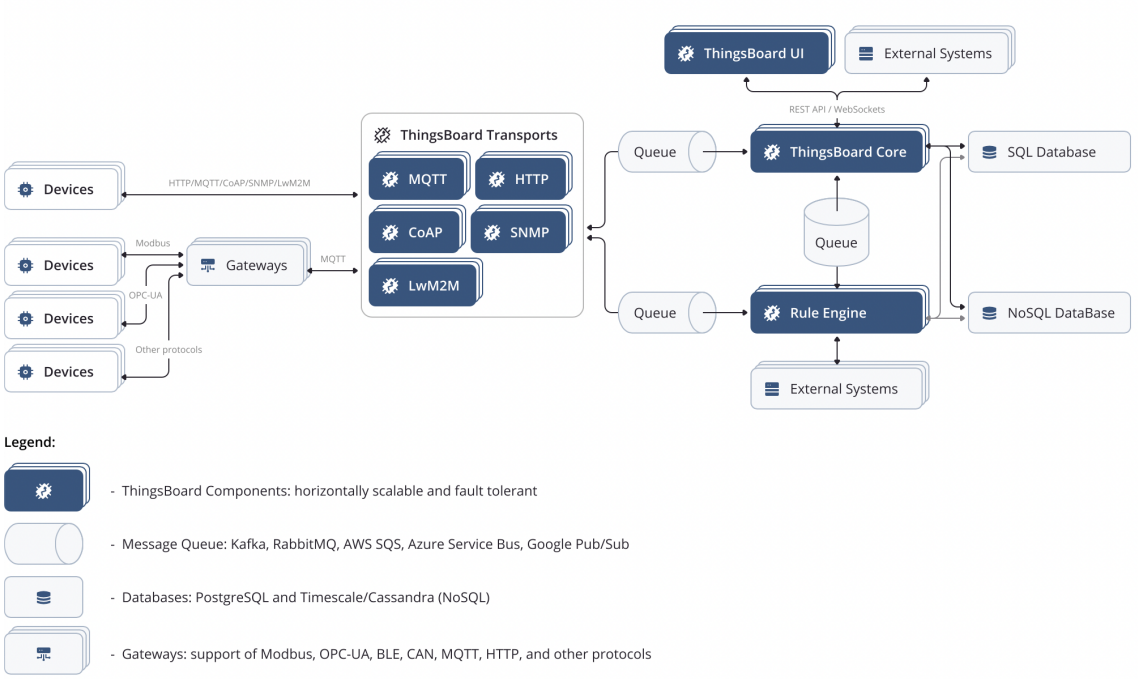


Abbildung 3.1: Architektur des gesamten Thingsboard-Stack [78]

Abbildung 3.1 zeigt den grundlegenden Aufbau der Software. Neben der großen Bandbreite an unterstützten Transportprotokollen steht neben den Kernfunktionalitäten eine Rule Engine zur Verfügung, mit deren Hilfe individuelle eventbasierte Abläufe umgesetzt werden können. Registrierte Geräte und Messwerte können über anpassbare Dashboards visualisiert werden. Weiterhin werden die meisten SQL- und NOSQL-Datenbanksysteme unterstützt und können als Datenhaltungssysteme registriert werden.

Zur Bereitstellung werden kostenpflichtige Cloud-Lösungen als auch, je nach Edition, kostenlose On-Premise-Lösungen angeboten. Dabei werden wiederum verschiedene Modi zum Deployment angeboten. Die Software kann als monolithische Applikation auf einem einzelnen Server, als auch unter Verwendung einer Microservice Architektur auf einem Cluster betrieben werden. Thingsboard versucht durch das breite Angebot an unterstützten Protokollen und Funktionalitäten möglichst viele Anwendungsszenarien abzudecken. Dabei wird die Verwaltung der Geräte, die Erfassung von Sensorwerten, die Steuerung von Aktoren in einer Nutzeroberfläche vereint. Dementsprechend erfordert sowohl die Nutzung als auch die Konfiguration der Software einen erheblichen Lernaufwand.

Thingsboard ermöglicht mit der grafischen Nutzeroberfläche einen einfachen Einstieg in die Nutzung der Software, jedoch kommt diese bei komplexeren Problem schneller an ihre Grenzen oder diese sind schwerer umzusetzen, da die Implementierung an die vorgegebenen Strukturen gebunden ist. Beispielsweise wird die MQTT-Topicstruktur hier vom Hersteller vorgegeben und ist dementsprechend bei Bedarf nicht einfach anzupassen oder zu erweitern. Gerade im Kontext von Forschungsprojekten besteht gegebenenfalls nicht die Möglichkeit neue Erkenntnisse zu erproben und umzusetzen, da eine direkte Abhängig zum Gesamtsystem besteht. Weiterhin gilt es auch das Monetarisierungsmodell zu betrachten. Aufgrund der Aufteilung in kostenpflichtige Enterprise-Lösungen und kostenfreie Community-Version steht zwar eine frei verfügbare Lösung zur Verfügung, jedoch ist die zukünftige Entwicklung in Punkten Lizenzierung, Monetarisierung und der somit in der freien Version verfügbaren Funktionalität nicht abzusehen. Im Hinblick auf die genannten Aspekte ist in diesem Fall die Entwicklung einer eigenen Lösung zu bevorzugen. So kann im Kontext der IoT-Forschung eine schnellere Iteration von Experimenten zur Umsetzung neuer Erkenntnisse der Forschung erprobt werden, da eine bessere Anpassung, Transparenz und Flexibilität gewährleistet werden kann. [78]

3.3 Standardisierte Kommunikation im IoT-Umfeld

Ein wesentlicher Aspekt im IoT-Umfeld ist die Kommunikation. Darunter versteht man die Kommunikation zwischen Geräten, die Publikation von erfassten Sensordaten, das Versenden von Steuerbefehlen an Aktoren. Viele IoT-Systeme setzen dabei eine eigene Kommunikationsstruktur um, was eine erhebliche Fragmentierung von Kommunikationsstrukturen und Nachrichtenformaten in der IoT-Gerätelandschaft zur Folge hat. Dementsprechend gibt es immer mehr Bemühungen diesem Trend entgegenzuwirken, indem versucht wird einen allgemeinen Konsens in diesem Aspekt zu finden. Im Folgenden werden zwei Resultate solcher Bemühungen vorgestellt. Diese schlagen eine standardisierte Kommunikationsstruktur für IoT-Geräte vor, welche über das Transportprotokoll MQTT kommunizieren.

3.3.1 Eclipse Sparkplug Spezifikation

Die *Eclipse Sparkplug*-Spezifikation ist eine auf dem *OASIS Mqtt v3.1.1 Standard* aufbauende Spezifikation. Ins Leben gerufen wurde die Sparkplug-Spezifikaiton von der Eclipse Foundation. Die Eclipse Foundation beschäftigt sich in verschiedenen Arbeitsgruppen mit verschiedenen IoT-Themen. In diesem Kontext wird das Ziel verfolgt einen einheitlichen Industriestandard für die Kommunikation über das MQTT-Protokoll zu etablieren. Im Kern der Spezifikation stehen drei Aspekte. Die Definition eines allgemeinen Topic Namespace soll einen homogene Kommunikationsstruktur ermöglichen, so dass alle Geräte unabhängig des Herstellers auf die gleiche Weise in der Lage sind zu kommunizieren. Analog zur Definition des Topic-Namensraums wird ein allgemein gültiges Nachrichtenformat definiert. Weiterhin definiert die Spezifikation die Umsetzung eines Zustandsmanagement, so dass der Zustand eines Gerätes zu jeder beliebigen Zeit beobachtet werden kann.[41, p.6]

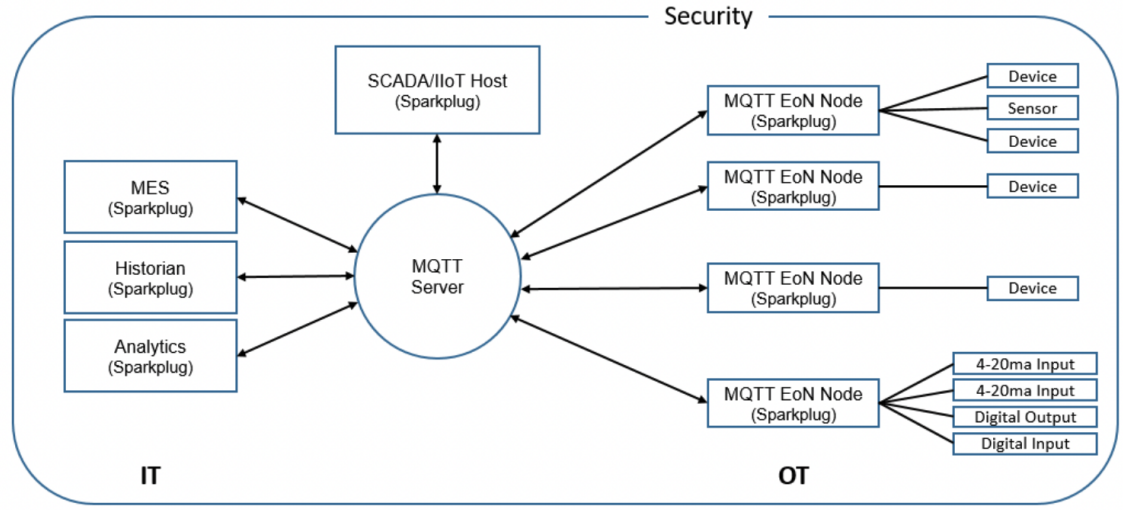


Abbildung 3.2: Infrastruktur eines Sparkplug kompatiblen Systems [41, p. 8]

Im Fokus der Spezifikation steht die Vernetzung von Anwendungen, die Sensordaten konsumieren mit den letztlich physischen Geräten, die diese Daten produzieren. Alle Applikationen dieser Art lassen sich unter dem Begriff *Information Technology (IT)* zusammenfassen. Diesem gegenüber steht der Begriff *Operation Technology (OT*). Darunter versteht man alle Geräte an das Netzwerk anzubindende Peripheriegeräte. Durch die Kommunikation über MQTT kompatible Message-Broker wird diese Brücke von *IT* zu *OT* geschlagen. Die Sparkplug-Spezifikation führt, wie in Abbildung 3.2 skizziert, Terminologie ein um die verschiedenen Komponenten in einer solchen Infrastruktur voneinander abzugrenzen. Im Zentrum, zuständig für die Orchestration von Nachrichten über festgelegte Topics steht ein Message-Broker. Eine Komponente die ebenfalls eine zentrale Rolle einnimmt ist der IIoT-Host. Dabei handelt es sich um eine oder mehrere MQTT-Client-Anwendungen, die auf entsprechenden Topics, von Geräten bereitgestellte Informationen konsumieren. In der Regel fungieren diese Anwendungen als Verwaltungs- und Monitoring-Instanz von im Netzwerk aktiven Peripheriegeräten. Eine EoN-Node stellt ein MQTT fähiges Gerät dar. Dabei kann es sich entweder um ein Sensor oder Aktor mit einer nativen MQTT-Client-Integration handeln oder um ein Gateway, welches ein oder mehrere Sensoren verwaltet und an das System anbindet. Als *Application Node* werden alle Anwendungen bezeichnet, die auf entsprechenden Topics Daten von EoN-Geräten konsumieren oder

Daten über diese an EoN-Geräte senden. [41, p. 8f]

Zur Umsetzung dieser Infrastruktur stützt sich die Sparkplug-Spezifikation auf diverse standards und Open Source Software. Grundbaustein bildet die *OASIS MQTTv3.1.1*-Spezifikation des MQTT-Protokolls, welche durch die Sparkplug Spezifikation erweitert wird. Als MQTT-Client Implementierung wird der *Eclipse Paho*-Client als Referenzimplementierung betrachtet. Nachrichten werden mittels des binären Nachrichtentyps *Google Protocol Buffer* umgesetzt und durch das *Kura Google Protocol Buffer Schema* definiert. [41, p. 10]

```
namespace/group_id/message_type/edge_node_id/[device_id]
```

Listing 3.1: Topic-Struktur der Eclipse Sparkplug Specification

Die von der Spezifikation, wie Listing 3.1 zu entnehmen, vorgeschlagene Topic-Struktur bildet eine hierarchische Struktur ab. Auf oberster Ebene wird eine Namespace-Kennung angegeben. In der Spezifikation werden zwei verschiedene Payload Definitionen, Sparkplug A und Sparkplug B, vorgestellt. Das namespace Element bezieht sich auf die jeweilige Version der Payload Definition und kann einen der folgenden Werte annehmen [41, p. 12]:

- **spAv1.0**
- **spBv1.0**

Das Element *group_id* impliziert eine logische Gruppierung von EoN-Nodes, dabei ist zu beachten, dass die Benennung einer Gruppe von Restriktionen unterliegt. Eine *group_id* muss ein gültiger UTF-8 kodierter String sein, ausgenommen der reservierten Zeichen *"+" und "#"*, welche der Nutzung der Wildcard-Funktionalität obliegen. [41, p. 12] Wie der Inhalt einer Nachricht verarbeitet werden soll, wird durch das *message_Type*-Element definiert. Dabei gilt zu beachten, dass die Kodierung des Payloads abhängig von der Version der Sparkplug-Spezifikation abhängt. Diese wird durch das *namespace*-Element angegeben. Das *edge_node_id*-Element dient als eindeutige Kennung eines EoN-Node-Devices. In Kombination mit der *group_id* stellt die *edge_node_id* eine eindeutige Kennung eines Gerätes dar. Ein *device_id* Element kennzeichnet ein Gerät das physisch oder logisch an eine EoN-Node gekoppelt ist. Diese Kennung muss sich von anderen Devices, die mit der gleichen EoN-Node verbunden sind, unterscheiden, kann jedoch von einer EoN-Node zu einer anderen identisch sein. Grundsätzlich müssen alle im Topic spezifizierten Elemente als alphanumerischer String im UTF-8 Format angegeben werden. Da der gesamte Topic-Name mit jeder Nachricht kommuniziert wird, sollte dieser so kurz wie möglich gehalten werden, um die Nachrichtengröße gering zu halten. [41, p. 13]

Um dem Empfänger einer Nachricht zu signalisieren, wie der Inhalt einer Nachricht zu verarbeiten ist, werden über das Element *message_type*, wie Tabelle 3.1 zu entnehmen, verschiedene Nachrichtentypen definiert. Diese dienen im Wesentlichen der Verwaltung des Lebenszyklus eines Gerätes, der Kommunikation von erfassten Sensordaten und dem Empfang von Steuerungsdaten. Zusätzlich wird über das Präfix eines Nachrichtentyps der Gerätetyp differenziert. Mit Ausnahme des Nachrichtentyps *STATE*, indiziert das *Präfix "D"* den Gerätetyp Device und das *Präfix "N"* den Gerätetyp EoN-Node für alle Nachrichtentypen. [41, p. 14] Im Folgenden wird das Präfix einer Nachricht ausgelassen, da die Funktionalität für *EoN* und *EoN* analog deiniert ist.

Nachrichtentyp	Beschreibung
NBIRTH	Senden von Birth-Zertifikaten zur Bereitstellung von Node-Informationen.
NDEATH	Senden von Death-Zertifikaten bei unerwarteten Verbindungsabbrüchen.
NDATA	Senden von, durch eine Node erfasste Sensormetriken.
NCMD	Senden von Steuerungsbefehlen an eine Node.
DBIRTH	Senden von Birth-Zertifikaten zur Bereitstellung von Device-Informationen.
DDEATH	Senden von Death-Zertifikaten bei unerwarteten Verbindungsabbrüchen.
DDATA	Senden von, durch ein Device erfasste Sensormetriken.
DCMD	Senden von Steuerungsbefehlen an ein Device.
STATE	Senden von Birth- und Death-Zertifikaten einer IIoT-Host-Applikation.

Tabelle 3.1: Nachrichtentypen der Sparkplug Specification [41, p. 14]

Nachrichten des Typs *BIRTH* werden direkt nach erfolgreicher Initialisierung des MQTT-Clients versandt. Diese dienen der automatischen Bereitstellung eines Gerätes im Netzwerk und enthalten alle Informationen die zur Interaktion mit einem Gerät notwendig sind wie beispielsweise Metadaten, Sensormetriken, Aktorfunktionalität und weitere konfigurationsrelevante Metriken die ein Gerät bereitstellt. Konträr zu Nachrichten des Typs BIRTH dienen *DEATH-Nachrichten* als Indikator, dass die Verbindung eines Gerätes unterbrochen wurde oder das dieses vom System abgemeldet wurde. DEATH-Nachrichten werden bei der Initialisierung einer MQTT-Session per *LWT* festgelegt. Dementsprechend ist der Message-Broker verantwortlich Death-Zertifikate zuzustellen, sobald dieser registriert, dass die Verbindung einer Session unterbrochen wurde. Alle Daten die ein Sparkplug kompatibles Gerät erfasst, werden über eine *DATA-Nachricht* publiziert und alle Steuerungsbefehle werden über eine *CMD-Nachricht* empfangen. [41, p. 14ff]

Der Nachrichteninhalt ist durch eine einheitliche Struktur festgelegt, die es ermöglicht beliebige Werte unterschiedlicher Typen zu übermitteln. Dabei verwendet jeder Nachrichtentyp die gleiche wie in Listing 3.2 dargestellte Struktur. Metriken repräsentieren hierbei die zu übermittelnden Werte. Die Bezeichnung einer Metrik wird durch das Attribut *name* spezifiziert, *alias* kann für eine alternative Bezeichnung verwendet werden. Weiterhin wird für jede Metrik ein Zeitstempel definiert, der den Zeitpunkt der Erstellung der Metrik beschreibt, sowie der Datentyp, den eine Metrik repräsentiert als auch den konkreten Wert selbst. Das Attribut *seq* wird zur Validierung von Nachrichten verwendet. Eine *NBIRTH*-Nachricht enthält immer die Sequenznummer 0. Jede weitere Nachricht von diesem Gerät und auch jede Nachricht von verwalteten Geräten führt diese Nummer sequenziell fort. [41, p. 44ff]

```
{
"timestamp": <timestamp>,
"metrics": [{
        "name": <metric_name>,
        "alias": <alias>,
        "timestamp": <timestamp>,
        "dataType": <datatype>,
        "value": <value>
}],
```

```
"seq": <sequence_number>
}
```

Listing 3.2: Einheitlich definierte Payload Struktur

Sparkplug stellt durch eine umfassende Spezifikation einer standardisierten Kommunkationsstruktur eine solide Grundlage für die Kommunikation zwischen IoT-Anwendungen und IoT-Geräten. Durch die Abstraktion von konkreter Hardware eignet sich das Konzept gut um verschiedenste Geräte in ein System einzubinden. Die Anbindung von Geräten bedingt jedoch, dass diese bereits umfänglich im Voraus konfiguriert sind und in der Lage sind alle Metadaten selbst bereitzustellen.

Die Umsetzung der gesamten Sparkplug Spezifikation gestaltet sich als sehr umfangreich und komplex, da sowohl auf Seite der EoN-Geräte als auch auf Seite der Application-Nodes und IIoT Host-Applications entsprechende Sparkplug fähige MQTT-Client Anwendungen umgesetzt werden müssen. Jedoch eignen sich die die in der Sparkplug Spezifikation eingeführten Konzepte als Orientierung zur Konzeption einer simplifizierten Kommunikationsgrundlage für die Forschung im Rahmen des CiTe-Testfeldes.

3.3.2 Homie Convention

Die *Homie Convention* baut auf der *OASIS MQTT-Spezifikation* auf, indem sie diese um weitere Aspekte erweitert. Kernaspekte sind die Definition einer Topic-Struktur und damit einhergehend Konventionen zur Einordnung von entsprechenden IoT-Geräten und deren bereitgestellten Daten. Weiterhin definiert die Spezifikation einen Lebenszyklus für Geräte, durch den der Status eines Gerätes in einem entsprechenden IoT-System widerspiegelt wird. Grundsätzlich unterscheidet die Spezifikation zwischen Geräten, die eine Client Implementierung gemäß der Spezifikation implementieren und Homie kompatiblen Controllern. Clients publizieren Nachrichten auf den im Folgenden vorgestellten Topics. Controller repräsentieren Komponenten, die automatisch bereitgestellte Geräte erkennen und in der Lage sind, mit diesen zu kommunizieren. Essentielle Voraussetzung für die gesamte Kommunikation, stellt die Verwendung des *QoS-Level 1* und Nachrichten mit gesetztem *retained-Flag* dar.
homie:doc

```
homie/device-ID/node-ID/property
```

Listing 3.3: Topic-Struktur der Homie-Convention

Die in Listing 3.3 aufgeführte Topic-Struktur definiert eine Topologie von verfügbaren Komponenten. *Devices* stellen physische Geräte die aus mehreren weiteren Komponenten, den *Nodes* zusammengesetzt sind. Jede Node stellt wiederum ein oder mehrere *Properties* zur Verfügung. Eine Property stellt dabei eine Eigenschaft eines Gerätes dar, die eine konkrete Datenquelle abbildet. Den letzten Baustein stellen *Attributes* dar, wobei jeder zuvor vorgestellten Komponente ein oder mehrere Attribute zugeordnet werden. Attribute sind per Konvention festgelegt und repräsentieren Metadaten der einzelnen Komponenten Device, Node und Property. Diese dienen in erster Linie zur automatischen Bereitstellung von Geräten in einem durch die Homie Convention gestützten System. Alle Attribute nutzen das Symbol $ als Präfix ihres Topicnamens, dementsprechend sollten Topic Namen nicht mit $ beginnen, da es sich um ein reserviertes Zeichen handelt.[48, ch. 7]

homie / device-ID / $device-attribute	
$homie	Aktuell genutzte Version der Homie Convention.
$name	Lesbarer Name eines Device.
$state	Ein Status, definiert durch die Homie Convention definierten Lebenszyklus.
$nodes	Alle Nodes die ein Device bereitstellt. Bei mehreren Devices werden diese Komma separiert dargestellt.
$extensions	Unterstützte Erweiterungen. Bei mehreren Erweiterungen, werden diese Komma separiert dargestellt.

Tabelle 3.2: Durch die Homie Convention definierte Device-Attribute [48, sec. 7.1]

Tabelle 3.2 führt alle verfügbaren Attribute eines Devices auf. Diese enthalten Informationen über die verwendete Version der Homie-Spezifikation, die Angabe eines lesbaren Namens eines Geräts. Die Homie-Spezifikation erlaubt die Implementierung von Erweiterungen, diese werden bei Verwendung anhand ihrer Kennung über das Attribut *extensions* vermerkt. Unter dem Attribut *nodes* werden alle von einem Gerät bereitgestellten Nodes kommuniziert. [48, sec. 7.1]

Lebenszyklus	Beschreibung
init	Dieser Zustand signalisiert, dass ein Gerät mit dem MQTT-Broker verbunden wurde, aber noch nicht betriebsbereit ist. Der init-Zustand ist optional und ist reserviert für Geräte deren Initialisierung längere Zeit benötigt.
ready	Hat ein Gerät alle Attribute publiziert und ist betriebsbereit wird dies mit dem ready-Zustand signalisiert.
disconnected	Wurde ein Geräte ordentlich abgemeldet, signalisiert ein Client dies mit diesem Zustand.
sleeping	Signalisiert, dass ein Gerät temporär nicht verfügbar ist.
lost	Verliert ein Gerät, beispielsweise durch Ausfälle des Netzwerks die Verbindung, muss dieser Zustand publiziert werden. Diese Nachricht wird bei Initialisierung als LWT-Nachricht festgelegt.
alert	Treten auf Seite eines Gerätes Fehler in der Ausführung auf muss eine Nachricht mit diesem Zustand publiziert werden um zu signalisieren, dass ein Eingreifen zur Fehlerbehebung notwendig ist.

Tabelle 3.3: Lebenszyklus von Devices [48, sec.7.1.2]

Das Attribut *state* repräsentiert den aktuellen Zustand eines Geräts gemäß des in der Spezifikation definierten Lebenszyklus. Dieser besteht wie in Tabelle 3.3 beschrieben aus sechs verschiedenen Zuständen. Diese werden entsprechend als Inhalt einer Nachricht auf dem *state* Attribut-Topic kommuniziert. [48, sec.7.1.2]

homie / device-ID / $node-attribute	
$name	Lesbarer Name einer Node.
$type	Der Typ einer Node.
$properties	Von einer Node bereitgestellte Properties. Bei Angabe mehrerer Properties werden diese Komma separiert angegeben.

Tabelle 3.4: Durch die Homie-Spezifikation definierte Node-Attribute [48, sec.7.2.1]

Alle in Tabelle 3.4 beschriebenen Attribute einer einzelnen Node sind beschreibender Natur. Darunter befinden sich analog zu den Devices ein Attribut zur Angabe eines Namens, sowie das Attribut *properties*, unter dem alle von einer Node bereitgestellten Properties angegeben werden. Das Attribut *type* dient zur genaueren Einordnung einer Node und ist ebenfalls von beschreibender Natur. [48, sec.7.2]

homie / device-ID / node-ID / property-ID / $property-attribute	
$name	Lesbarer Name des Property-Attributs.
$datatype	Der Datentyp eines Property-Attributs.
$format	Spezifikationen zur Formatierung eines Datentyps. Beispielsweise ein Datumsformat.
$settable	Spezifiziert ob ein Property gesetzt werden kann. Das ist der Fall bei von Aktoren bereitgestellten Properties.
$retained	Indikator, ob Nachrichten mit gesetztem Retained-Flag versendet werden.
$unit	Die Einheit einer Property. Beispielsweise C° bei einem Temperatursensor.

Tabelle 3.5: Durch die Homie Convention definierte Property-Attribute[48, sec.7.3.1]

Auf der untersten Hierarchiebene werden, wie in Tabelle 3.5 aufgeführt, alle Metadaten einer Property Komponente angegeben. Da über ein Property erfasste Sensordaten kommuniziert werden, beschreiben die Attribute die in diesem Kontext erfassten Daten. Der Datentyp wird mit Hilfe des Attributs *datatype* beschrieben. Die Attribute *format* und *unit* beschreiben die Darstellung und die konkrete Einheit der Daten. Aktoren werden im Rahmen der Konvention nicht konkret als solche gekennzeichnet. Die Identifizierung bezieht sich nicht auf ein konkretes Gerät sondern findet auf Ebene der Properties statt.[48, sec.7.3] Dabei wird unterschieden ob ein Property *settable* ist oder nicht. Ist das Attribut gesetzt, können Steuerungsbefehle kommuniziert werden indem diese über den Kanal *homie/device-ID/node-ID/property-ID/set* publiziert werden. [48, sec.7.3.2]

Die Homie Spezifikation bildet eine einfache Grundlage für einen strukturierten Aufbau von IoT-Anwendungen deren Kommunikation auf Basis des MQTT-Protokolls erfolgt. Ein wesentlicher Kritikpunkt stellt jedoch die fehlende Möglichkeit dar, Geräte

logisch zu gruppieren. Aus diesem Grund eignet sich die Spezifikation eher für IoT-Anwendungsfälle im Bereich Home-Automation, da hier keine komplexe Gruppierung von Geräten erforderlich ist. Weiterhin bietet die Spezifikation nur bedingt Raum für eine flexiblere Erweiterung von weiteren Metadaten. Diese können zwar als weitere Attribute angegeben werden, jedoch stellt jedes weitere Attribut einen weiteren Kommunikationskanal dar, was bei einer Skalierung, beispielsweise in der Größenordnung einer Smart City, zu einem nicht unerheblichen Overhead in der Verarbeitung der Nachrichten auf den zahlreichen Kommunikationskanälen führt. Dennoch stellen Konzepte wie der vorgestellte Lebenszyklus ein durchdachtes System dar, welches als Orientierung für die Konzeption einer Kommunikationsstruktur verwendet werden kann.

3.4 Etablierung von IoT-Standards

Durch die stetig zunehmende Relevanz von IoT- als auch IIoT-Anwendungen und Geräten sind in den letzten Jahren unzählige Initiativen und Arbeitsgruppen entstanden, mit dem gemeinsamen Ziel eine Plattform für IoT-Anwendungen zu schaffen. Die breite Fragmentierung und Inkonsistenz von gestellten Anforderungen, stellt ein großes Problem in diesem Prozess dar. Mit diesem Thema befasst sich das vom International Electrotechnical Commission (IEC) ins Leben gerufene Whitepaper, welches in Koorperation mit SAP und dem Fraunhofer Institute for Applied and Integrated Security (AISEC) erarbeitet wurde. [84] Gerade im privaten Sektor besteht Unklarheit welche Aspekte sich zur Standardisierung eignen, da je nach Szenario die Wettbewerbsfähigkeit in der Industrie gefährdet wird. Dabei spielt ebenso der Datenschutz eine große Rolle. IoT-fähige Geräte sind in der Lage eine große Bandbreite an Daten zu erfassen und zu kommunizieren. Dabei existieren Interessenkonflikte zwischen Herstellern von IoT-Hard- und Software und den entsprechenden End-Nutzern. Es existiert kein einheitlicher Konsens darüber wie mit Daten Domänenübergreifend verfahren werden soll oder darf. Ebenfalls im Konflikt mit der Etablierung von einheitlichen Standards stehen oft geopolitische Interessen. Entsprechende regulatorische Maßnahmen können sich länderabhängig unterscheiden. Diese Maßnahmen wird angeheftet, dass sie sich meist nachteilig auf einen Standardisierungsprozess auswirken, da etwaige regierungsspezifische Anforderungen, aufgrund ihrer meist restriktiven Natur, Einschränkungen auf die Entwicklung und Forschung im privaten Sektor erwirken und somit der Erforschung von Standards entgegenwirken. Neben einigen Aspekten die sich negativ auf die Entwicklung von IoT-Standards auswirken, sind jedoch auch diverse positive Aspekte zu beobachten, die diesem Trend entgegenwirken. Viele der existierenden Initiativen zeigen zunehmend Interesse an der Zusammenarbeit. Dies fördert in erster Linie die Verringerung der weiteren Fragmentierung, indem involvierte Initiativen und Organisationen vermehrt Anforderungen definieren statt neue Standards zu erarbeiten. Diese Anforderungen werden entsprechend zentral von relevanten Standard Defining Organization (SDO)s gesammelt und in existierende Standards integriert. Zunehmend beschäftigen sich immer mehr Initiativen und Organisationen aktiv mit der Analyse der internationalen Standardisierung im IoT-Bereich um bestehende Probleme zu ermitteln und diesen entgegenzuwirken.

Für die zukünftige Entwicklung differenziert das IEC die Entwicklung von horizontalen Standards und vertikalen Standards. Horizontale Standards sollen eine universelle Grundlage über geopolitische Grenzen und Fachbereiche hinaus auf internationaler Ebene bilden. Die vertikale Standardisierung, welche zugleich auch einer der Gründe für die breite Fragmentierung der Standards bildet, soll zukünftig lediglich für den Fachbe-

reich benötigte Anforderungen definieren, mit deren Hilfe auf existierenden Standards aufgebaut werden kann. [84, p. 87ff]

4 Analyse

In einer Smart City können Prozesse durch Sensoren und Aktoren automatisiert werden. Um dies zu ermöglichen, müssen diese auf einfache Weise in die Infrastruktur der Smart City integriert werden können. Verschiedene Akteure sind in der Lage, Daten von Sensoren zu prozessieren oder Systeme durch das Senden von Steuerungsbefehlen zu verwalten. Um eine manuelle Buchführung von angeschlossenen Geräten zu vermeiden, ist eine Verwaltungsinstanz erforderlich, die interessierten Anwendungen die Kommunikation zu entsprechenden Geräten, sowie die Software gestützte Verwaltung von Geräten und deren Metadaten, ermöglicht. Ziel dieser Arbeit ist es durch Konzeption und Implementierung eines entsprechenden Systems, die bestehende Infrastruktur des CiTe-Testfelds um diese Funktionalitäten zu erweitern. Ein solches System sollte nahtlos in die bestehende Architektur integrierbar sein. Durch die gegebene Architektur des Clusters bietet sich an dieser Stelle eine Microservice Architektur an, da einzelne Services auf einfache Weise im Cluster bereitgestellt werden können und das Gesamtsystem so eine hohe Modularität und Wartbarkeit fördert. Zentrale Anforderungen umfassen somit eine möglichst automatische Erfassung von neu angeschlossenen Geräten, sowie eine Möglichkeit zur Verwaltung von angeschlossenen Geräten und deren Metadaten, als auch deren Persistierung in einem geeigneten Datenhaltungssystem zur späteren Abfrage.

4.1 Anwendungsfälle

Im Folgenden werden verschiedene Anwendungsfälle aufgeführt, welche bestmöglich alle denkbaren Szenarien in Bezug auf Nutzung und Weiterentwicklung des zu entwickelnden Services abbilden. Alle im Nachfolgenden ermittelten Anforderungen wurden durch Absprache mit dem Team des Labors für Verteilte Systeme ermittelt.

Registrierung eines neuen Sensors oder Aktors

Neue Sensoren oder Aktoren sollen auf möglichst einfache Weise in das System eingebunden werden. Das System soll die neu angeschlossene Komponente weitestgehend automatisiert identifizieren und entsprechende Metadaten in einem Datenhaltungssystem persistieren. Weiterhin soll die Möglichkeit bestehen Metadaten nachträglich manuell zu konfigurieren. Diese Funktionalität wird beispielsweise benötigt, wenn keine vollständig automatisierte Erfassung möglich ist. Die Bereitstellung der relevanten Informationen für ein neu registriertes Gerät erfolgt über einen, durch den Message-Broker zur Verfügung gestellten Nachrichtenkanal.

Entfernen eines Sensors oder Aktors

Bereits in das System eingebundene Sensoren oder Aktoren sollen bei Bedarf auf möglichst einfache Weise entfernt werden können. Das System soll in der Lage sein zu erkennen wenn ein Sensor physisch entfernt wurde. Weiterhin sollte ein potentieller Nutzer in

der Lage sein zu prüfen, ob ein Sensor sicher entfernt werden kann. Ein Sensor oder Aktor kann sicher entfernt werden, wenn es keine aktiven Consumer gibt, das heißt es gibt keine Services die aktiv den vom Sensor veröffentlichten Datenstrom konsumieren. Sollte ein Sensor im Falle eines Defektes ausfallen, dann soll diese Information im entsprechenden Eintrag im Datenhaltungssystem markiert werden.

Dynamische Erweiterung von Metadaten

Aufgrund der sehr heterogen geprägten Sensor-(Aktor-)Landschaft ist anzunehmen, dass die bestehende Datenstruktur zur Abbildung der Metadaten nicht immer ausreichen wird um alle Informationen darzustellen. Deshalb sollte die Möglichkeit bestehen, die Metadatenstruktur möglichst dynamisch, ohne hohen Implementierungsmehraufwand um ein neues Datum zu erweitern.

Abruf und Manipulation von Metadaten

Über eine entsprechende universelle Schnittstelle (z.B. REST) sollen gespeicherte Metadaten zu einzelnen oder mehreren Sensoren oder Aktoren abrufbar sein. Hierbei sollte die Möglichkeit bestehen Filterkriterien anzugeben um ausschließlich gewünschte Informationen zu erhalten. Weiterhin sollte auch die Möglichkeit bestehen, bereits persistierte Daten zu ändern.

Deployment

Die Anwendung sollte auf möglichst einfache Weise in einem Cluster bereitgestellt werden. Dies umfasst unter anderem auch die Konfiguration des Services in einer Cluster Umgebung.

Horizontale Skalierung

Bei Bedarf soll der Service auf einfache Weise horizontal skalierbar sein. Wünschenswert ist es, dass dies automatisch passiert. So könnten bei hoher Anfragelast weitere Instanzen initiiert werden, auf welche die Last verteilt wird.

Monitoring und Logging

Damit im Clusterbetrieb gewährleistet ist, dass alle Serviceinstanzen fehlerfrei funktionieren, ist es erforderlich entsprechende Metriken nach außen bereitzustellen. Dies sind beispielsweise Informationen über CPU- und Speicherauslastung, aber auch Network-Traffic. Solche Metriken können im gegebenen Kontext über einen entsprechenden Endpunkt zur Verfügung gestellt werden, welcher von kompatiblen Monitoring Anwendungen konsumiert werden kann. Jeglicher, im Cluster betriebene Microservice, sollte zumindest rudimentäres Logging über die Standardstreams. stdout und stderr implementieren.

Weiterentwicklung des Service

Für neu in das Projekt eingebundene Entwickler soll es leicht möglich sein den Service entsprechend bei Bedarf zu erweitern. Im gegebenen Kontext handelt es sich bei neuen Entwicklern primär um Studenten, welche im Rahmen von Abschluss- oder Projektarbeiten am CiTe-Testfeld arbeiten. Daher sollte der Aufbau des Service möglichst modular

erfolgen, um eine einfach Erweiterbarkeit zu gewährleisten. Weiterhin sollte analog auch eine entsprechend umfassende Dokumentation des zu entwickelnden Services betrieben werden.

4.2 Anforderungen

Aus den beschriebenen Szenarien lassen sich vier Kernelemente ableiten, die einen Schwerpunkt dieser Arbeit bilden.

- **Broker (BK)**:Der Message Broker, im Kontext des DSL-Clusters RabbitMQ mit aktiviertem MQTT-Plugin, dient als primäres Mittel zur Kommunikation im Gesamtsystem.
- **Datenhaltungssystem (DB)**: Es wird ein geeignetes Datenhaltungssystem benötigt, welches für den Clusterbetrieb geeignet ist. Hierbei gilt zu evaluieren inwieweit existierende Systeme den Anforderungen genügen.
- **Microservice (MS)**: Ein Service zur Verwaltung von Sensor- und Aktorgeräten muss konzipiert werden. Dieser wird eine zentrale Komponente der Microservicearchitektur zur Arbeit mit IoT kompatiblen Geräten im CiTe-Testfeld darstellen.
- **Raspberry Pi (PI)**: Raspberry Pi Computer dienen als Schnittstelle um Sensoren und Aktoren im Gesamtsystem bereitzustellen. Diese sollen über den Message Broker Daten publizieren oder mit anderen im Cluster betriebenen Anwendungen kommunizieren.

Alle nachfolgenden Anforderungen werden in funktionale und nicht-funktionale Anforderungen aufgeteilt, entsprechend der oben beschriebenen Kernelemente kategorisiert und anlehnend an das Prinzip des MosCow-Priorisierungsschemas, in *Must-Have (MUST)* und *Should-Have (SHOULD)* eingeteilt. [58] Must-Have Anforderungen sind essentielle Anforderungen und müssen im Rahmen der technologischen Umsetzbarkeit umgesetzt werden. Should-Have Anforderungen sind nicht erfolgsentscheidend für die Umsetzung der Anwendung, stellen jedoch Anforderungen dar die entscheidend zur Qualität der Anwendung beitragen und dementsprechend, insofern nicht im Rahmen dieser Arbeit umsetzbar, für die zukünftige Implementierung relevant sind.

4.2.1 Message-Broker

Der Message-Broker steht bereits als Cluster-Deployment zur Verfügung und ist somit bereits fester Bestandteil des CiTe-Testfelds. Dementsprechend sind die in Tabelle 4.1 dargestellten Anforderungen bis auf *FR.BK.001* und *FR.BK.002* bereits ausreichend (siehe Tabelle 7.1) erfüllt.

Da die MQTT-Spezifikation keine standardisierte Topic-Struktur festlegt, wird die Konzeption einer solchen als Anforderung festgelegt. In diesem Zusammenhang spezifiziert die offizielle MQTT-Spezifikation ebenfalls kein festgelegtes Nachrichtenformat. Um eine einheitliche Kommunikation über alle im Testfeld bereitgestellten Anwendungen hinweg zu ermöglichen, soll ebenfalls ein einheitliches Nachrichtenformat entworfen werden. Die zu konzipierende Topic-Struktur und jeweiligen Nachrichtenformate, müssen dementsprechend von allen relevanten Anwendungen verwendet werden.

ID	Anforderung	Priorität
	Funktionale Anforderungen	
FR.BK.001	Spezifikation einer einheitlichen Topic-Struktur.	*MUST*
FR.BK.002	Spezifikation eines einheitlichen Nachrichtenformats.	*MUST*
FR.BK.003	Logging zur Bereitstellung von Anwendungsinformationen.	*SHOULD*
FR.BK.004	Aggregation von Log-Nachrichten in der Clusterumgebung über separate Monitoring-Anwendung.	*SHOULD*
FR.BK.005	Bereitstellung von Metriken bezüglich der Ressourcen-auslastung über entsprechende Endpunkte.	*SHOULD*
	Nicht-Funktionale Anforderungen	
NFR.BK.001	Bereitstellung über Kubernetes.	*MUST*
NFR.BK.002	Unkompliziertes Deployment.	*MUST*
NFR.BK.003	Möglichst geringe Latenz bei Serveranfragen.	*MUST*
NFR.BK.004	Hohe Verfügbarkeit im Clusterbetrieb.	*MUST*
NFR.BK.005	Hohe Resilienz im Clusterbetrieb.	*MUST*
NFR.BK.006	Möglichkeit zur horizontalen Skalierung.	*MUST*

Tabelle 4.1: Anforderungen an die Message-Broker-Anwendung

4.2.2 Datenhaltungssystem

Ein geeignetes Datenhaltungssystem wird benötigt um alle relevanten Metadaten zu persistieren und um auf diese zuzugreifen. Dementsprechend soll das zu verwendende System eine flexible Möglichkeit anbieten, um Daten in unterschiedlicher Granularität abzufragen. Um die Anwendung auf zuverlässige Weise in einem Cluster-Umfeld zu betreiben, sollte diese Mechanismen implementieren, um einen zuverlässigen Betrieb zu ermöglichen. Alle erforderlichen Aspekte sind Tabelle 4.2 zu entnehmen.

ID	Anforderung	Priorität
	Funktionale Anforderungen	
FR.DB.001	Flexible Abfrage von Daten über geeignete Abfragesprache.	*MUST*
FR.DB.002	Logging zur Bereitstellung von Anwendungsinformationen.	*SHOULD*
FR.DB.003	Aggregation von Log-Nachrichten in der Clusterumgebung über separate Monitoring-Anwendung.	*SHOULD*
FR.DB.004	Bereitstellung von Metriken bezüglich der Ressourcenauslastung über entsprechende Endpunkte.	*SHOULD*
	Nicht-Funktionale Anforderungen	
NFR.DB.001	Bereitstellung über Kubernetes.	*MUST*
NFR.DB.002	Unkompliziertes Deployment.	*MUST*
NFR.DB.003	Möglichst geringe Latenz bei Serveranfragen.	*MUST*
NFR.DB.004	Hohe Verfügbarkeit im Clusterbetrieb.	*MUST*
NFR.DB.005	Hohe Resilienz im Clusterbetrieb.	*MUST*
NFR.DB.006	Möglichkeit zur horizontalen Skalierung.	*MUST*
NFR.DB.007	Gewährleisten einer starken Konsistenz der Daten im Clusterbetrieb.	*MUST*

Tabelle 4.2: Anforderungen an das Datenhaltungssystem

4.2.3 Microservice zur Metadatenverwaltung

Peripheriegeräte im Sinne von Sensoren und Aktoren die sich außerhalb des Clusters befinden, werden über ein Raspberry Pi-Computer angebunden. Entsprechende Informationen sollen dem Cloud-Backend durch Kommunikation über einen Message-Broker zur Verfügung gestellt werden. Ein im Cluster bereitgestellter Microservice soll in der Lage sein diese Informationen zu empfangen und in einem Datenhaltungssystem zu persistieren. Eine universelle Schnittstelle soll die Erstellung und Manipulation von Metadaten ermöglichen. Dabei soll auch die Kommunikation mit Peripheriegeräten über den Message-Broker möglich sein, um Geräte remote gesteuert zu registrieren oder zu löschen.

Die Anwendung soll als skalierbarer Microservice im *DSL-Cluster* zur Verfügung gestellt werden. Um dies zu ermöglichen, soll der Prozess zum Deployment und der horizontalen Skalierung der Anwendung möglichst einfach gestaltet werden. Letztlich steht die Verwaltung der Peripheriegeräte sowie der zuverlässige Betrieb in einer Cluster-Umgebung im Fokus. Alle Anforderungen sind im Detail Tabelle 4.3 zu entnehmen.

ID	Anforderung	Priorität
	Funktionale Anforderungen	
FR.MS.001	Bereitstellen einer universellen Schnittstelle zur Ausführung von CRUD-Operationen auf Metadaten.	*MUST*
FR.MS.002	Backendseitige Registrierung von Sensoren über MQTT-Kommunikation.	*MUST*
FR.MS.003	Backendseitiges Abmelden von Sensoren über MQTT-Kommunikation.	*MUST*
FR.MS.004	Bidirektionale Kommunikation über eine festgelegte Topic-Struktur.	*MUST*
FR.MS.005	Persistierung und Abruf von Metadaten im Datenhaltungssystem via CRUD-Operationen.	*MUST*
FR.MS.006	Automatisches Erstellen und Löschen von Metadaten, anhand über den Message Broker empfangener Geräteinformationen.	*MUST*
FR.MS.007	Bereitstellen eines geeigneten Datenmodells um entsprechende Gerätedaten abzubilden.	*MUST*
FR.MS.008	Synchronisation zwischen mehreren Microservice-Instanzen, Empfang von zu persistierenden Metadaten.	*MUST*
FR.MS.009	Integration einer Zugriffsteuerung für die universelle Schnittstelle, um den Zugriff auf Ressourcen ggf. einzuschränken.	*SHOULD*
FR.MS.010	Logging zur Bereitstellung von Anwendungsinformationen.	*SHOULD*
FR.MS.011	Bereitstellung von Metriken bezüglich Ressourcenauslastung über einen separaten Endpunkt.	*SHOULD*
FR.MS.012	Aggregation von Log-Nachrichten in der Clusterumgebung über separate Monitoring-Anwendung.	*SHOULD*
	Nicht-Funktionale Anforderungen	
NFR.MS.001	Ermöglichen einer externalisierten Konfiguration im Cluster-Deployment-Kontext.	*MUST*
NFR.MS.002	Flexible Erweiterbarkeit der Anwendung.	*MUST*
NFR.MS.003	Bereitstellung über Kubernetes.	*MUST*
NFR.MS.004	Unkompliziertes Deployment.	*MUST*
NFR.MS.005	Möglichst geringe Latenz bei Serveranfragen.	*MUST*
NFR.MS.006	Hohe Verfügbarkeit im Clusterbetrieb.	*MUST*
NFR.MS.007	Hohe Resilienz im Clusterbetrieb.	*MUST*
NFR.MS.008	Möglichkeit zur horizontalen Skalierung.	*MUST*

Tabelle 4.3: Anforderungen an die Backend-Implementierung der Metadatenverwaltung

4.2.4 Raspberry Pi - Sensoranbindung

Als verwaltende Instanz bildet ein Raspberry Pi-Computer mit der GrovePi+-Erweiterung, Sensoren und Aktoren in das Gesamtsystem ein. Zur Realisierung dieser Funktionalität muss dieser mit dem Message-Broker kommunizieren und eine leicht verwendbare Möglichkeit zur Verfügung stellen, angeschlossene Geräte anzusteuern oder auszulesen. Weiterhin spielt die Konfiguration der Sensoren eine zentrale Rolle. Hierzu soll, wie Tabelle 4.4 zu entnehmen, die Konfiguration über eine manuell bereitgestellte Konfigurationsdatei erfolgen. Nach erfolgreicher Initialisierung, kommunizieren alle Geräte relevante Metadaten, um sich dem Backend-System bekannt zu machen.

Weiterhin soll nach Möglichkeit, Funktionalität zur vollautomatischen Erfassung von Sensoren und Aktoren nach Anschluss an das GrovePi+-Board umgesetzt werden. Dies schließt unter anderem die Erfassung des Gerätetyps[1], die Art der Signale die ein Gerät verarbeiten kann[2] und der konkrete Hardware-Pin an den ein Gerät angeschlossen wurde.

ID	Anforderung	Priorität
	Funktionale Anforderungen	
FR.PI.001	Speichern der Hardwarekonfiguration von Sensoren und Aktoren.	*MUST*
FR.PI.002	Automatische Bereitstellung von angeschlossenen und konfigurierten Sensoren im Netzwerk über entsprechende Topics.	*MUST*
FR.PI.003	Automatische Registrierung und Abmeldung von angeschlossenen, jedoch noch nicht konfigurierten Sensoren, anhand über den Message-Broker empfangener Befehle.	*MUST*
FR.PI.004	Manuelle Konfiguration von angeschlossenen Geräten über eine Konfigurationsdatei.	*MUST*
FR.PI.005	Angeschlossene Sensoren können erfasste Sensormetriken senden.	*MUST*
FR.PI.006	Angeschlossene Aktoren können Steuerbefehle empfangen.	*MUST*
FR.PI.007	Logging zur Bereitstellung von Anwendungsinformationen.	MUST
FR.PI.008	Automatische Erkennung und Konfiguration von angeschlossenen Sensoren.	*SHOULD*
	Nicht-Funktionale Anforderungen	
NFR.PI.001	Flexible Erweiterbarkeit der Anwendung.	*MUST*
NFR.PI.002	Operabilität auf Raspberry-Pi.	*MUST*

Tabelle 4.4: Anforderungen an die Anwendung zur Anbindung von Sensoren

[1] Hierbei soll ermittelt werden, ob es sich um einen Sensor oder Aktor handelt.
[2] Grundlegend werden Grove-Sensoren in digitale und analoge Geräte kategorisiert.

5 Konzeption

Um die zuvor ermittelten Anforderungen zu erfüllen, wird zunächst eine geeignete Architektur konzipiert. Das Konzept soll nahtlos in die gegebene Infrastruktur des CiTe-Testfeldes integriert werden oder diese gegebenenfalls erweitern. Die im Nachfolgenden durchgeführte Konzeption ist abgesehen von der Clusterverwaltung durch Kubernetes und Kommunikation über den Message Broker RabbitMQ, technologieunabhängig zu betrachten.

5.1 Mehrschichtige Architektur

Eine mehrschichtige Architektur ermöglicht es im Rahmen der Entwicklung einer Softwarearchitektur bestimmte Funktionalitäten voneinander abzugrenzen. Diese Abgrenzung kann in unterschiedlicher Granularität und unterschiedlichem Abstraktionsgrad erfolgen. Um eine bessere Einordnug im Kontext der gegebenen Infrastruktur zu ermöglichen, wird das Gesamtsystem, welches durch das CiTe-Testfeld abgebildet wird, zunächst verschiedene Architekturschichten abstrahiert.

In der Anwendungsentwicklung werden diese über eine Differenzierung und entsprechende Kategorisierung der zu implementierenden Komponenten, anhand deren Aufgabe im System definiert. Bei klassischen Anwendungen wird häufig zwischen Präsentationsschicht, Anwendungsschicht und Datenschicht unterschieden. Zur besseren Einteilung des Clusters wird eine erweiterte Schichtenarchitektur genutzt, die die Datenschicht durch die Infrastrukturschicht ersetzt. Üblicherweise beinhaltet die Datenschicht das Datenhaltungssystem sowie die Implementierung einer Schnittstelle, die eine Interaktion mit dem Datenhaltungssystem ermöglicht.

In modernen verteilten Systemen, in denen immer mehr auf Middleware, wie Nachrichtenbroker oder Tools zur Überwachung und Log-Aggregation gesetzt wird, ist keine klare Einteilung über das klassische 3-Schichtenmodell möglich. Aus diesem Grund führt Evans statt der üblichen Datenschicht (Data Layer), die Infrastrukturschicht (Infrastructure Layer) ein. Diese dient der Kategorisierung von Middleware-Anwendungen, Datenhaltunssystemen und weiteren infrastrukturrelevanten Anwendungen. [39, p.105 ff.]

Im Rahmen des CiTe-Testfeldes fasst die Komponente *Model City* alle Sensoren und Aktoren zusammen die über Raspberry-Pi-Geräte angesteuert werden können. Sensoren, Aktoren oder diese verwaltende, als Gateways, bezeichneten Geräte befinden sich am Rande eines Netzwerkes. Aus diesem Grund werden diese auch als Edge of Network Devices bezeichnet. Zur Einordung von EoN-Geräten wird die Schichtenarchitektur um eine letzte in IoT Architekturen etablierten Schicht, der *Wahrnehmungsschicht* (Perception Layer) erweitert. [83, p.15 f] Diese Schicht fasst alle *EoN-Geräte* und Anwendungen zusammen, wodurch sich eine Kategorisierung durch die nachfolgenden Schichten ergibt:

- **Präsentationsschicht:** Diese Schicht beinhaltet alle Clientanwendungen, die losgelöst von einer Serveranwendung über eine universelle Schnittstelle mit dieser kommunizieren.

- **Anwendungsschicht:** Auf dieser Schicht befinden sich alle Anwendungen, die eine bestimmte Geschäftslogik implementieren.
- **Infrastrukturschicht:** Die Infrastrukturschicht beinhaltet Datenbanken, Middleware wie Nachrichtenbroker, oder Software zur Überwachung von Anwendungen. Auf einer abstrakteren Ebene können in der Konzeption einer Anwendung dieser Schicht auch konkrete Spezifikationen zugeordnet werden. Eine Spezifikation die definiert wie Anwendungen über ein bestimmtes Nachrichtenprotkoll kommunizieren, ist ebenso in dieser Schicht einzuordnen.

Gemäß dieser Einordnung kann das CiTe-Testfeld wie Abbildung 5.1 zu entnehmen, in diese Schichten aufgeteilt werden. Alle Sensoren und Raspberry Pi-Geräte, befinden sich auf der Wahrnehmungsschicht. Jegliche Frontend-Anwendungen sind der *Präsentationsschicht* zuzuordnen. Beide genannten Layer kommunizieren über universelle Schnittstellen mit den Anwendungen innerhalb des Clusters. Die *Anwendungsschicht* und *Infrastrukturschicht* sind somit beide innerhalb des Clusters zu lokalisieren. Letztere schließt alle infrastrukturrelevanten Anwendungen ein. Alle weiteren Anwendungen sind auf der Anwendungsschicht einzuordnen.

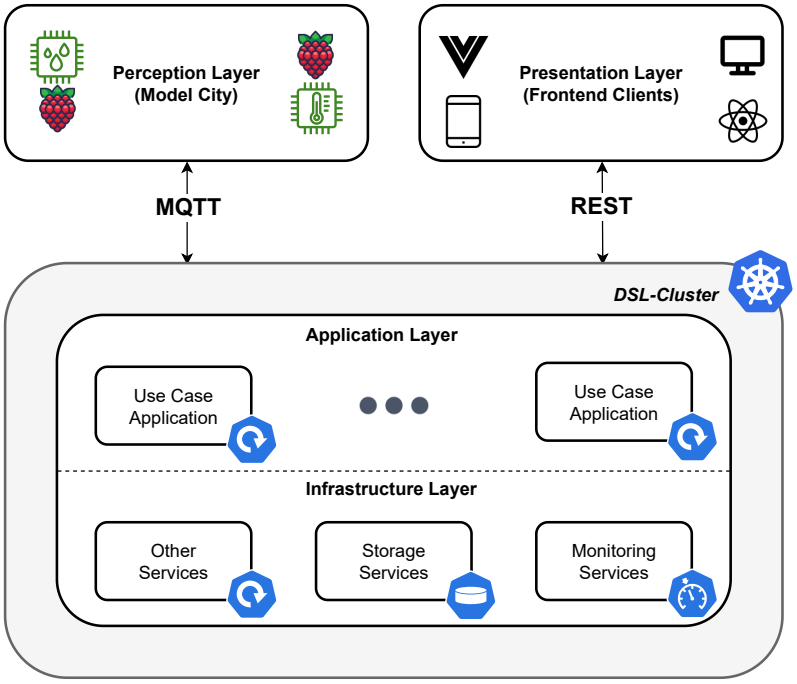


Abbildung 5.1: Architektur des CiTe-Testfeldes unterteilt in Schichten

5.2 Infrastructure Layer

5.2.1 Spezifikation einer standardisierten Kommunikation

Machine to Machine Kommunikation via MQTT erlaubt es einer Anwendung selbst zu bestimmen auf welchem Topic sie Daten sendet oder empfängt. In vielen IoT-Anwendungen hat sich dementsprechend die Verwendung einer hierarchischen Topic Namensgebung etabliert. Diese unterscheidet sich jedoch von Hersteller zu Hersteller. Eine solche hierarchische Topic Struktur wurde in bisherigen Arbeiten nicht oder nur rudimentär berück-

sichtigt. Um zukünftigen Arbeiten im Kontext des CiTe-Testfeldes eine entsprechende Kommunikationsstruktur anzubieten wird im Folgenden, in enger Anlehnung an der in Abschnitt 3.3.1 vorgestellten Sparkplug Specification, eine solche einheitliche Spezifikation konzipiert, die speziell auf das Smart City Szenario im Rahmen des Testfeldes angepasst ist.

Zunächst findet eine Einteilung der *EoN-Devices* statt. Als *EoN-Node-Device* werden im Folgenden alle netzwerkfähigen Geräte bezeichnet, die eigenständig in der Lage sind über eine integrierte MQTT-Clientanwendung zu kommunizieren. Als zentrale Schnittstelle zur Anbindung von Sensoren und Aktoren dient die Kommunikation via MQTT-Protokoll über den Message-Broker. Bei *EoN-Node-Devices* kann es sich um Sensoren, Aktoren oder Gateways handeln. Alle Geräte, im Folgenden als *EoN-Legacy-Devices* bezeichnet, die selbst nicht netzwerkfähig sind, werden über ein Gateway verwaltet und an das Netzwerk angebunden. Im gegebenen Szenario fungiert der Raspberry Pi im Zusammenspiel mit der Grovepi+-Erweiterung entsprechend als *EoN-Node-Device* und alle Grove-Sensoren als EoN-Legacy-Device.

```
namespace / group_id / message_type / node_id / [device_id]
```

Listing 5.1: Topic Template zur standardisierten Kommunikation

Im ersten Schritt stellt das in Listing 5.1 aufgeführte Topic-Template das Kernelement der standardisierten Kommunikation dar.

- **namespace**
  Die erste Hierarchieebene dient lediglich der Zuordnung einer Anwendung. Da die Kommunikationsstruktur in erster Linie für das CiTe-Projekt entwickelt wird, wird als Namespace der Wert *cite* festgelegt.

- **group_id**
  Der Group Identifier dient der Gruppierung von Geräten. Dieser kann beispielsweise eine physische Gruppierung oder eine logische Gruppierung von Geräten abbilden.

- **message_type**
  Durch den Message Typ werden verschiedene Kommunikationskanäle einer EoN-Node oder eines EoN-Devices definiert. Wie in Tabelle 5.1 dargestellt, erfüllt jeder dadurch definierte Kommunikationskanal eine spezielle Funktion. In der *Sparkplug Specification* bereits eingeführte Nachrichtentypen werden wiederverwendet und die Typen *DCONFIG* und *NCONFIG* werden zusätzlich eingeführt.

EoN-Node	EoN-Legacy-Device	Beschreibung
NBIRTH	DBIRTH	Analog zur Sparkplug Spezifikation, wird dieser Nachrichtentyp verwendet, um Geräte im Netzwerk bereitzustellen.
NDEATH	DDEATH	Analog zur Sparkplug Spezifikation dient dieser Typ zur Signalisierung, dass ein Gerät nicht mehr verfügbar ist.
NDATA	DDATA	Erfasste Sensormetriken werden über diesen Typ kommuniziert.
NCMD	DCMD	Steuerungsbefehle werden über diesen Typ kommuniziert.
NCONFIG	DCONFIG	Dieser Typ dient als Ergänzung um einen separaten Nachrichtenkanal zur Konfiguration eines Gerätes bereitzustellen.
NSTATE	DSTATE	Dieser Typ dient als Ergänzung, um eine Statusänderung des Geräts nach dem Report-By-Exception-Prinzip zu kommunizieren.

Tabelle 5.1: Nachrichtentypen unterteilt nach Gerätetyp

- **node_id**
  Der Node Identifier ist eine *eindeutige ID* die ein *EoN-Node-Device* identifiziert. Ein Kommunikationskanal eines *EoN-Node-Device* enthält dementsprechend im Topic-Namen ein eindeutiges Identifizierungsmerkmal.
- **device_id**
  Der Device-Identifier ist eine *eindeutige ID* um *EoN-Legacy-Devices* zu identifizieren. Diese Hierarchiebene ist, abhängig vom Typ des EoN-Node-Devices, eine optionale Hierarchieebene.

Zur Einhaltung eines konsistenten Benennungsschemas für einen Topic und die einzelnen Topic-Ebenen ist es wichtig Richtlinien zur Benennung zu definieren. Damit Topic-Namen leichter lesbar sind, sollen lediglich Ziffern, kleingeschriebene Buchstaben und Bindestriche verwendet werden. Analog zu klassischen Unix Dateisystemen fungieren Schrägstriche als separierende Zeichen. Aus diesem Grund sind führende Schrägstriche zu vermeiden, da diese eine neue Topic-Ebene, die durch ein Null-Zeichen definiert ist, einführt und dementsprechend in Client Anwendungen beim Parsen von Topic-Namen Probleme verursachen kann. Weiterhin sind die Zeichen # *und* + bei der Benennung von Topics nicht zu verwenden, da es sich bei diesen um reservierte Zeichen für die in Abschnitt 2.1.7.7 beschriebene Wildcard Funktionalität des MQTT-Protokolls handelt. Damit jede, auch eine neu initiierte Anwendung die auf konfigurationsrelevanten Topics als Subscriber fungiert, alle Nachrichten erhält, wird festgelegt, dass alle Nachrichten mit dem gesetzten Flag *retained* und mindestens *QoS Level 1* verschickt werden. Dies gilt insbesondere für die Nachrichtentypen *BIRTH*, *DEATH*, *CONFIG* und *STATE*.

5.2.1.1 Eindeutige Identifikation von EoN-Devices

Durch die vorgegebene Topic-Hierarchie ist es erforderlich, dass all *EoN-Node-Devices* eine eindeutige ID besitzen und in der Lage sind diese selbst zu generieren. Besteht diese Funktionalität nicht, müssen diese manuell konfiguriert werden. *EoN-Legacy-Devices* erfordern ebenso, im Kontext eines *EoN-Node-Devices*, eine eindeutige *ID*. Eine verlässliche Möglichkeit zur Generierung von eindeutigen IDs sind *Universally Unique Identifiers (UUIDs)*. Eine UUID ist eine 128 Bit lange Zeichenkette mit der Garantie eindeutig zu sein. *RFC 4122* etabliert verschiedene Versionen der UUID-Generierung. Diese lassen sich wie in Tabelle 5.2 aufgeführt in drei Kategorien einteilen. [70]

Art der Generierung	Version	Beschreibung
Zeit basiert	UUIDv1	Verwendet Zeitstempel und die MAC-Adresse eines Geräts zur Erzeugung von UUIDs.
Namens-basiert	UUIDv3, UUIDv5	Verwendet als Eingabe einen Namespace und Name. Erzeugt einen Hash aus diesen Eingaben und verwendet diesen zur Generierung einer reproduzierbaren UUID.
RNG-basiert	UUIDv4	Verwendet echte Zufallszahlen oder Pseudo-Random-Number-Generatoren zur Generierung von UUIDs.

Tabelle 5.2: Verschiedene Arten der UUID-Generierung

Gemäß dieser Unterscheidung sind entweder *Version 1* oder *Version 4* die bevorzugte Wahl. IDs sollen in diesem Kontext nur die Anforderung der Einzigartigkeit erfüllen, da sie lediglich zur Identifizierbarkeit von Geräten genutzt werden, müssen sie nicht reproduzierbar sein.

5.2.1.2 Entwurf eines Nachrichtenschemas

Für eine strukturierte Kommunikation ist eine einheitliche, auf jeden Nachrichtentyp anwendbare Nachrichtenstruktur entscheidend. Dazu muss zunächst ein universelles Nachrichtenformat definiert werden. An dieser Stelle unterscheidet man grundlegend zwischen *textbasierten* und *binären* Nachrichtenformaten. Binäre Nachrichtenformate, wie beispielsweise *Google Protocol Buffers*, serialisieren den Nachrichteninhalt in ein binäres Format. Kompatible Clients sind in der Lage empfangene Nachrichten wieder zu deserialisieren und auszulesen. Dadurch ergibt sich der Vorteil, dass Nachrichten komprimiert werden und eine geringere Größe aufweisen. Jedoch ist hier ein Implementierungsmehraufwand erforderlich, da ein Nachrichtenschema in einer speziell entwickelten Sprache in eigenen Dateien definiert und kompiliert werden muss. Text basierte Formate wie JSON oder XML stellen den Nachrichteninhalt in einem lesbaren Format dar. Diese können von den meisten Programmiersprachen verarbeitet werden, indem der Inhalt auf simple Map ähnliche Datenstrukturen abgebildet wird. [13]

Um die Entwicklung im Kontext der Lehre transparenter zu gestalten, verwendet diese Arbeit das Text basierte Nachrichtenformat *JSON* für eine bessere Nachvollziehbarkeit und Lesbarkeit von Nachrichten. Grundsätzlich werden die in Tabelle 5.3 aufgeführten Datentypen unterstützt. [71]

Datentyp	Beschreibung
Number	Dezimahlzahlen analog zu den gängigen Programmiersprachen.
String	UTF-8 kodierte Strings.
Array	Klassische Array Struktur. Elemente müssen nicht vom gleichen Typ sein.
Object	Objekte in JSON Struktur.
Boolean	Einer der Werte true oder false.
Null	Null Wert. Wird als null angegeben.

Tabelle 5.3: JSON unterstützte Datenformate

Für alle Nachrichtentypen wird die in Listing 5.2 dargestellte Nachrichtenstruktur verwendet. Diese orientiert sich ebenfalls eng an der *Sparkplug-Spezifikation*. Eine Nachricht enthält auf oberster Ebene die Felder *timestamp* und *metrics*. Das metrics Feld enthält alle Elemente die über eine Nachricht übertragen werden sollen. Dabei muss immer ein *Name*, ein *Zeitstempel*, ein *Datentyp* und der *Wert* angegeben werden. Der Name einer Metrik bezeichnet stets den entsprechenden Nachrichtentyp unabhängig vom Gerätetyp[1], gefolgt von einem *Forward Slash* als Trennzeichen, auf welches wiederum der Name des entsprechenden Attributs folgt.

```
{
    "timestamp": "TIMESTAMP",
    "metrics" : [
        {
            "name": "MESSAGETYPE/ATTRIBUTE",
            "timestamp": "TIMESTAMP",
            "dataType": "DATATYPE",
            "value": "JSON_VALUE"
        }
    ]
}
```

Listing 5.2: Payload Struktur

5.2.1.3 Automatische Bereitstellung von EoN-Devices

In Kapitel 3.3.1 wurden bereits die Grundkonzepte der Eclipse Sparkplug-Spezifikation vorgestellt. Darunter auch das Konzept von *Birth- und Death-Zertifikaten* zur Bereitstellung oder Abmeldung von Geräten. Neu registrierte Geräte publizieren, sobald sie erfolgreich initialisiert wurden, über den Topic mit Nachrichtentyp *NBIRTH (EoN-Node-Devices)* oder *DBIRTH (EoN-Legacy-Devices)* eine Nachricht um dem Gesamtsystem mitzuteilen, dass sie verfügbar und betriebsbereit sind. Das *Birth-Zertifikat* enthält unter anderem etwaige benötigte Metadaten, die später in diesem Kapitel im Zuge der Objektrepräsentation der Metadaten noch vorgestellt werden. Geräte die sich vom System abmelden senden über den Topic mit Nachrichtentyp *NDEATH* oder *DDEATH* eine Nachricht, um alle Teilnehmer zu informieren, dass sie nicht mehr verfügbar sind. Durch diesen Lebenszyklus ist

[1] Beispielsweise DATA statt NDATA oder DDATA

es möglich, neue Geräte implizit, mittels der Wildcard-Funktionalität allen interessierten Services eines Systems bekannt zu machen.

5.2.1.4 Konfiguration von EoN-Devices

Da im gegebenen Kontext die extern gesteuerte Konfiguration von Geräten, insbesondere der *EoN-Node-Devices* erforderlich ist, können entsprechende Nachrichten zur Konfiguration über den Nachrichtentyp *NCONFIG* oder *DCONFIG* publiziert werden. Dieser Nachrichtentyp erweitert die im Rahmen der Sparkplug-Spezifikation vorgestellten Nachrichtentypen, um die Möglichkeit, gerätespezifische Konfigurationsdaten zu übermitteln.

5.2.1.5 Senden und Empfangen von Daten

Jegliche von Sensoren und Aktoren produzierte Daten werden durch Angabe des Nachrichtentyps *DDATA* oder *NDATA* im Topic Schema versendet. Für Aktoren relevante Steuerungsdaten werden über *NCMD* oder *DCMD* Nachrichten publiziert.

5.2.2 NoSQL /SQL - Auswahl eines geeigneten Datenspeichers

Zur Verwaltung von Geräten in einem Sensornetzwerk benötigte Stamm- und Metadaten erfordern ein geeignetes Datenhaltungssystem. In einer Smart City werden diese Daten benötigt um verschiedene Szenarien zu realisieren. Darunter auch sicherheitskritische Anwendungen der städtischen Infrastruktur. Eine zentrale Anforderung stellt die starke Konsistenz der Daten, sowie eine hohe Verfügbarkeit dieser dar. Die Gewährleistung einer hohen Verfügbarkeit und einer konsistenten Sicht zu jedem beliebigen Zeitpunkt auf hinterlegte Daten stellt einen essenziell wichtigen Aspekt der Anwendung dar.

Datenspeicher zur Persistierung von unstrukturierten Daten gewährleisten in der Regel eine hohe Verfügbarkeit, sowie eine gute horizontale Skalierbarkeit, nehmen jedoch Abstriche in der Konsistenz der Daten in Kauf. Moderne Systeme wie beispielsweise *Cassandra* implementieren das Konzept der *Eventual Consistency*, bieten jedoch Möglichkeiten an, das Level[2] der Konsistenz zu variieren. Die verschiedenen Level definieren sich durch die Zahl der replizierten Knoten die eine Lese- oder Schreiboperation bestätigen müssen. [11]

Klassische relationale Datenbanken hingegen basieren auf einem etablierten mathematischen und theoretischen Modell. Dementsprechend sind Datenspeicher für strukturierte Daten in der Welt der verteilten Systeme aufgrund ihres komplexeren Aufbaus weniger häufig vertreten. Ein Kernkonzept solcher Systeme sind Transaktionen. Transaktionen stellen atomare Einheiten, die eine oder mehrere Lese- und Schreiboperationen ausführen. Um den simultanen Zugriff auf Daten zu steuern, legt das *Isolationslevel* einer Transaktion [32] fest, ob ein simultaner Zugriff erlaubt oder nicht erlaubt ist. Hierbei existieren mehrere *Isolationsgrade*, vom niedrigsten Grad *Read Uncommitted* bis hin zum höchsten Grad *Serializable*. Ein hoher Isolationsgrad bedeutet lediglich, dass während eine Transaktion ausgeführt wird keine andere Operation, die das gleiche Datum betrifft, ausgeführt werden darf. Durch das höchste Isolationslevel kann so eine hohe Konsistenz gewährleistet werden. Die Umsetzung dieser Konzepte in verteilten System ist sehr komplex. Dennoch gibt es diverse Systeme, wie beispielsweise *CockroachDB* die diese Konzepte umsetzen, um sowohl eine starke Konsistenz als auch eine hohe Verfügbarkeit zu garantieren.

[2] Diese Option wird oft als *Tunable Consistency* bezeichnet.

Im Rahmen der Erweiterung einer modernen städtischen Infrastruktur um ein geeignetes Datenhaltungssystem spielen diverse Aspekte eine Rolle. Alle für die städtischen Infrastruktur relevanten Prozesse müssen gegebenenfalls auf eine Vielzahl an Sensordaten zugreifen oder Geräte ansteuern. Sind diese Daten beim Auslesen nicht aktuell oder sind beispielsweise steuerbare Geräte gelistet trotz, dass sie nicht mehr verfügbar sind, kann dies sich negativ auf die korrekte Ausführung dieser Prozesse auswirken. Mögliche Beispielszenarien wären die Berechnung einer Route für einen Krankenwagen anhand aktueller Verkehrsdaten oder eine automatisierte Ampelschaltung für einen Notfalleinsatz von Feuerwehr oder Polizei. In diesen Beispielen würde eine Datenhaltung die lediglich *Eventual Consistency* gewährleistet, im schlimmsten Fall zur Gefährdung von Personen führen. Ein weiteres potentielles Szenario in dem eine starke Konsistenz eine Rolle spielt, stellt die Triangulierung einer Position anhand von Bilddaten dar. Wobei sich die Bilddaten erzeugenden Geräte an unterschiedlichen Positionen befinden. Für eine genaue Triangulierung einer Position wären zu einem beliebigen Zeitpunkt der Positionsbestimmung aktuelle Daten von allen Geräten notwendig. Liegen jedoch veraltete Daten vor, kann unter Umständen kein verlässliches Ergebnis produziert werden.

Da es sich im gegebenen Kontext, gemäß dem in Abschnitt 5.3 vorgestellten Datenmodell um weitestgehend strukturierte Daten handelt und die Gewährleistung von *Eventual Consistency* in der späteren Anwendung nicht ausreichend ist, da sie auch Szenarien wie zuvor beschrieben abdecken soll, muss ein geeignetes SQL kompatibles relationales Datenhaltungssystem verwendet werden.

5.2.3 Monitoring und Log Aggregation

Damit Anwendungen fehlerfrei funktionieren ist es notwendig die Ressourcenauslastung von einzelnen Anwendungen zu beobachten, um mögliche Fehler zu erfassen. Zur Umsetzung dessen, soll der Service einen entsprechenden HTTP-Endpunkt zur Bereitstellung von Metriken über die aktuelle Auslastung implementieren. Mögliche Metriken umfassen unter anderem die Auslastung der CPU, des Arbeitsspeichers, die Zahl der verarbeiteten Requests über einen bestimmten Zeitraum oder die aktuelle Speicherbelegung. Die so bereitgestellten Metriken können genutzt werden, um eine Anwendung anhand ihrer Ressourcenauslastung zu skalieren sowie über eine entsprechende Monitoring-Komponente, Anwendungsinformationen zur Visualisierung und Clusterüberwachung zu aggregieren. Analog zur Aggregation und Bereitstellung von Metriken wird eine Anwendung bereitgestellt um clusterweit Logs von Anwendungen zu aggregieren und bereitzustellen. Rudimentäres Logging erfolgt in der Regel über die Standarddatenströme *stderr*, zur Ausgabe von Anwendungsfehlern und *stdout*, zur Ausgabe von Anwendungsinformationen.

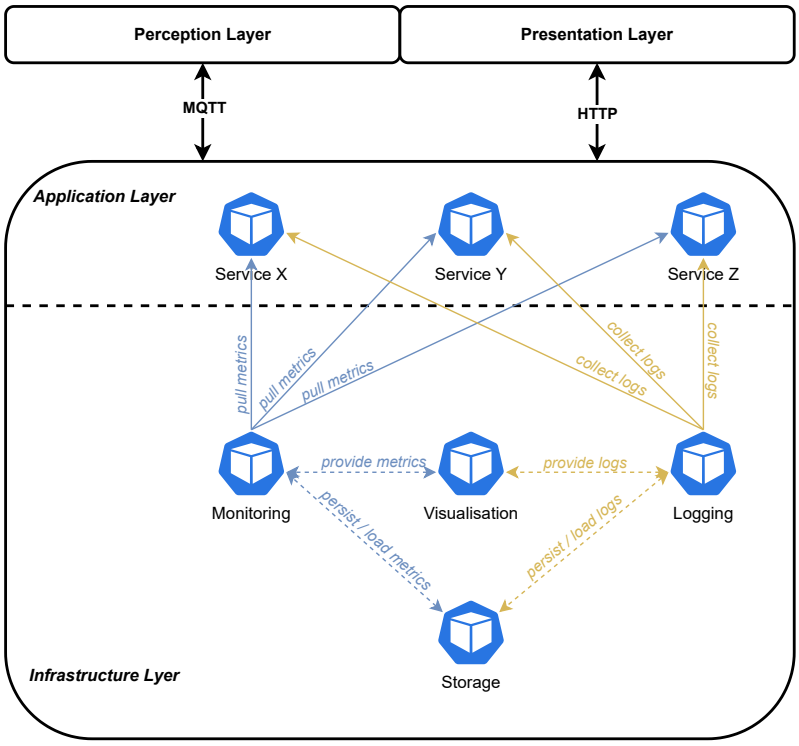


Abbildung 5.2: Monitoring und Logging im CiTe-Testfeld

Wie in Abbildung 5.2 skizziert, sammelt die Monitoring-Anwendung über einen Polling-Mechanismus in regelmäßigen Intervallen Metriken von allen Anwendungen die eine entsprechende Ressource zur Verfügung stellen. Weitere Tools bieten Funktionalität an, die gesammelten Daten[3] zu visualisieren.

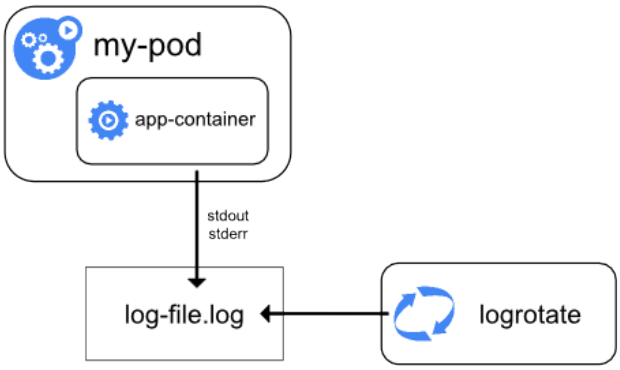


Abbildung 5.3: Kubernetes Logging Architektur [28]

Abbildung 5.3 zeigt die schematische Darstellung einer Log-Architektur in einem Kubernetes Cluster. Dabei werden Logs, die über die Standardstreams ausgegeben werden vom *kubelet Service* erfasst und an die entsprechende Container-Engine[4] weitergeleitet, die diese dann in Log-Dateien innerhalb des Containers persistiert. [28] Eine entsprechende Anwendung aggregiert die so erstellten Logs innerhalb des Clusters und ermöglicht

[3] Oft wird die Visualisierung, über speziell auf das Szenario angepasste Dashboards visualisiert.
[4] Unter Engine versteht man die Containerlaufzeitumgebung wie beispielsweise die weitläufig verbreitete Docker Engine.

analog zum Monitoring Mechanismus eine Visualisierung mittels entsprechender Tools.

5.3 Modellierung von Domain Objekten

Für die semantische Darstellung aller benötigten Daten eines *EoN-Devices* muss ein entsprechendes Datenmodell entwickelt werden. Diese Entitäten stellen im Kern einer Anwendung die Geschäftsobjekte dar.

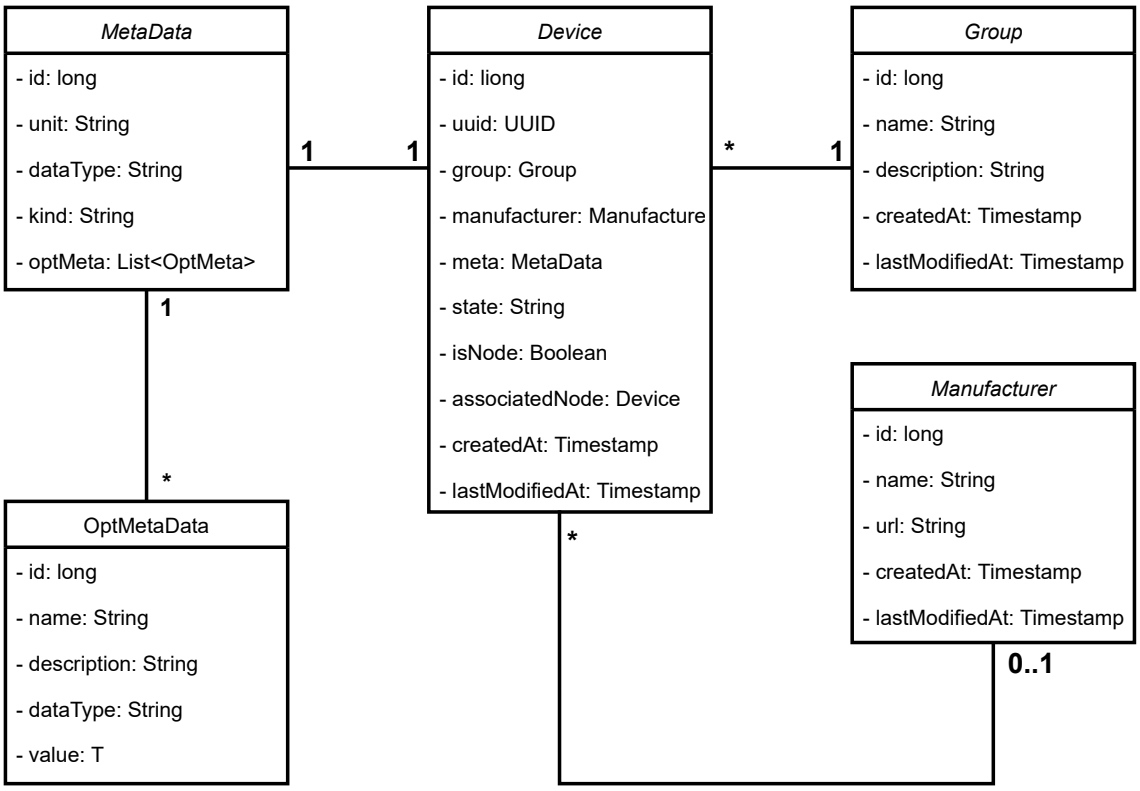


Abbildung 5.4: Objektrepräsentation des Datenmodells

Abbildung 5.4 stellt die Repräsentation dieser Objekte und deren Beziehungen untereinander dar. Im Zentrum steht die Objektdarstellung eines EoN-Devices. Da sowohl *EoN-Node-Devices* als auch *EoN-Legacy-Devices* eine große Schnittmenge an gemeinsamen Attributen besitzen, werden beide durch die Entität *Device* repräsentiert. Zur Unterscheidung dient ein boolescher Indikator. EoN-Legacy-Devices verweisen über das Feld *associatedNode* auf das zugehörige Gerät das als Gateway fungiert. Zur leichteren Verwaltung ist die Möglichkeit zur Gruppierung von Geräten anhand von ähnlichen Merkmalen von Vorteil. Gruppierungsmerkmale können die räumliche Position mehrerer Geräte, die Art der erfassten Daten oder jede beliebige logische Gruppierung sein. Die Gruppierung stellt gleichzeitig die Namensgrundlage für das *group_id* Attribut des Topic Namens. Ein *Group Objekt* legt einen Namen fest, welcher der in Abschnitt 5.2.1 beschriebenen Namenskonvention für Topics entsprechen muss.

Sensornetzwerke einer Smart City nutzen eine weite Bandbreite an Geräten von verschiedenen Herstellern. Durch diesen Aspekt unterscheiden sich diese in ihrer konkreten Anbindung und Implementierung. Ein *Manufacturer Objekt* kann mehreren Geräten zugeordnet werden um entsprechende Verweise auf herstellerspezifische Dokumentation zu speichern.

Durch das heterogen geprägte Angebot an Geräten werden wesentliche gerätespezifische Metadaten über die Entität *MetaData* beschrieben. Diese umfassen den Datentyp den ein Sensor produziert oder ein Aktor konsumiert, eine Einheit[5] die dem Datentyp eine konkrete semantische Bedeutung zuordnet, sowie ein Indikator zur Einordnung des Gerätes in die Kategorien *Sensor*, *Aktor* und *Gateway*. Weitere spezifische Metadaten können dynamisch durch eine *OptMeta* Entität erstellt werden und einer *MetaData* Entität zugeordnet werden.

5.4 Perception Layer - EoN Gateway

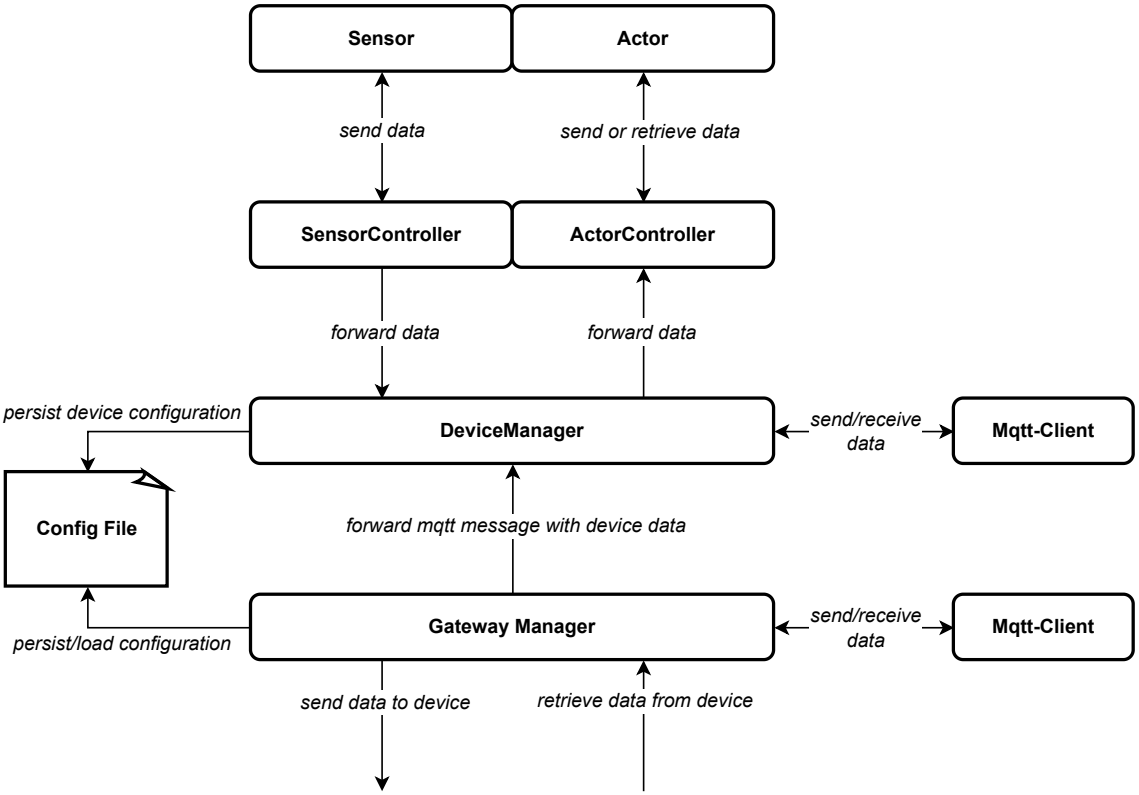


Abbildung 5.5: Architekturmodell Sensoranbindung

Abbildung 5.5 skizziert die Architektur der Anwendung zur Anbindung von Sensoren und Aktoren im *Perception Layer*. Zentrale Komponenten stellen die Komponenten *Gateway Manager* zur Verwaltung des Gateways[6], sowie *Device Manager* zur Verwaltung von einzelnen Grove Sensoren und Aktoren dar. Beide Komponenten kapseln jeweils eine Client Implementierung zur MQTT Kommunikation. Einzelne Sensoren oder Aktoren werden über einen entsprechenden *Controller* gesteuert. Dieser stellt Funktionalität zum auslesen von Sensordaten oder zum Senden von Steuerungsbefehlen an Aktoren bereit. Eine Controller Instanz wird wiederum von einer Device Manager Instanz gekapselt und ergänzt somit die zur Brokerkommunikation benötigte Konnektivität. Jede Instanz eines Device Manager verwaltet einen Sensor oder Aktor. Auf oberster Ebene verwaltet pro Gateway eine Gateway Manager Instanz ein oder mehrere Device Manager Instanzen. Die

[5] Beispielsweise eine standardisierte SI-Einheit wie Grad Celsius
[6] Der Raspberry PI mit GrovePi Erweiterung fungiert als Gateway

Konfiguration und damit auch die Initialisierung von Device Manager Instanzen wird über die Angabe der Konfiguration mittels eines strukturierten Textformats in einer Datei angegeben. Die Gateway Manager Komponente ist in der Lage diese Konfiguration bei der Initialisierung zu laden und die benötigten Device Manager mit passenden Controllern zu initialisieren. Alternativ können bereits verbundene Sensoren extern, über einen festgelegten Topic des Gateway Managers konfiguriert werden.

5.4.1 Konfiguration

Alle zur Konfiguration des Gateways und der Sensoren benötigten Daten werden im *JSON-Format* persistiert.

```
{
    "broker": {
        "host": 127.0.0.1,
        "port": 8883
        "security":{
            "ca_cert": "PATH",
            "tls_cert": "PATH",
            "tls_key": "PATH",
            "username": "USERNAME",
            "password": "PASSWORD"
        }
    }
    "gateway": {
        "group": "cite",
        "devices": [
            {
                "port": "PORT_TYPE",
                "pin": "PIN_NUMBER",
                "controller": "CONTROLLER_TYPE"
            }
        ]
    }
}
```

Listing 5.3: Gateway Konfiguration

Listing 5.3 beschreibt alle benötigten Konfigurationsdaten. Die Broker Konfiguration beinhaltet alle nötigen Verbindungsdaten um eine Verbindung mit dem Message-Broker im Cluster herzustellen. Darunter die *IP-Adresse*, der entsprechende *Port*, sowie alle für die im DSL-Cluster konfigurierte TLS v.1.2 Verschlüsselung benötigten *Sicherheitszertifikate*. Unter dem optionalen Wert *gateway* werden unter anderem die zur Netzwerkkommunikation benötigte *Group ID*, sowie alle vom Gateway verwalteten Sensoren angegeben. Zur Konfiguration eines einzelnen Sensors wird die Angabe des erforderlichen *Hardwareports* des Gateways, die *Nummer des Gerätepins*, sowie die Angabe eines *optionalen Controller-Typs* benötigt. Der Controller-Typ gibt an mit welchem Controller ein Sensor initialisiert wird. Es soll standardmäßig ein generischer Controller mit rudimentären *Read- und Write-Operationen bereitgestellt werden* , der mit den meisten Sensoren kompatibel ist. Potentiell

können zukünftig komplexere Sensoren zum Einsatz kommen, welche eine erweiterte Funktionalität benötigen. In diesem Fall können weitere Controller implementiert werden und über das Attribut *controller* in der Konfiguration angegeben werden.

Gleichzeitig dient die Gatewaykonfiguration als Teil einer einfachen Implementierung eines Mechanismus zur Ausfallsicherheit. Die Anwendung soll in der Lage sein den ursprünglichen Zustand in potentiellen Fehlerszenarien, die zum Ausfall des Gerätes führen, wiederherzustellen. Dazu persistiert der Gateway Manager jede Konfiguration eines neu initialisierten Geräts in der Konfigurationsdatei. So kann bei einem Neustart des Gateways dessen Zustand vor Ausfall wiederhergestellt werden.

Die Entscheidung die hardwarespezifische Konfiguration von Sensoren, welche den Sensortyp und die Gerätepin umfasst, ausschließlich lokal auf dem Dateisystem des Gateways zu persistieren, wurde in Abwägung weiterer Möglichkeiten getroffen. Die evaluierten Szenarien umfassen die exklusive Persistierung auf Backendseite und die redundante Persistierung im Backend als auch Gateway. Diese Szenarien gewähren lediglich den Vorteil die Konfiguration, auf direktem Weg über das Backend abzurufen. Nachteilig zu betrachten ist in beiden Szenarien die Einführung von Hardwarespezifika in das Datenmodell. Dies gilt es zu vermeiden, da die Verwendung von anderen Geräten mit einer abweichenden Konfiguration, ebenso in Betracht gezogen werden muss.

5.4.2 Broker Kommunikation

Da alle Geräte über einen Message-Broker an das Netzwerk angebunden werden, ist die MQTT-Kommunikation ein wichtiger Aspekt der Anwendung. Sowohl Gateway-Manager als auch Device-Manager kommunizieren auf mehreren festgelegten Topics, welche wie bereits in Abschnitt 5.2.1 beschrieben durch einen entsprechenden Nachrichtentyp definiert sind.

Metrik-Kennungen	Typ	Beschreibung
Senden von Daten		
MetaData/kind MetaData/unit MetaData/dataType MetaData/manufacturerName MetaData/manufacturerUrl	NBIRTH, DBIRTH	Metadaten die durch das vorliegende Datenmodell definiert sind. Diese werden in einem Birth-Zertifikat übermittelt.
OptMetaData/{property}	NBIRTH, DBIRTH	Metadaten die nicht durch das vorliegende Datenmodell definiert sind. Diese werden optional in einem Birth-Zertifikat übermittelt.
Info/identifier	NDEATH, DDEATH	ID des abzumeldenden Geräts. Wird bei einem Death-Zertifikat übermittelt.
Info/identifier Info/state Info/detail	NSTATE, DSTATE	ID, sowie Status und optional detaillierte Information zum Status eines Geräts. Wird in einer Status-Nachricht bei Änderung des Status übermittelt.
Inputs/{name}	NDATA, DDATA	Enthält erfasste Metriken. Diese werden nach ReportByException-Prinzip übermittelt.
Empfangen von Daten		
Config/changeGroup	NCONFIG, DCONFIG	Enthält den Wert zur Änderung der group_id eines EoN-Node-Device oder EoN-Legacy-Device.
Config/register	NCONFIG	Enthält notwendige Informationen zur Registrierung neuer Sensoren über ein als Gateway fungierendes EoN-Node-Device.
Config/delete	NCONFIG, DCONFIG	Löschen von EoN-Node-Devices oder EoN-Legacy-Devices. Meldet das Gerät vom Netzwerk ab.
Outputs/{name}	NCMD, DCMD	Enthält Steuerungsbefehle die von einem Peripheriegerät verarbeitet werden können.

Tabelle 5.4: Detaillierte Übersicht der Nachrichtentypen

5.4.2.1 Gateway Manager

Das Gateway, im gegebenen Kontext der Raspberry Pi, wird durch die Gateway Manager-Komponente repräsentiert. Diese soll Funktionalität zur Verfügung stellen um das Gerät im System automatisch bereitzustellen und abzumelden, erfasste Daten zu publizieren sowie Konfigurationsnachrichten und Steuerungsbefehle empfangen. Dabei ist anzumerken, dass nicht jedes EoN-Node-Device jede Funktionalität unterstützt, aber dennoch die nachfolgenden Subscriptions initialisiert werden, damit eine einheitliche Kommunikationsstruktur gegeben ist:

- **ciTe/group_id/NCONFIG/node_id**
- **ciTe/group_id/NCMD/node_id**

Automatische Bereitstellung Die in Tabelle 5.4 aufgeführte Kennung *MetaData* bezieht sich auf die im Datenmodell spezifizierten Metadaten. Dementsprechend sind die zu übermittelnden Metadaten durch die Entität *MetaData* spezifiziert. Die Angabe der Metadaten ist nicht zwingend erforderlich, sollte aber bei Konfiguration von Peripheriegeräten auf Seite des EoN-Devices, der Übersicht halber angegeben werden. Fehlende Metadaten können bei Bedarf, anschließend über die Backend-Anwendung nachgetragen werden. Die Verwendung der Kennung *OptMetaData* erlaubt die dynamische Angabe von Metadaten, die nicht durch das gegebene Modell definiert sind. Unter Berücksichtigung der erlaubten Typen, gemäß der JSON-Spezifikation können so bei Bedarf weitere beliebige Metadaten angegeben werden. Diese Daten werden nach Initialisierung des Gerätes in einem Birth-Zertifikat (NBIRTH) angegeben und dienen der automatischen Bereitstellung des Gerätes. Listing 5.4 stellt exemplarisch ein Birth-Zertifikat zu Bereitstellung des Raspberry Pi dar.

```
{
        "timestamp": "TIMESTAMP",
        "metrics":[
        {
            "name": "MetaData/kind",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "gateway"
        },
        ...
        {
            "name": "MetaData/manufacturerUrl",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "https://www.seeedstudio.com"
        },
        {
            "name": "OptMetaData/location",
            "timestamp": "TIMESTAMP",
            "dataType": "object"
            "value": {
                "x": "779236.44",
                "y": "6314546.41",
```

```
                "srs": "EPSG:3857",
                "url": "https://epsg.io/3857"
                }
        }
    ]
}
```

Listing 5.4: Bereitstellung des Raspberry-Pi über eine NBIRTH-Nachricht

Neben der Bereitstellung wird der Nachrichtentyp *NDEATH* verwendet um zu signalisieren, dass ein durch den Gateway Manager verwaltetes Gerät keine Verbindung mehr zum Message-Broker besitzt. Ein solches Death-Zertifikat wird bei Initialisierung des Gateway-Managers als LWT-Nachricht auf dem entsprechenden Topic festgelegt, sodass diese, sobald die Verbindung zum Broker geschlossen wird, die Nachricht versendet wird. Eine empfangene Nachricht auf dem NDEATH-Topic auf Backendseite kann unabhängig ihres Inhaltes als Signal zur Abmeldung eines Gerätes genutzt werden, da die Kennung des abzumeldenden Gerätes aus der Topic-Struktur abgeleitet werden kann. Der Inhalt einer solchen Nachricht spielt dementsprechend keine entscheidende Rolle und kann optional wie in Listing 5.5 dargestellt angegeben werden.

```
{
    "timestamp": "TIMESTAMP",
    "metrics": [
        {
            "name": "Info/identifier",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "b3465aac-755e-46b8-a16d-c2a0856c2ad7"
        }
    ]
}
```

Listing 5.5: Abmeldung des Raspberry-Pi über eine NDEATH-Nachricht

Publizieren von Sensormetriken Aufgrund der Funktion des Gateway-Managers als verwaltende Instanz eines Gateways erfasst dieser selbst keine Sensordaten, die publiziert werden müssen. Prinzipiell würden Daten über *NDATA*-Nachrichten publiziert werden, da unter den gegebenen Rahmenbedingungen keine Erfassung von Daten durch das Gateway vorgesehen ist, werden keine *NDATA*-Nachrichten versendet.

Empfang von Konfigurationsnachrichten Die Konfiguration eines Gateways, verwaltet durch einen Gateway-Manager, umfasst die Registrierung neuer Sensoren, die Änderung der *group_id*, sowie das Löschen des Gateway Managers, was wiederum das Gerät und alle verwalteten Sensoren vom System abmeldet. Für jede Funktionalität wird, wie Tabelle 5.4 zu entnehmen, eine Metrik mit der Kennung *Config* spezifiziert. Durch die zuvor eingeführte generische Payloadstruktur ist die Angabe mehrerer dieser Operationen in einer *NCONFIG*-Nachricht an das Gateway möglich. In Listing 5.6 ist eine Nachricht dargestellt, die exemplarisch jede dieser Operationen aufführt. Alternativ kann für jedes Listenelement unter dem Attribut *metrics* eine eigenständige Nachricht verschickt werden.

```
{
    "timestamp": "TIMESTAMP",
    "metrics": [
        {
            "name": "Config/register",
            "timestamp": "TIMESTAMP",
            "dataType": "object",
            "value": {
                "port_type": "analog",
                "pin": 2,
                "device_type": "sensor",
                "controller": "genericSensorController"
            }
        },
        {
            "name": "Config/delete",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "b3465aac-755e-46b8-a16d-c2a0856c2ad7"
        },
        {
            "name": "Config/changeGroup",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "cite-001"
        }
    ]
}
```

Listing 5.6: Exemplarische Darstellung einer NCONFIG-Nachricht

Die Registrierung stellt eine Besonderheit dar, da diese bestimmte Informationen benötigt, die der im CiTe-Testfeld genutzten Hardware geschuldet ist. Dementsprechend werden die folgenden Informationen zur Registrierung benötigt:

- **port_type** Dieses Attribut spezifiziert die Art des Hardware-Anschlusses des GrovePi+-Boards und kann die Werte *analog*, *digital* oder *i2c* annehmen.
- **pin** Über das Attribut *pin* wird der konkrete Anschluss spezifiziert. Dieser kann einen *Integer*-Wert annehmen der den Hardware-Pin angibt.
- **device_type** Das Attribut *device_type* definiert den Typ des Sensors und kann die Werte *sensor* oder *actor* annehmen.
- **controller** Da die Implementierung der Gateway-Anwendung die Erweiterung um individuell erstellte Controller erlauben soll, kann unter *controller* der zu verwendende Controller, zur Interaktion mit der Hardware, spezifiziert werden.

Die hier benötigten Informationen zur Registrierung neuer Geräte über das Gateway, beziehen sich speziell auf die im CiTe-Testfeld verwendeten Grove-Sensoren und Aktoren

und sind dem Aufbau des GrovePi+-Boards geschuldet. Für andere Gateway-Implementierungen, kann prinzipiell der gleiche Nachrichtentyp verwendet werden, lediglich die enthaltenen Informationen müssen den erforderlichen Hardwarespezifikationen angepasst werden.

Empfangen von Steuerungsbefehlen Im Kontext dieser Arbeit handelt es sich bei den verwendeten EoN-Node-Devices um Geräte die als Gateway fungieren. Da diese keine Steuerungsbefehle entgegennehmen, findet über den Nachrichtentyp NCMD keine Kommunikation statt.

5.4.2.2 Device Manager

Einzelne Sensoren und Aktoren nutzen die über den Device-Manager bereitgestellte Broker-Kommunikation zur Publikation von erfassten Daten oder zum Empfang von Steuerbefehlen. Analog zum Gateway-Manager initialisiert jede Device-Manager-Instanz die nachfolgenden Subscriptions:

- **ciTe/group_id/DCONFIG/node_id/device_id**
- **ciTe/group_id/DCMD/node_id/device_id**

Automatische Bereitstellung Die automatische Bereitstellung eines durch den Device-Manager verwalteten Sensors oder Aktors erfolgt durch Publikation eines *Birth-Zertifikat* (DBIRTH). Analog zum Gateway-Manager können in einer DBIRTH-Nachricht alle in Tabelle 5.4 dargestellten Metadaten mit den Kennungen *MetaData* und *OptMetaData* angegeben werden. Listing 5.7 stellt exemplarisch ein Birth-Zertifikat, in welchem Angaben zum Gerätetyp (*kind*), Einhheit (*unit*) und Datentyp (*dataType*) gemacht werden, dar.

```
{
    "timestamp": "TIMESTAMP",
    "metrics": [
        {
            "name": "MetaData/kind",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "sensor"
        },
        {
            "name": "MetaData/unit",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "degree celcius"
        },
        {
            "name": "MetaData/dataType",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "float"
        }
    ]
```

```
}
```

Listing 5.7: Birth-Zertifikat zur Bereitstellung eines Sensors

Publizieren von Daten Alle von einem Device Manager erfassten Sensordaten werden über eine DDATA-Nachricht publiziert. Dabei wird unter Verwendung, der in Tabelle 5.4 aufgeführten Kennung *Inputs*, die zu publizierende Metrik angegeben. Eine DDATA-Nachricht wird ebenfalls, wie Listing 5.8 zu entnehmen, mit der generischen Payload-Struktur veröffentlicht.

```
{
    "timestamp": "TIMESTAMP",
    "metrics": [
        "name": "Inputs/temperature",
        "timestamp": "TIMESTAMP",
        "dataType": "string",
        "value": 42
    ]
}
```

Listing 5.8: DDATA-Nachricht eines Sensors

Empfang von Konfigurationsnachrichten Im gegebenen Kontext soll durch einen Device Manager die Funktionalität zur Änderung der *group_id* eines Geräts zur Verfügung gestellt werden. Diese kann durch eine entsprechende DCONFIG-Nachricht, wie in Listing 5.9 dargestellt, aufgerufen werden.

```
{
    "timestamp": "TIMESTAMP",
    "metrics": [
        {
            "name": "Config/changeGroup",
            "timestamp": "TIMESTAMP",
            "dataType": "string",
            "value": "cite-001"
        }
    ]
}
```

Listing 5.9: DCONFIG-Nachricht zur Änderung der Group-ID eines Gerätes

Empfangen von Daten Durch einen Device Manager verwaltete Aktoren sind in der Lage über DCMD-Nachrichten Steuerungsbefehle zu erhalten. Die verfügbaren Steuerungsbefehle sind durch die Implementierung des Aktor-Controllers vorgegeben und sind dementsprechend anhand der vorliegenden Implementierung zu wählen. Mögliche Steuerungsbefehle werden mit der Kennung *Outputs* identifiziert. Listing 5.10 zeigt beispielhaft die Ansteuerung einer konfigurierbaren LED, die eine Änderung der Farbe über entsprechende Befehle ermöglicht.

```
{
    "timestamp": "TIMESTAMP",
    "metrics": [
        {
            "name": "Outputs/LED/green",
            "timestamp": "TIMESTAMP",
            "dataType": "bool",
            "value": true
        }
    ]
}
```

Listing 5.10: DCMD-Nachricht zur Steuerung eines Aktors

5.4.3 Anbindung

5.4.3.1 Plug' n Play Anbindung

Eine Plug'n Play Geräterkennung im Kontext des Raspberry Pis und des GrovePi Erweiterungsboard erfordert mehrere Aspekte die automatisiert erfasst werden müssen. Der Hardware-Pin auf dem ein entsprechendes Gerät angeschlossen wurde, sowie ob es sich bei einem Gerät um einen *digitalen*, *analogen* oder *i2c* kompatiblen Sensor handelt, sind Informationen die benötigt werden, damit ein Sensor oder Aktor in Betrieb genommen werden kann.

In der in Abschnitt 3.1.1 vorgestellten Arbeit beschäftigt sich Wagner in Ihrer Abschlussarbeit unter anderem mit der automatischen Erfassung von Sensoren und Aktoren an einem GrovePi Erweiterungsboard. Das Resultat der in diesem Kontext durchgeführten Recherche und Experimente ergab, dass es zum aktuellen Zeitpunkt keine zuverlässige Methode gibt um Grove Geräte akkurat zu erfassen.

Die vom Hersteller zur Verfügung gestellte Bibliothek für das Grovepi-Board, bietet eine Funktion zur Detektion von angeschlossenen I2C-Geräten. Alle I2C-Sensoren und -Aktoren des Herstellers SeeedStudio besitzen eine festgelegte Adresse. Jedoch ist auch mit dieser Funktionalität eine automatische Erfassung nicht möglich, da durch die mehrfache Vergabe von Adressen keine eindeutige Zuordnung zu einem Gerät möglich ist. [49] Folglich ist eine vollständige automatische Plug'n Play Erfassung im gegebenen Kontext aufgrund der vorliegenden hardwareseitigen Limitierung nicht umsetzbar und wird dementsprechend in der Konzeption nicht weiter behandelt.

5.4.4 Manuelle Anbindung

Da eine automatische Erfassung aufgrund der Limitierungen nicht ohne weiteres möglich ist, müssen Sensoren manuell konfiguriert werden, damit diese betriebsfähig sind. Dies erfordert die *Angabe des Hardwarepins*, die Einordnung des Sensortyps in *digital*, *analog* oder *i2c*. Diese Angabe kann entweder, wie in Abschnitt 5.4.1 beschrieben, über eine *lokale Konfigurationsdatei* erfolgen oder externalisiert über einen entsprechenden MQTT-Topic kommuniziert werden. Wird das Attribut *gateway* in der Konfigurationsnachricht nicht angegeben, muss die Konfiguration externalisiert über die Broker-Kommunikation erfolgen.

Im ersten Schritt wird bei Initialisierung der Anwendung die Konfiguration geladen. Es wird geprüft, ob sich ein Backup in den Konfigurationsdaten befindet. Ist ein Backup vorhanden, werden alle darin angegebenen Geräte initialisiert. Anschließend sollten alle Geräte, insofern diese noch physisch angeschlossen sind, betriebsbereit sein. Andernfalls wird zunächst der Gateway Manager initialisiert. Dieser generiert zunächst eine *UUIDv1*, welche die *node_id* in der Topic Struktur repräsentiert. Ist eine *Group-ID* in der Konfiguration angegeben, wird diese übernommen. Wurde keine Group-ID angegeben, wird die Group-ID *DEFAULT* verwendet. Nach vollständiger Initialisierung und Verbindung mit dem Message Broker schickt der Gateway Manager ein *Birth-Zertifikat* um das Gateway dem System bekannt zu machen. Anschließend werden relevante Informationen bezüglich aller angeschlossenen Geräte, entweder über die *lokale* oder *externe Konfiguration* bezogen. Der Gateway Manager generiert für jedes Gerät wiederum eine *UUIDv1* und initialisiert, falls nicht anders angegeben, für jedes Gerät einen Device-Manager mit der generierten ID und der Group-ID des Gateways. Um alle Geräte dem System verfügbar zu machen sendet jeder Device-Manager ein Birth-Zertifikat auf dem entsprechenden Topic.

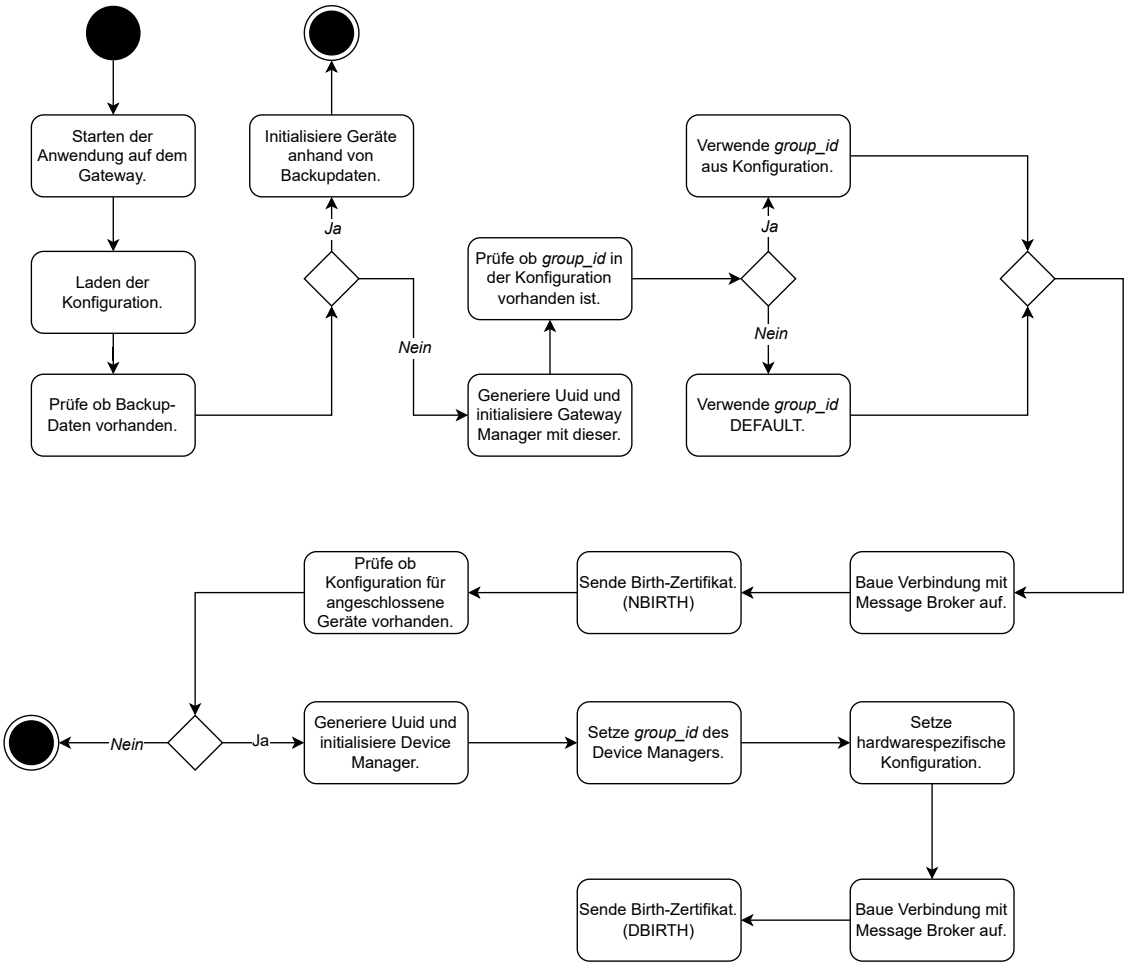


Abbildung 5.6: Ablauf der Anbindung von Geräten.

5.5 Application Layer - Microservice Metadatenverwaltung

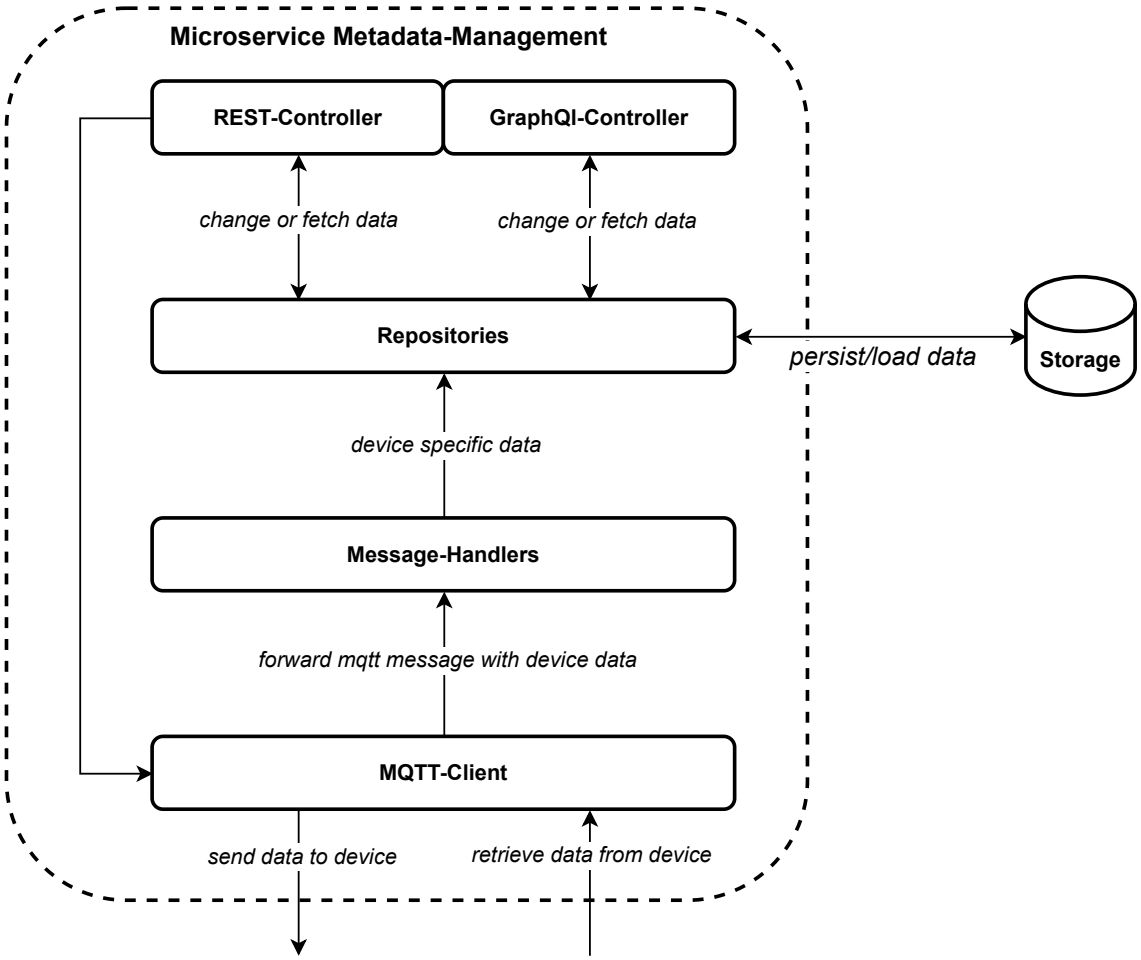


Abbildung 5.7: Architekturmodell Metadatenverwaltung

Die Verwaltung der durch *EoN-Devices* bereitgestellten Metadaten bildet den Kern des Gesamtsystems. Dieses System wird, wie in Abbildung 5.7 dargestellt als Microservice, welcher im Cluster des CiTe-testfeldes bereitgestellt werden kann, umgesetzt. Die Bereitstellung und Konfiguration der zu verwaltenden Geräte erfolgt über eine *Broker Client Implementierung*. Über diese können Daten auf festgelegten Topics gesendet und empfangen werden. Neue Geräte werden ebenso automatisch der Anwendung bekannt gemacht. *Message Handler* definieren für verschiedene Nachrichtentypen Callbacks um entsprechend zu reagieren. Die Persistierung erfolgt über Daten *Repositories*. Diese kapseln Funktionalität zur Interaktion mit einem Datenhaltungssystem. Über eine *universelle Schnittstelle* können andere Anwendungen Daten abrufen oder je nach Sicherheitsrichtlinie erstellen oder manipulieren, sowie auf fest definierten Brokerkanälen EoN-Geräte konfigurieren. Zur Umsetzung dieser Funktionalität soll primär eine *REST-Schnittstelle* angeboten werden. Als Alternative wird weiterhin eine Schnittstelle nach Spezifikation der Abfragesprache *GraphQL* integriert. Einer Client-Applikation steht es dementsprechend frei, welche Schnittstelle genutzt werden kann.

5.5.1 Broker Kommunikation

Damit Metadaten von im Perception-Layer angesiedelten Geräten verwaltet werden können, ermöglicht die Anwendung eine bidirektionale Kommunikation mit Geräten über den Message-Broker. Konfigurationsnachrichten können wie in Abschnitt 5.4.2 beschrieben an einzelne Geräte versandt werden. In erster Linie dient die Anwendung jedoch der Verwaltung von im Perception-Layer betriebenen Geräten. Um dies zu ermöglichen werden wie in Abschnitt 5.2.1 beschrieben, entsprechende Nachrichten die eine bestimmte Stufe im Lebenszyklus eines Gerätes repräsentieren, empfangen und verarbeitet, indem Änderungen im verknüpften Datenhaltungssystem persistiert werden.

Mit Hilfe der Wildcard Funktionalität werden Informationen zu Geräten auf einfache Weise aggregiert, wodurch ein einfacher Mechanismus zur automatischen Bereitstellung umgesetzt werden kann. Dementsprechend hält eine Instanz der Anwendung Subscriptions auf die nachfolgenden Topics:

- **simcity/+/NBIRTH/+ und simcity/+/DBIRTH/+/+**
  Alle neu registrierten *EoN-Devices* können so automatisch erfasst werden. Auf diesem Kanal empfangene Nachrichten repräsentieren ein neu registriertes Gerät. Ein Gerät, dass ein Birth-Zertifikat sendet wird als lauffähig betrachtet und somit mit allen weiteren relevanten Daten und dem State *ONLINE* im Datenhaltungssystem persistiert.

- **simcity/+/NDEATH/+ und simcity/+/DDEATH/+/+**
  Auf diesem Topic werden Deathzertifikate eines *EoN-Device* empfangen. Die Anwendung prüft ob der Status nicht bereits auf *OFFLINE* oder *ERROR* gesetzt wurde. Ist der letzte Zustand *ONLINE* wird der neue Status *OFFLINE* gesetzt.

- **simictiy/+/NSTATE/+ und simcity/+/DSTATE/+/+**
  Dieser Topic stellt einen reservierten Kanal zur Kommunikation von Statusinformationen des Gerätes zur Verfügung. Eine Nachricht indiziert die fehlerfreie Ausführung unter Angabe des Zustandes *ONLINE* oder *ERROR* im Falle eines Fehlers. Wird der Zustande *ERROR* kommuniziert, persistiert die Anwendung den Zustand im Datenhaltungssystem.

5.5.2 Persistierung im Kontext von verteilten Systemen

Die Kommunikation zum Datenhaltungssystem wird mit Hilfes des *Repository Patterns* umgesetzt. Durch Kapselung des unterliegenden Datenhaltungssystem sowie der Abfragesprache kann ein *technologieunabhängiges globales Interface* für den Zugriff auf ein beliebiges Datenhaltungssystem bereitgestellt werden. Ein Repository stellt dabei ein *Kollektion einer Entität* dar. Dementsprechend wird jeder in Abschnitt 5.3 eingeführten Entität ein Repository zugeordnet. Dieses stellt grundlegende *CRUD-Funktionalität* in Form von Funktionen zur Verfügung um Daten im Datenhaltungssystem zu manipulieren und abzufragen. Funktionsaufrufe eines Repositories nehmen Entity-Objekte entgegen um die so repräsentierten Daten zu persistieren. Diese Aufrufe werden in entsprechende Statements in der Abfragesprache[7] des Datenhaltungssystem umgewandelt und dem Datenhaltungssystem übermittelt.

[7] Entsprechende Implementierungen bieten oft verschiedene Datenbanktreiber an um eine große Bandbreite an Abfragesprachen und Dialekten zu unterstützen.

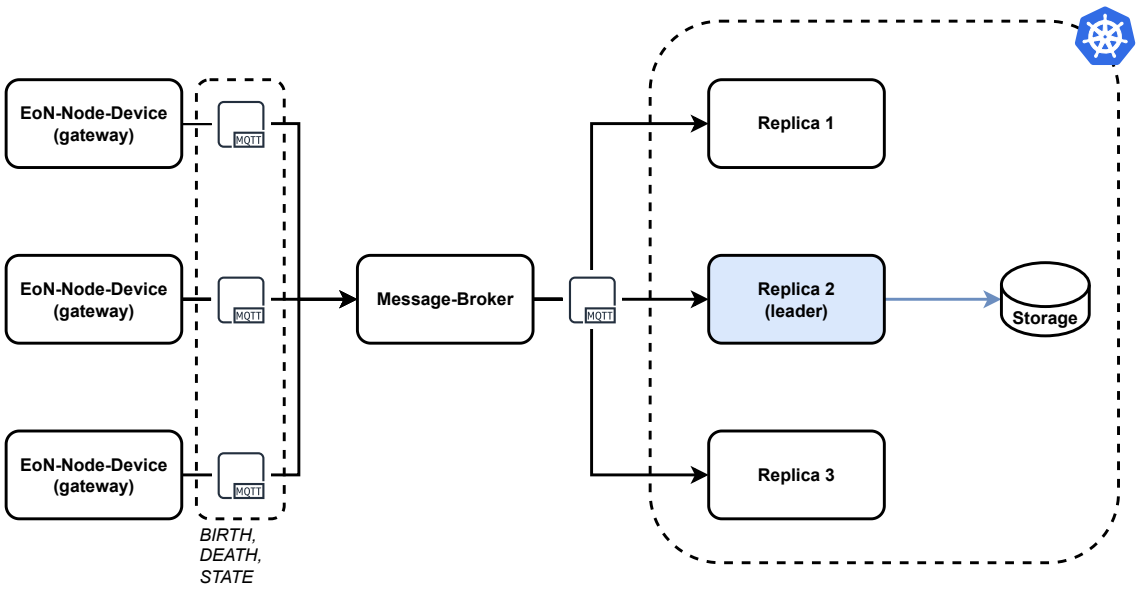


Abbildung 5.8: Persistierung von automatisch bereitgestellten Gerätedaten

Wie im vorigen Abschnitt 5.5.1 beschrieben, werden über den Message-Broker übermittelte Daten zu Geräten im Perception-Layer im Datenhaltungssystem persistiert. In Folge dessen, dass Nachrichten im Kontext des MQTT Protokolls via Broadcast an alle Subscriber verschickt werden, versuchen alle Empfänger die gleichen Daten zu persistieren. Damit dadurch entstehende Konflikte vermieden werden können muss, wie in Abbildung 5.8 gezeigt, eine Instanz der Anwendung bestimmt werden, die ein Datum persistieren darf. Bei dieser Problemstellung handelt es sich um ein bekanntes Problem aus dem Bereich der verteilten Systeme, welches gemeinhin als *Leader Election* Problem bezeichnet wird. Wie in Abschnitt 2.1.3 beschrieben, muss ein passender Algorithmus entsprechend der vorliegenden Netzwerktopologie ausgewählt werden. Die vorgestellten Algorithmen benötigen ein gewisses Maß an a priori Wissen bezüglich anderer vorhandener Instanzen der Anwendung. Repliken der Anwendung im Cluster besitzen keine Kenntnis über andere Repliken. Unter Zunahme der Broker gestützten Kommunikation beschreiben Services im Cluster ein vollständig verbundenes Mesh-Netzwerk , bei dem jeder Knoten mit jedem anderen kommunizieren kann. Dementsprechend sind die in Abschnitt 2.1.3 vorgestellten Algorithmen nicht geeignet um einen Koordinator zu bestimmen.

Eine einfache Variante zur Bestimmung eines Koordinators im Kontext eines durch Kubernetes verwalteten Clusters basiert auf dem Prinzip des *Distributed Lock* wie in Abbildung 5.9 grob skizziert. Im Namespace einer Anwendung wird ein *ConfigMap* oder *Lease* Objekt verwendet um einen Koordinator zu bestimmen. Dabei hält der Koordinator einen *Lease*, in Form eines *Zeitstempels*, der in einem festgelegten Intervall erneuert werden muss. Alle anderen Repliken prüfen regelmäßig ob der Koordinator seinen Lease erneuert hat. Stellen Repliken fest, dass der Koordinator den Zeitstempel nicht im festgelegten Intervall erneuert hat, versuchen alle aktiven Repliken ein *Lock* auf das Objekt zu akquirieren und sich als neuen Koordinator zu vermerken. Dabei wird die Replik die zuerst Zugriff erhält als Koordinator ernannt. [68]

Eine jede Instanz muss eine Variable *isCoordinator* verwalten, über welche bei jedem relevanten Persistierungsschritt geprüft wird, ob eine Instanz den Koordinatorstatus besitzt. Auf diese Weise ist lediglich ein Koordinator in der Lage relevante Daten zu persistieren.

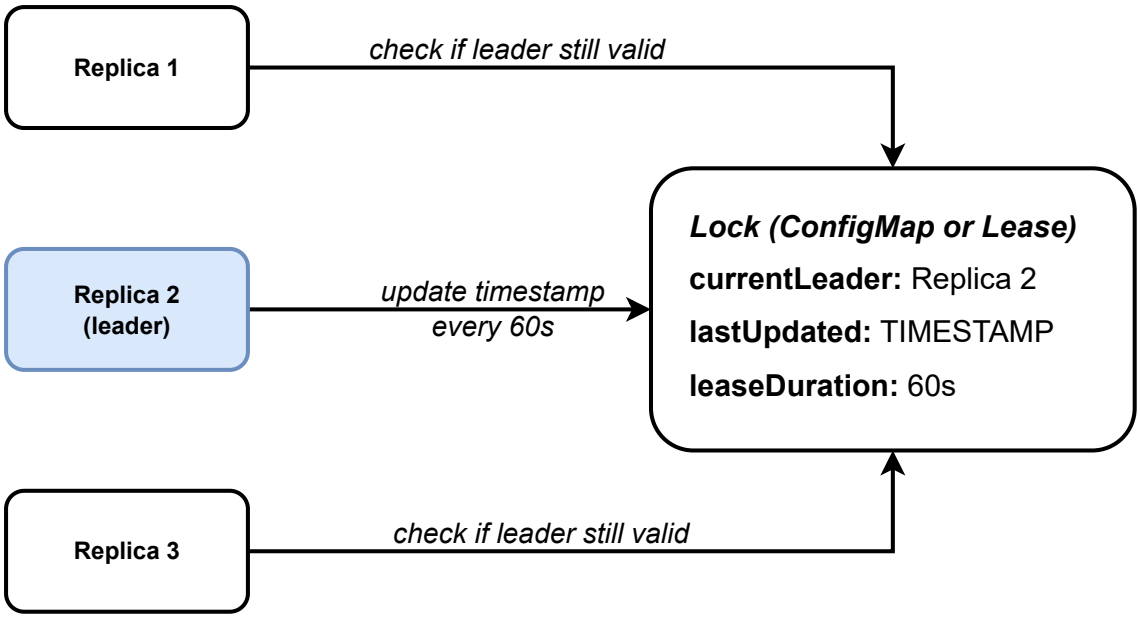


Abbildung 5.9: Bestimmung eines Koordinators in Kubernetes

Dabei erfordert die Persistierung von Daten aus *Birth- und Deathzertifikaten*, sowie *STATE-Nachrichten* die Wahl eines Koordinators, da diese Nachrichten von allen Repliken eines Deployments gleichzeitig empfangen werden.

5.5.3 Universelle Schnittstelle

Eine externe Schnittstelle soll bereit gestellt werden, um Clients zu ermöglichen Daten abzufragen oder zu ändern. Als allgemeiner Konsens zur Entwicklung einer universellen Schnittstelle für Anwendungen im CiTe-Testfeld, wurde als Konvention die Verwendung von REST-Schnittstellen festgelegt. Dieses Konzept ist weit verbreitet und wird von den meisten Frameworks unterstützt. Da sich diese Arbeit in Bezug auf das CiTe-Testfeld im Kontext der Lehre bewegt, wird im Folgenden neben einer REST-Schnittstelle eine alternative Schnittstelle basierend auf der Abfragesprache GraphQL konzipiert um eine entsprechende Grundlage für zukünftige Forschungsarbeiten zu schaffen.

GraphQL wurde entwickelt um einigen Nachteilen einer REST-Architektur entgegenzuwirken. Im Fokus steht hier das Problem des *Under- und Overfetchings*, das bei klassischen REST-Architekturen auftreten kann. Overfetching bezieht sich auf den Overhead einer Anfrage, wenn nur ein Subset der abgefragten Daten über einen Endpunkt benötigt wird. Das kann je nach Größe der Daten in einem System erheblichen Einfluss auf die Performance haben. Underfetching hingegen beschreibt das Problem, dass eine einzelne Abfrage an einen Endpunkt nicht alle benötigten Daten zurückgibt und somit eine weitere Anfrage zu einem weiteren Endpunkt erzwingt. Dadurch entsteht ein Overhead in der Zahl der benötigten Anfragen um alle relevante Daten zu erhalten. GraphQL soll diese Probleme lösen indem nur relevante Daten geliefert werden. Diese werden durch eine eigene Abfragesprache definiert wodurch Overfetching vermieden werden kann. Die Abfrage mehrerer unabhängiger Entitäten würde in einer REST Architektur mehrere Anfragen zu unterschiedlichen Ressourcen erfordern. Da die GraphQL-Spezifikation eine Abfragesprache definiert, die es erlaubt beliebig viele, voneinander unabhängige Entitäten anzugeben, können diese im Zuge einer einzelnen Anfrage auf Serverseite durch

die entsprechenden Repositories abgerufen werden und an den Client zurückgegeben werden.

Jedoch bietet GraphQL im Vergleich nicht nur Vorteile. Durch Verwendung einfacher HTTP-Caches kann in REST-Architekturen ohne viel Mehraufwand natives Caching genutzt werden. GraphQL unterstützt kein natives Caching und die Implementierung entsprechender Mechanismen gestaltet sich wesentlich komplexer. Weiterhin kann in REST-Anwendungen dadurch, dass eine Anfrage an eine Ressource einem Funktionsaufruf und somit einer Datenbankabfrage eines Repositories entspricht, die Frequenz eingehender Anfragen limitiert werden, um serverseitigen Performance Problemen entgegenzuwirken. Da GraphQL-Anfragen sich nicht zwingend auf eine einzige Entität beschränken, kann eine Anfrage auf Serverseite viele Datenbankabfragen zur Folge haben, wodurch sich ein Rate-Limiting-Mechanismus wesentlich komplexer gestaltet. Dementsprechend bieten beide Möglichkeiten je nach Anwendungsfall diverse Vor- als auch Nachteile.

5.5.3.1 REST-Schnittstelle

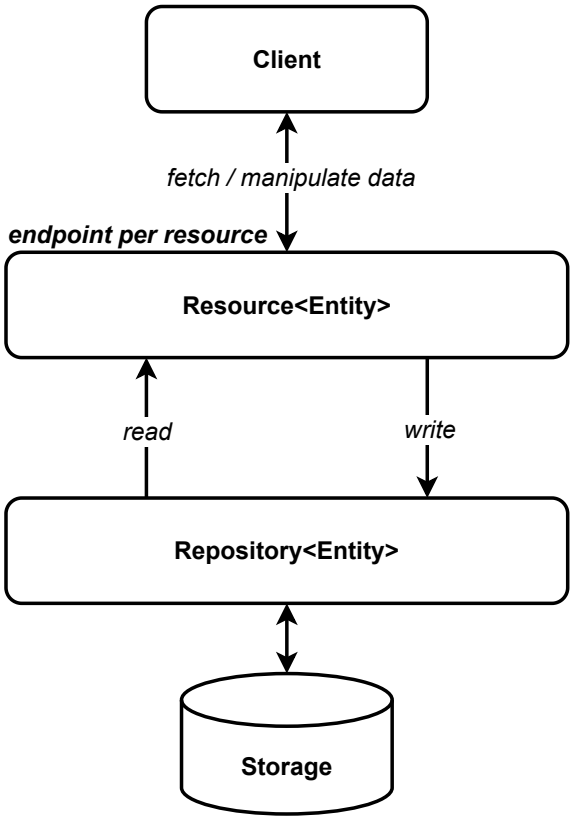


Abbildung 5.10: REST-Architekturkonzept

Eine auf den verfügbaren Repositories aufbauende REST-Schnittstelle stellt, wie in Abbildung 5.10 skizziert, pro Repository, das eine Entität verwaltet eine eigene Ressource bereit über die ein Client auf gespeicherte Daten zugreifen kann. Jede Ressource kapselt die *CRUD-Funktionalität* eines Repositories indem, wie in Tabelle 5.5 aufgeführt, alle Operationen auf passende *HTTP-Verben* abgebildet werden. Bei allen Schreiboperationen werden die Daten im Body eines Requests im *Textformat JSON* übermittelt.

HTTP-Verb	CRUD-Funktionalität
POST	Erstellen einer neuen Entität.
GET	Abfragen einer Entität.
PUT, PATCH	Ersetzen oder Aktualisieren einer bereits existierenden Entität.
DELETE	Löschen einer existierenden Entität.

Tabelle 5.5: Zur Verfügung gestellte Operationen einer REST-Schnittstelle

Wie in Abschnitt 2.1.5 beschrieben soll die REST-Architektur im gegebenen Kontext dem *Maturity Model 3* entsprechen. Durch Einführung des *HATEOAS*-Prinzips soll eine bessere Nutzbarkeit der Schnittstelle ermöglicht werden, indem jede Abfrage Verweise auf relevante Ressourcen in Form von Hyperlinks enthält. Dabei hält eine Antwort immer einen Verweis auf die Ressource die abgefragt wurde selbst, sowie zu allen weiteren Ressourcen eine relationale Verknüpfung zu der abgefragten Ressource besitzen. Zur Dokumentation der Schnittstelle wird der Openapi-Standard genutzt. Die gängigsten Frameworks zur Implementierung von REST-Schnittstellen bieten Erweiterungen an um eine Dokumentation gemäß Openapi-Standard zu integrieren. Dabei wird ein eigener Endpunkt zur Einsicht in die Dokumentation bereitgestellt. Über die Dokumentation wird eine interaktive Nutzeroberfläche bereitgestellt die es auch ermöglicht alle Ressourcen direkt zu testen. [77]

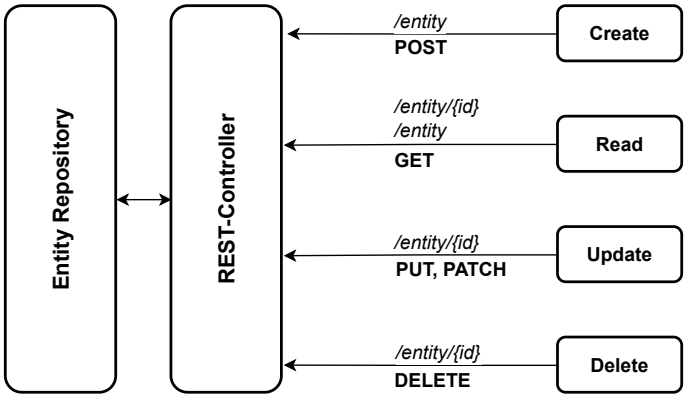


Abbildung 5.11: Übersicht - Verallgemeinerte Darstellung der REST-Endpunkte

Abbildung 5.11 zeigt eine verallgemeinerte Darstellung der zu entwickelnden REST-Endpunkte. Jede Funktionalität wird über eine entsprechende URL angesprochen. Bei den in Abbildung 5.11 dargestellten URLs handelt es sich um relative URLs. Die *Update*- und *Delete*-Operationen beziehen sich immer auf eine einzelne bereits existente Enität, weshalb auch eine ID, welche die Entität im Datenhaltungssystem identifiziert angegeben werden. Mithilfe der *Read*-Operation kann wahlweise eine einzelne Entität oder alle Entitäten abgefragt werden. Um eine neue Entität zu erstellen, wird mittels der *Create*-Operation ein *POST*-Request an den Pfad */entity gesendet*. Dabei spezifiziert der Payload im Body einer solchen Anfrage die zu erstellende Entität.

5.5.3.2 GraphQL-Schnittstelle

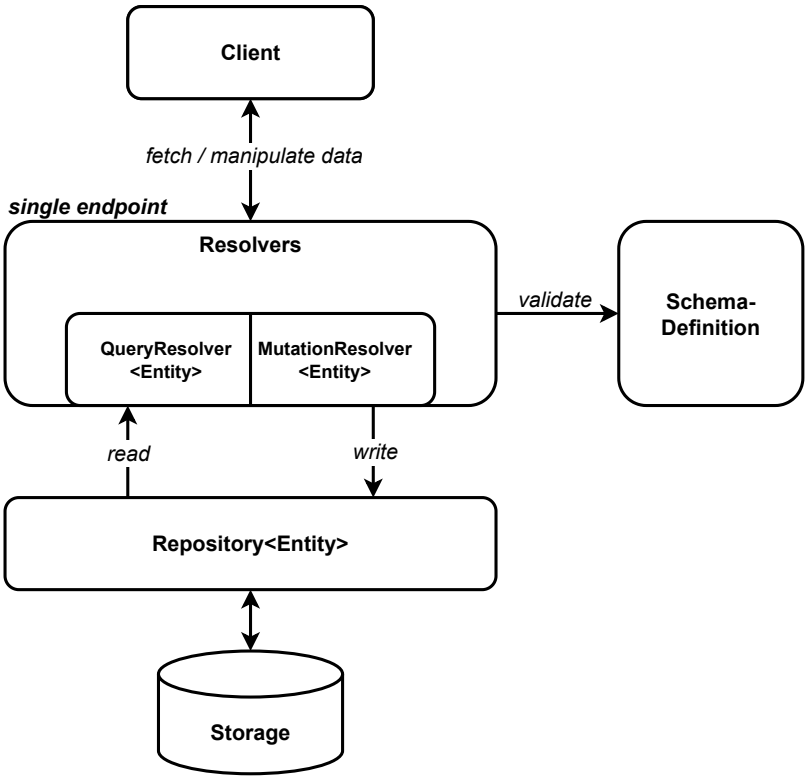


Abbildung 5.12: GraphQL-Architekturkonzept

Im Kern einer GraphQL-Schnittstelle steht die Definition eines Schemas. Das Schema einer GraphQL-Anwendung definiert in separaten Dateien die Gestalt der Daten die abgefragt werden können. Neben einer Objektdarstellung der Daten, definiert ein Schema *Queries* und *Mutations*. Abbildung 5.12 skizziert den Aufbau der GraphQL-Schnittstelle. Die Schema-Definition dient der Typvalidierung von Anfrageergebnissen und Funktionsaufrufen. Eine Resolver-Komponente verarbeitet Anfragen und kapselt die Interaktion mit den entsprechenden Repositories. Ähnlich der REST-Architektur muss für jedes Repository eine passende Resolver-Komponente implementiert werden. Beide Architekturen unterscheiden sich in diesem Punkt jedoch darin, dass GraphQL lediglich einen Endpunkt zur Verfügung stellt. Die Unterscheidung der Entitäten findet auf Serverseite statt, wodurch eine einzige Anfrage ausreicht. Eingehende Anfragen werden bevor sie verarbeitet werden validiert indem sie mit dem Schema abgeglichen werden. Eine weitere Dokumentation ist bei einer GraphQL-Anwendung nicht erforderlich, da diese bereits implizit über die Definition des Schemas erfolgt. Die meisten GraphQL-Implementierungen erlauben aufgrund ihrer introspektiven Natur die Abfrage vorhandener Schemata. [51]

5.5.3.3 Bereitstellung von Metriken

Über einen weiteren Endpunkt stellt die Anwendung Metriken bezüglich ihrer Ressourcenauslastung bereit. Dieser stellt mindestens die Metriken *Speicherauslastung*, *CPU Auslastung* zur Verfügung. Insofern genutzte Frameworks oder externe Monitoring Anwendungen keine Namenskonvention für einen Endpunkt vorgeben, sollen Metriken die Ressource */metrics* bereitgestellt werden.

5.5.3.4 Externalisierte Konfiguration via MQTT

Die Persistierung von Daten im Datenhaltungssystem über die REST- oder GraphQL-Schnittstelle erfordert gegebenenfalls weitere direkte Konfiguration der zugehörigen Geräte im Perception Layer. Um Clients die Konfiguration von Geräten im Perception-Layer zu ermöglichen, wird eine separate Ressource bereitgestellt, über die analog zu einer REST Ressource entsprechende Funktionalität zur Verfügung gestellt wird. Diese umfasst die Registrierung von Sensoren über ein Gateway, das Abmelden von Sensoren über ein Gateway und die Änderung der Group-ID eines EoN-Devices.

5.5.3.5 Zugriffskontrolle

Um den Zugriff auf bestimmte Ressourcen einzuschränken werden Userrollen verwendet. Die Granularität der Zugriffsmöglichkeit sollte dabei so flexibel wie möglich gestaltet werden, mindestens aber die nachfolgenden Rollen definieren.

- **Read Only**
  Diese Rolle erlaubt einem Client lediglich reinen Lesezugriff.
- **Full Access**
  Repräsentiert die Rolle eines Administrators. Diese Rolle hat Zugriff auf den gesamten Funktionsumfang der bereitgestellten API inklusive der Ressourcen zur Konfiguration von Geräten im Perception Layer.
- **Restricted Access**
  Schließt den Zugriff auf die API zum Zugriff auf Perception Layer aus. Ansonsten hat diese Rolle die selben Berechtigungen wie die Rolle Full Access.

Verschiedene Zugriffsstufen werden benötigt, da nicht jeder Client oder jede Anwendung innerhalb des Clusters vollen Zugriff auf jede Funktionalität besitzen soll. Anwendungen, die beispielsweise Sensordaten verarbeiten, benötigen lediglich Lesezugriff um alle benötigten Metadaten abzurufen.

5.5.4 Deployment und Konfiguration

Aufgrund der gegebenen Deployment Optionen wird die Anwendung als Service auf dem von Kubernetes verwalteten Cluster bereitgestellt. Für die Bereitstellung wird das *One-Container-Per-Pod*-Prinzip angewandt. Hierbei steht ein einzelner Pod für eine Instanz der Anwendung, was eine leichtere Verwaltung ermöglicht. Damit eine hohe Verfügbarkeit gewährleistet werden kann, wird die Anwendung nicht durch einzelne Pods zur Verfügung gestellt, sondern über ein *ReplicaSet* gekapselt um sicherzustellen, dass zu jeder Zeit eine festgelegte Anzahl von Pods zur Verfügung steht. Weiterhin wird ein *Deployment*-Objekt, das wiederum das *ReplicaSet* kapselt, verwendet, damit Änderungen in der Anwendung nahtlos ausgerollt werden können. Damit die Anwendung über ihre Schnittstelle außerhalb, sowie innerhalb des Clusters angesprochen werden kann, werden weiterhin *Service* Objekte benötigt, die diese Funktionalität bereitstellen. Für die interne Kommunikation wird lediglich ein Service vom *Typ ClusterIP* benötigt, der eingehende Anfragen von Anwendungen innerhalb des Clusters an eine entsprechende Instanz der Anwendung weiterleitet. Die externe Kommunikation wird durch die Verwendung eines Services vom *Typ NodePort* ermöglicht. Die Anwendung wird auf der IP-Adresse einer jeden Node im Cluster über einen statisch festgelegten Port bereitgestellt. Im gegeben

Kontext ist diese Möglichkeit ausreichend, jedoch sollte im Produktivbetrieb für die externe Bereitstellung ein Service Objekt vom *Typ LoadBalancer* verwendet werden. Diese Konfiguration erfordert weiteren Overhead, da diese Art Service weitere Abhängigkeiten in Form einer *externen Load Balancer* Anwendung erfordert. Da Kubernetes je nach Konfiguration nur rudimentäres Load Balancing betreibt, indem Pods über das *Round Robin Prinzip* oder *zufällig* ausgewählt werden, ist ein solches Szenario bei einer extrem hohen Auslastung denkbar. Ein vollständiges Deployment, sowie die benötigten Services werden entsprechend der gängigen Vorgehensweise in einem Konfigurationsfile im *YAML Format* definiert und anschließend über das Kubernetes eigene *CLI Tool kubectl* ausgerollt. [16]

Die Konfiguration der Anwendung erfolgt ähnlich der Anwendung zur Sensoranbindung über eine in einem strukturierten Textformat definierten Konfiguration. Diese enthält anwendungsspezifische Konfiguration, abhängig von in der Implementierung genutzten Bibliotheken und Daten zur Verbindung mit Message-Broker und Datenbank. Da die Anwendung im Clusterkontext betrieben wird und somit auch mehrere Instanzen der selben Anwendung konfiguriert werden müssen, wird zur Vermeidung von Mehraufwand in der Konfiguration das Prinzip der externalisierten Konfiguration genutzt. Kubernetes bietet zur Entkopplung der Konfiguration einer Anwendung die Möglichkeit, konfigurationsrelevante Daten in entsprechende Kubernetes-Objekten, namentlich *Secrets* und *ConfigMaps*, auszulagern. [14] Das in Abbildung 5.13 aufgeführte Schema ordnet die Konfigurationsdaten inhaltlich verschiedenen Objekten zu. Sensible Daten wie Zertifikate oder Passwörter werden in Secrets statt ConfigMaps persistiert, da diese zumindest eine rudimentäre Base64-basierte Verschlüsselung nutzen.

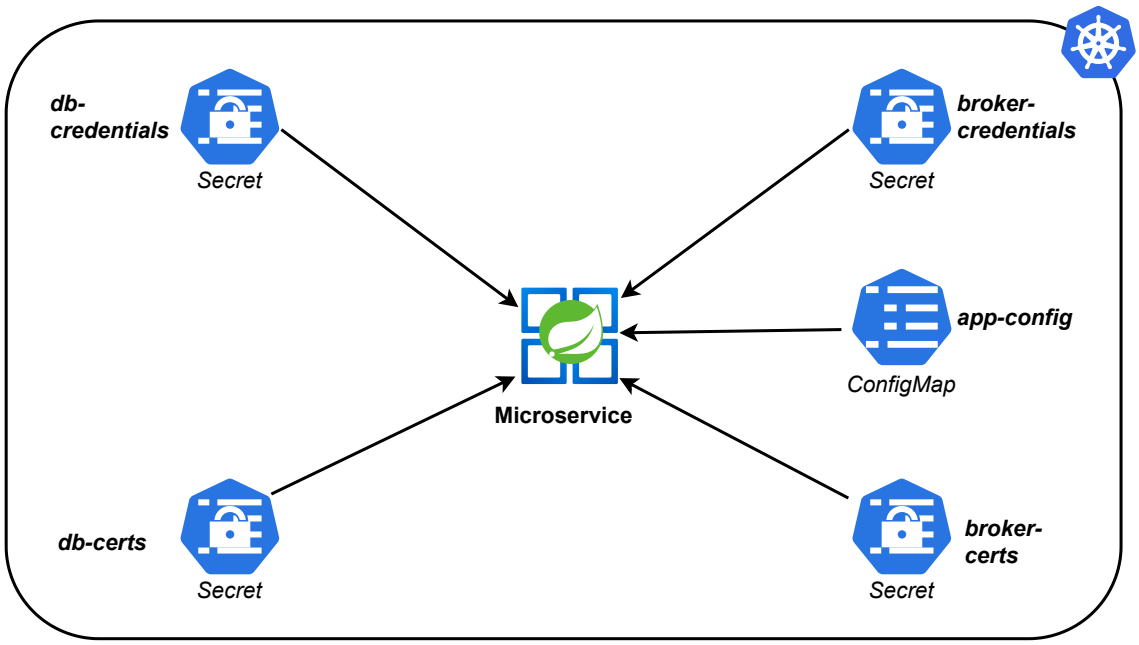


Abbildung 5.13: Schematische Darstellung einer externalisierten Konfiguration

5.5.5 Logging

Logging stellt einen essentiellen Aspekt einer jeden Anwendung dar. Log Ausgaben können rein informativer Natur sein oder eine fehlerhafte Ausführung signalisieren. Damit

eine inhaltliche Kategorisierung, sowie eine angemessene Angabe des Schweregrades einer Log-Ausgabe möglich ist, werden die gängigen im Nachfolgenden aufgeführten Log-Level unterstützt. [10]

- **fatal**
  Schwerwiegende Fehler, die zu einem vorzeitigen Abbruch führen. Fehler dieser Kategorie werden auf dem *stderr* Stream ausgegeben.
- **error**
  Signalisiert Laufzeitfehler und wird ebenfalls auf dem *stderr* Stream ausgegeben.
- **warn**
  Dieses Log Level signalisiert beispielsweise die Verwendung veralteter APIs oder andere unerwünschte Laufzeitereignisse. Die Ausgabe erfolgt auf dem Standardstream *stdout*.
- **info**
  Signalisiert relevante Laufzeitereignisse wie beispielsweise Startup oder Shutdown Events. Logs dieser Kategorie werden ebenfalls auf dem Standardstream *stdout* ausgegeben.
- **debug**
  Dieses Log Level dient als Hilfsmittel in der Entwicklung um detaillierte Informationen zum Ablauf eines Systems auszugeben. Die Ausgabe erfolgt ebenso auf dem Standardstream *stdout*.
- **trace**
  Logs dieser Kateogorie sollen detailliertere Ausgaben als das vorige Level ausgeben. Diese dienen zur Diagnose eines Systems und sollen möglichst detailliert den gesamten Programmablauf abbilden. Die Ausgabe erfolgt auf dem Standardstream *stdout*.

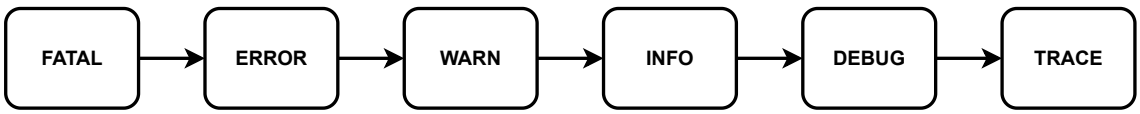


Abbildung 5.14: Log Level Hierarchie

Durch Konfiguration eines globalen Log-Levels bei der Initialisierung wird gesteuert, welche Log-Nachrichten ausgegeben werden. Alle Log-Level bilden eine hierarchische Ordnung. Bei der globalen Konfiguration werden alle in der Hierarchieordnung untergeordneten Level, in der Ausgabe ignoriert. Im produktiven Betrieb wird das Level *Info* verwendet. In der Entwicklung werden wahlweise die Level *Debug* oder *Trace* verwendet. Grundsätzlich erfolgt die Ausgabe aller Log-Nachrichten auf den Standardstreams *stderr* und *stdout*.

6 Implementierung

6.1 Datenbank

Anhand der aus der Konzeption hervorgehenden Entscheidung eine SQL-kompatible, sowie skalierbare Datenbanklösung zu verwenden, fiel die Entscheidung auf *CockroachDB*. Diese Entscheidung ist darauf begründet, dass CockroachDB eine umfassend dokumentierte Datenbanklösung anbietet, die nahtlos in eine bestehende Clusterumgebung integriert werden kann und alle gängigen Deployment-Möglichkeiten unterstützt. In der Entwicklung der Kern-Funktionalität verfolgt CockroachLabs das Konzept der Open-Source-Entwicklung, was ein ebenso wichtiger Aspekt im Rahmen der Lehre und Forschung in Bezug auf das Thema Smart City darstellt. Weiterhin ist eine konsistente Sicht der Daten garantiert. [12]

6.1.1 Datenbankschema

Das in Abbildung 6.1 dargestellte Datenbankschema wird in der Anwendung umgesetzt. Es muss lediglich eine Datenbank und ein User erstellt werden um Zugriff auf die Datenbank zu erhalten. Die Erstellung aller nötigen Tabellen wird durch die Microservice-Anwendung durchgeführt.

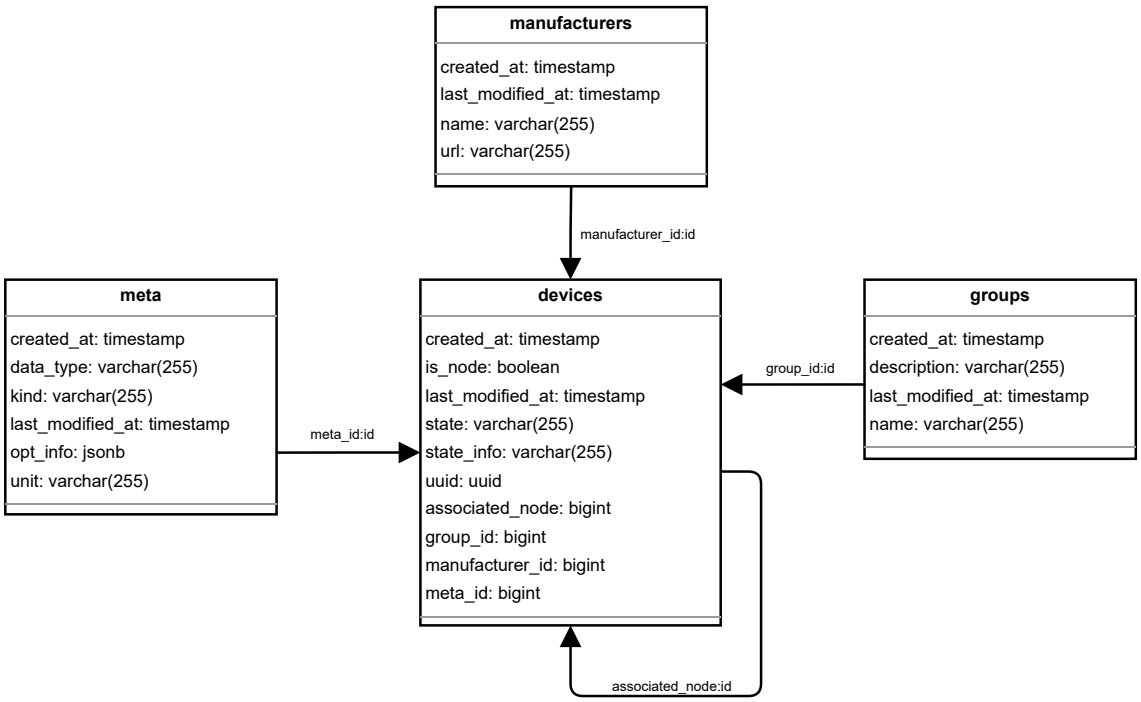


Abbildung 6.1: ER-Diagramm des verwendeten Datenbankschemas

6.1.2 Deployment

Das Deployment eines CockroachDB-Clusters kann auf einfache Weise durch *Helm-Charts* oder mithilfe des *Kubernetes-Operator* durchgeführt werden. Aufgrund des einfacheren Deployments des Datenbank-Clusters wurde dies mithilfe des von CockroachLabs bereitgestellten Kubernetes-Operator durchgeführt. Hierbei wurde im ersten Schritt eine *CustomResourceDefinition* heruntergeladen und mittels *kubectl* im Cluster erstellt. Die *CustomResourceDefinition* stellt eine Erweiterung der Kubernetes-API dar und erlaubt es eigene Kubernetes-Objekte zu definieren. Weiterhin wird der Namespace *cockroach-operator-system* erstellt in welchem alle relevanten Kubernetes-Objekte erstellt und verwaltet werden. Im nächsten Schritt wird ein *CrdbCluster*-Object deployed, welche alle vom *Operator* benötigten Informationen enthält um den Cluster zu initialisieren. Bei Bedarf kann das *CrdbCluster*-Object angepasst werden, um beispielsweise zu spezifizieren, welche CPU- und Speicherressourcen der Cluster verwenden darf. Der Operator initialisiert alle benötigten Kubernetes-Objekte.

Um auf eine Instanz Zugriff zu erhalten wird über den integrierten SQL-Client in der initialen Konfiguration ein User mit entsprechenden Rechten erstellt. Der Zugriff erfolgt mittels *kubectl*, indem über eine Shell-Instanz im Pod *cockroach-db-client-secure* die mitgelieferte CLI-Anwendung geöffnet wird. Nach vollständiger Konfiguration kann die Datenbank über die bereitgestellten Services angesprochen werden. Weiterhin stellt das Deployment eine Web-Anwendung bereit, welche Zugriff auf ein Dashboard mit umfangreichen Metriken zum gesamten Cluster-Deployment erlaubt. [8] Der Deployment Prozess kann in Anhang A.5 nachvollzogen werden.

6.2 Gateway - Sensoranbindung

Um eine einfache, zugängliche und erweiterbare Codebasis zur Verfügung zu stellen, wird die Software zur Anbindung und Verwaltung der Sensoren und Aktoren in der Programmiersprache Python umgesetzt. Python gilt als einsteigerfreundliche und verständliche Programmiersprache. Zur Interaktion mit Sensoren und Aktoren über das GrovePI+ Board wird von Herstellerseite bereits eine umfängliche Programmbibliothek zur Verfügung gestellt. Diese bietet Basisfunktionalitäten wie Lese- und Schreiboperationen an, als auch gerätespezifische Funktionen, die auf der Basisfunktionalität aufbauen. [45] Unter diesen Aspekten bietet sich die Implementierung in Python an, um die in Abschnitt 5.4 konzipierte Client-Anwendung umzusetzen.

6.2.1 Controller zur Hardwareinteraktion

An das GrovePi+-Board angeschlossene Geräte, können über die mitgelieferte Programmbibliothek angesprochen werden. Daten können über Leseoperationen ausgelesen werden oder Steuerbefehle können über Schreiboperationen an ein Gerät übermittelt werden. Da komplexere Geräte in manchen Fällen mehrere aufeinanderfolgende Schreib- oder Leseoperationen erfordern, existieren sensorspezifische Funktionen, die die Funktionalität für ein bestimmtes Gerät kapseln. Diesbezüglich müssen für verschiedene Geräte, verschiedene Controller implementiert werden, um deren Funktionalität zu nutzen. Für einen generischen Aufbau wird für Geräte des Typs *Aktor*, als auch für Geräte des Typs *Sensor* jeweils ein Interface bereitgestellt, auf dessen Basis die notwendigen Controller implementiert werden können. Somit kann die Codebasis für beliebige Geräte erweitert

werden.

```
from abc import ABCMeta, abstractmethod
from utilities import PortType

class ISensorController(metaclass=ABCMeta):

    @abstractmethod
    def __init__(self, port_type: PortType, pin: int):
        self.__port_type = port_type
        self.__pin = pin

    @property
    @abstractmethod
    def port_type(self) -> PortType:
        pass

    @port_type.setter
    def port_type(self, value):
        self.__port_type = value

    @property
    @abstractmethod
    def pin(self) -> int:
        pass

    @pin.setter
    def pin(self, value):
        self.__pin = value

    @abstractmethod
    def read(self):
        pass
```

Listing 6.1: Generisches Interface als Grundlage zur Implementierung eines Controllers

Listing 6.1 zeigt das Interface für Sensoren. Hierbei werden für einen Controller erforderliche Methoden definiert. Aufgrund der dynamischen Typisierung der Programmiersprache Python existiert kein klassisches Interface-Konzept. Hier wird auf das Konzept von abstrakten Klassen zurückgegriffen, um diese Funktionalität umzusetzen.[2] Mit deren Hilfe kann eine Basisklasse erstellt werden, welche alle Attribute und Methoden definiert, die eine Controller-Klasse bereitstellen muss. Es werden jeweils die Interfaces *IActorController* für Aktoren und respektive *ISensorController* für Sensoren zur Verfügung gestellt. Die meisten Geräte können über einfache Lese- und Schreiboperationen angesprochen werden. Diese Funktionalität wird in der mitgelieferten Programmbibliothek des GrovePi+-Boards über die folgenden Methoden bereitgestellt:

- analogRead(pin)
- analogWrite(pin, value)
- digitalRead(pin)
- digitalWrite(pin, value)

Durch Verwendung dieser Methoden, sowie der Unterscheidung des Gerätetyps wurden die generischen Controller *GenericSensorController* und *GenericActorController* umgesetzt. Beide Controller implementieren respektive, die durch das jeweilige Interface

definierten Methoden. Der in Listing 6.2 aufgeführte Codeabschnitt repräsentiert exemplarisch die Implementierung des generischen Sensor-Controllers.

```
from controller.interfaces.isensor_controller import ISensorController
from utilities import PortType
from grovepi import analogRead, digitalRead

class GenericSensorController(ISensorController):

    def __init__(self, pin: int, port_type: PortType, **kwargs):
        self.__pin = pin
        self.__port_type = port_type

    def read(self):
        if self.port_type == PortType.analog:
            return analogRead(pin=self.pin)
        elif self.port_type == PortType.digital:
            return digitalRead(pin=self.pin)
```

Listing 6.2: Generischer Controller zur Interaktion mit Sensoren

Die GrovePi-Bibliothek bietet neben den genannten Standardmethoden zum Lesen und Schreiben noch weitere gerätespezifische Methoden zur Interaktion. Diese können zur Implementierung der Controller genutzt werden. [45]

6.2.2 Netzwerkkommunikation

Controller für Sensoren und Aktoren bilden auf unterster Ebene eine Schnittstelle zur Interaktion mit den entsprechenden Geräten. Damit angeschlossene Geräte in das Sensornetzwerk eingebunden werden können, muss Funktionalität zur Kommunikation mit dem Message-Broker bereitgestellt werden. Analog zu den *Controller-Interfaces*, wird das Interface *IMqttClient* implementiert. Durch Verwendung der Python-Implementierung des *Eclipse-Paho*-Client wird die Funktionalität zur Interaktion mit dem Message-Broker bereitgestellt. Weiterhin definiert das in Listing 6.3 dargestellte Interface, benötigte Methoden, die die in Abschnitt 5.4.2 vorgestellte Logik zur Bereitstellung eines Gerätes implementieren.

```
from abc import ABCMeta, abstractmethod
import paho.mqtt.client as paho

class IMQTTClient(paho.Client, metaclass=ABCMeta):

    @abstractmethod
    def __init__(self, client_id="", clean_session=None,
                 userdata=None, protocol=paho.MQTTv311,
                 reconnect_on_failure=True):
        paho.Client.__init__(self,
                             client_id=client_id,
                             clean_session=clean_session,
                             userdata=userdata,
                             protocol=protocol,
                             reconnect_on_failure=reconnect_on_failure)

    @paho.Client.connect_callback
    @abstractmethod
    def on_connect(self, client, userdata, flags, rc):
        pass
```

```
21                 ...
22                 # Other on_foo methods as specified by the paho.Client class are
       specified
23     ...
24                 @abstract_method
25     def handle_config_message(self, client, userdata, message):
26         pass
27
28                 @abstract_method
29     def handle_command_message(self, client, userdata, message):
30         pass
31
32                 @abstract_method
33     def set_lwt(self):
34         pass
35
36                 @abstract_method
37     def birth(self):
38         pass
39
40                 @abstract_method
41     def publish_state(self, state: str, opt_info: str):
42         pass
```

Listing 6.3: Interface zur Implementierung des MQTT-Clients

Eingehende Nachrichten benötigen Callback-Methoden, welche die Logik implementieren wie diese zu verarbeiten sind. Methoden mit dem Präfix *handle_* definieren Callback-Methoden für Nachrichten des Typs *CONFIG* und *CMD*. Bei der Initialisierung der Anwendung werden Message-Handler für alle relevanten Topics registriert. Vollständig initialisierte Geräte müssen ein Birth-Zertifikat auf dem dafür vorgesehenen Topic publizieren. Die dafür benötigte Logik wird in der Funktion *birth* umgesetzt. Meldet sich ein Gerät vom System ab oder verliert die Verbindung, sendet das Gerät ein Death-Zertifikat um zu signalisieren, dass es nicht mehr verfügbar ist. In der Funktion *set_lwt* wird Logik implementiert mittels der eine LWT-Nachricht festgelegt wird. Nach dem *Report By Exception*-Prinzip publiziert ein Gerät seinen Status über den Nachrichtentyp *STATE*.

Zusätzlich werden Callback-Methoden spezifiziert, welche verschiedenen Events im Lebenszyklus eines MQTT-Clients entsprechen. Dabei handelt es sich um alle Methoden die mit dem Präfix *on_* beginnen. Diese werden in der Implementierung in erster Linie zur Unterstützung der Entwicklung verwendet, indem Log-Ausgaben in den einzelnen Callback-Methoden spezifiziert werden, damit der Empfang, das Versenden und Initiieren von Subscriptions mittels Log-Nachrichten besser nachvollzogen werden kann. [5]

Weiterhin erfordert die Broker-Kommunikation in der Produktivumgebung des CiTe-Testfelds die Authentifizierung des Clients mittels einer Zertifikatskette, sowie die Angabe eines Usernamen und Passworts. Hierzu muss zunächst der SSL-Kontext initialisiert werden, wofür drei verschiedene Dateien im *.pem-Format* benötigt werden. Das *CA-Zertifikat* dient der Identifikation der *Certificate Authority*. Zur Authentifizierung gegenüber dem Server werden das *Client-Zertifikat* sowie ein Private-Key verwendet. Der SSL-Kontext muss beim Verbindungsaufbau explizit gesetzt werden. Listing 6.4 zeigt den Aufbau einer verschlüsselten Verbindung unter Verwendung der Python-Implementierung des *Eclipse-Paho*-Clients.

```
1 import ssl
2 import paho.mqtt.client as paho
```

```
3
4  def ssl_context(ca_cert: str, client_cert: str, client_key: str) -> ssl.SSLContext:
5      context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
6      context.load_verify_locations(ca_cert)
7      context.load_cert_chain(client_cert, client_key)
8      return context
9
10 context = ssl_context(ca_cert,client_cert, client_key)
11 client = paho.Client()
12 client.tls_set_context(ssl_ctx)
13 client.username_pw_set(username=_username, password=_password)
14 client.tls_insecure_set(True)
15 client.connect(host, port)
```

Listing 6.4: Aufbauen einer verschlüsselten Verbindung über den Eclipse-Paho-Client

6.2.3 Initialisierung

Im gegebenen Kontext werden zur besseren Abstraktion zwei verschiedene Geräte differenziert. Zum einen der Raspberry Pi als Gateway, und zum anderen die einzelnen Sensoren und Aktoren die über das Gateway verwaltet werden. Respektive wurde gemäß dem Interface zur Broker-Kommunikation jeweils eine Implementierung für das Gateway und eine für anzubindende Geräte umgesetzt. Die Komponente *DeviceManager* kapselt einen Controller zur Interaktion mit einem Gerät und stellt selbst die Funktionalität zur Kommunikation über den Message-Broker bereit. Jedes angebundene Peripheriegerät wird von einer eigenen *DeviceManager*-Instanz verwaltet.

Analog wird für das Gateway eine *GatewayManager*-Instanz initialisiert, welche wiederum Referenzen auf eine oder mehrere *DeviceManager*-Instanzen hält. Wie bereits in Abschnitt 5.2.1.1 beschrieben, benötigt jedes im System bereitgestellte Gerät eine eindeutige Kennung. Die Generierung einer eindeutigen Kennung erfolgt bei Initialisierung einer *DeviceManager*- oder *GatewayManager*-Instanz. Als Grundlage dient in diesem Fall die *Serial-ID* des Raspberry Pis. Dabei handelt es sich um eine global eindeutige, vom Hersteller vergebene Kennung, die fest in die Hardware integriert wurde. Die Serial-ID ist im Dateisystem eines Raspberry Pis unter */proc/cpuinfo* hinterlegt und kann so auf einfache Weise ausgelesen werden. Listing 6.5 zeigt die verwendete Funktion, um diese auszulesen. Mithilfe der Serial-ID wird so für das Gateway und für jedes Peripheriegerät eine eindeutige ID generiert, insofern keine ID über eine Konfigurationsnachricht oder Backup-Konfiguration ausgelesen wurde.

Im Anschluss wird die MQTT-Client-Verbindung hergestellt und die Subscriptions für den Empfang von Steuerungs- und Konfigurationsbefehlen, respektive für alle *DeviceManager*- und *GatewayManager*-Instanzen initialisiert. Anzumerken ist, dass die LWT-Nachricht mit der für den Versand von *DDEATH*- und *NDEATH*-Nachrichten gesetzt wird bevor die MQTT-Client-Instanz über die *connect*-Methode eine Verbindung zum Broker aufbaut. Sobald die Verbindung zum Broker besteht senden *DeviceManager*- und *GatewayManager*-Instanzen, das über die *birth*-Methode spezifizierte Birth-Zertifikat über *DBIRTH*- und *NBIRTH*-Nachrichten. Nach Senden des Birth-Zertifikats ist eine Instanz voll funktionsfähig und ist in der Lage gelesene Daten zu senden oder empfangene Daten zu verarbeiten.

```
1 ERROR_SERIAL = "ERROR000000000"
2
3 def uuid_from_serial() -> UUID:
```

```
4     serial = pi_serial()
5     if serial == ERROR_SERIAL:
6                                 # Fallback -> Use uuidv4 instead
7         return uuid.uuid4()
8     return uuid.uuid1(serial)
9
10 def pi_serial() -> str:
11     cpuserial = "0000000000000000"
12     try:
13         f = open('/proc/cpuinfo', 'r')
14         for line in f:
15             if line[0:6] == 'Serial':
16                 cpuserial = line[10:26]
17         f.close()
18     except:
19         cpuserial = ERROR_SERIAL
20     return cpuserial
```

Listing 6.5: Abfrage der Serial-ID und Generierung der UUID

6.2.4 Konfiguration und Backup

6.2.4.1 Allgemeine Konfiguration

Die Konfiguration der Anwendung erfolgt über eine lokal im Dateisystem hinterlegte Konfigurationsdatei. Diese spezifiziert, wie bereits in Listing 5.3 gezeigt, die benötigten Daten, um eine Verbindung mit dem Message-Broker aufzubauen, sowie alle nötigen Daten zur Konfiguration der anzubindenden Geräte.

Um einen einfachen Zugriff auf die Konfiguration innerhalb der Anwendung zu ermöglichen wurde eine Klasse implementiert, welche in der Lage ist, eine *Singleton-Instanz* zu erzeugen, welche die Konfigurationsdatei kapselt. Dabei werden die über das JSON-Format definierten Key-Value-Paare mittels der Datenstruktur *Dictionary* repräsentiert. Die Konfigurationsklasse bietet zum einfacheren Zugriff Funktionalität, um auf mehrfach geschachtelte Attribute zuzugreifen. Weiterhin wird die Konfiguration eines neu initialisierten Gerätes in der Konfigurationsdatei persistiert. Durch dieses Backup kann bei Neustart der Anwendung die zuvor gespeicherte Gerätekonfiguration geladen werden und der ursprüngliche Zustand wiederhergestellt werden.

6.3 Microservice - Metadatenverwaltung

Zur Umsetzung des Microservice wird die Programmiersprache Java verwendet. Die Implementierung erfolgt unter Verwendung des Spring Boot-Frameworks sowie entsprechender Erweiterungen. Abhängigkeiten in der Implementierung des Microservices werden über *Apache Maven* integriert. Alle benötigten Abhängigkeiten werden hierzu zentral in der *POM-Datei* angegeben.

6.3.1 Datenbankschnittstelle

Die Implementierung der Repositories für die Datenzugriffschicht werden mit der *Spring-Data-JPA*-Integration umgesetzt. Die *Java Persistence API (JPA)* spezifiziert die Interaktion mit Datenhaltungssystemen über einfache *Plain Old Java Objects (POJOs)*. Dabei werden

Felder einer Datenbanktabelle auf diese abgebildet, weshalb dieser Prozess auch als *Object Relational Mapping (ORM)* bezeichnet wird. Auf diese Weise können, nach dem Paradigma der objektorientierten Programmierung, Daten in einer Datenbank manipuliert oder abgefragt werden. Damit dieses Prinzip nahtlos in eine Spring-Anwendung integriert werden kann, stellt Spring die Erweiterung *Spring-Data-JPA* über die *Spring-Starter* zur Verfügung. Diese Abhängigkeit beinhaltet die JPA-Implementierung *Hibernate*, sowie weitere Funktionalität die die Integration einfacher gestaltet. Eingebunden wird die Erweiterung via Apache Maven indem, wie in Listing 6.6 aufgeführt, die *Spring Starter JPA*-Abhängigkeit in der pom.xml Datei angegeben wird.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Listing 6.6: Spring Data Jpa Maven Dependency

Die Datenzugriffschicht implementiert zunächst alle Objekte welche eine Entität des Datenmodells darstellen. Diese werden mit der Annotation *@Entity* annotiert, um diese für Hibernate als relevante Objekte zu kennzeichnen. Um Boilerplate-Code zu vermeiden wird die Bibliothek *Lombok* verwendet. Obligatorische Methoden wie Getter, Setter und der Konstruktor müssen nicht von Hand implementiert werden. Durch Angabe der Annotationen *@Getter*, *@Setter* und *@AllArgsConstructor* zur Kompilierzeit generiert, wodurch keine Performanceeinbußen entstehen.

Das Mapping einer Entity auf die entsprechende Tabelle einer relationalen Datenbank, kann mithilfe der optionalen Annotation *@Table* angegeben werden. Anzumerken ist an dieser Stelle, dass alle in Abbildung 5.4 dargestellten Entitäten bis auf *OptMeta* einem Objekt zugeordnet werden. OptMeta repräsentiert optionale Metadaten und wird als Attribut der Entität *MetaData* definiert. Dies konnte mittels des durch CockroachDB zur Verfügung gestellten Datentyp JSONB realisiert werden. Dieser erlaubt ähnlich eines dokumentenbasierten Datenhaltungssystem die Angabe von Daten in JSON-Format. Die Entscheidung fiel auf diese Lösung, da der durch die optionalen Metadaten repräsentierte Wert durch einen beliebigen, oder in diesem Kontext, einen durch die JSON-Spezifikation spezifizierten Datentyp repräsentiert werden kann. Alternativ würden Werte als String-Datentyp repräsentiert und auf Serverseite in den eigentlichen Datentyp umgewandelt werden. Auf diese Weise kann jedoch eine weitere serverseitige Verarbeitung eines Wertes vermieden und weitere Metadaten dynamisch ergänzt werden.

Im nächsten Schritt werden respektive zu den implementierten Entitäten, Repositories ergänzt. Deren Aufgabe ist es, konkrete, in Entitäten gekapselte Daten zu persistieren oder bei Abfrage von Daten in eben diese zu schreiben. Hierzu stellt *Spring Data JPA* das Interface *JPARepository* zur Verfügung. Ein Repository wird als Interface deklariert und erweitert, wie im Beispiel des in Listing 6.7 implementierten Repository der Group-Entität, das Interface *JPARepository*.

```
package htw.saar.deviceconfig.jpa.repository;

import htw.saar.deviceconfig.jpa.entity.Group;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import java.util.Optional;

public interface GroupRepository extends JpaRepository<Group, Long> {

    Group findByName(String name);

    Optional<Group> findById(long id);
}
```

Listing 6.7: Implementierung von Repositories mit Spring Data JPA

Spring-Data-JPA ermöglicht es Datenbankabfragen über die Angaben von Methoden im Interface zu implementieren. Standardmäßig stellt ein *JPARepository*-Methoden zur Persistierung, das umfasst die Erstellung als auch die Änderung sowie das Löschen und Abrufen von Entitäten zur Verfügung. Eine Besonderheit stellt hierbei die Möglichkeit komplexere Abfragen zu definieren, indem lediglich die Methodensignatur in der Interface Klasse angegeben wird. Dabei spielt die Benennung einer solchen Methode eine entscheidende Rolle, da *Spring Data JPA* in der Lage ist anhand bestimmter Keywords im Methodennamen die Funktionalität abzuleiten. Grundsätzlich bestehen diese als *Derived Queries* bezeichneten Methoden aus zwei Komponenten. Die erste Komponente, der *Introducer* definiert die Art der Abfrage und kann eines der Stichworte *find*, *exists*, *count* oder *delete* sein. Dem *Introducer* folgt das *Kriterium*, welches angibt, auf welchem konkreten Feld oder welchen Feldern eine Operation ausgeführt wird. Dadurch kann die Funktionalität eines Repositories beliebig erweitert werden. Neben der Möglichkeit Abfragen selbst zu definieren stellt das *JPARepository* bereits grundlegende Funktionalität zur Verfügung. [9]

6.3.2 REST - Schnittstelle

6.3.2.1 Repository Erweiterung mit Spring-Data-Rest

Eine REST-Schnittstelle kapselt wie bereits in Abschnitt 2.1.5 im Detail beschrieben, Funktionalität zum Abruf von Daten aus einem Datenhaltungssystem und stellt Ressourcen bereit die über HTTP-Requests angesprochen werden können. Zur Umsetzung der Schnittstelle wurde ebenfalls eine Erweiterung des Spring Frameworks verwendet. Wie Listing 6.8 zu entnehmen wird die Erweiterung *Spring Data REST* als weitere Abhängigkeit in der pom.xml Datei ergänzt.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

Listing 6.8: Maven Dependency zur Einbindung von Spring Data Rest

Spring-Data-Rest erlaubt es, bestehende *JPA-Repositories* zu erweitern, sodass REST-konforme Ressourcen zur Verfügung gestellt werden. Eine solche Schnittstelle entspricht dem in Abschnitt 2.1.5 vorgestellten *Maturity Model 3* und verwendet das *HAL-Format* für JSON-Output, der über die bereitgestellten Endpunkte zurückgegeben wird. Die Erweiterung erlaubt mit Hilfe einer *RepositoryDetectionStrategy*, welche in der Konfiguration einer Spring Anwendung angegeben wird, zu definieren anhand welcher Kriterien Repositories als REST-Ressource bereitgestellt werden. Um eine flexible Erweiterung zu ermöglichen,

wird die Strategie *ANNOTATED* verwendet. [74]

```
...
@RepositoryRestResource(collectionResourceRel = "groups", itemResourceRel = "group",
    path = "groups")
public interface GroupRepository extends JpaRepository<Group, Long> {
...
}
```

Listing 6.9: Bereitstellung eines Repository als Rest-Ressource

Durch die Strategie *ANNOTATED* werden lediglich Repositories, die wie in Listing 6.9 dargestellt, mit der Annotation *@RepositoryRestResource* versehen sind, berücksichtigt. Der Parameter *path* spezifiziert das Schlüsselwort das in der URL der Ressource genutzt wird. Die Ressource um Group-Entitäten abzurufen wäre dementsprechend über die URL *http://localhost:8080/api/groups* zu erreichen.

```
1  // GET Request: http://localhost:8080/api/groups/1
2  "_links": {
3      "self": {
4          "href": "http://localhost:8080/api/groups/1"
5      },
6      "group": { <-- itemResourceRel
7          "href": "http://localhost:8080/api/groups/1"
8      }}
9  // GET Request: http://localhost:8080/api/groups
10 "_links": {
11     "self": {
12         "href": "http://localhost:8080/api/groups"
13     },
14     "groups": { <-- collectionResourceRel
15         "href": "http://localhost:8080/api/groups"
16     }}
```

Listing 6.10: HAL- Verweise auf Collection- und Item-Ressourcen in einer Serverantwort

Gemäß des HATEOAS-Prinzip enthält jede Antwort auf eine Anfrage Verweise in Form von Hyperlinks auf die zugehörige *Collection*-Ressource, sowie *Item*-Ressource, insofern eine einzelne Entität abgefragt wurde (siehe Listing 6.10). Die *Collection*-Ressource beschreibt den Endpunkt unter dem eine Menge einer Entität abgerufen werden kann, wohingegen eine *Item*-Ressource die Abfrage einer einzelnen Entität anhand ihrer ID ermöglicht.

6.3.2.2 Automatisch generierte REST-Ressourcen

Die automatisch generierten Endpunkte für eine Ressource werden intern über einen Controller verwaltet. Für jede Entität wird der in Tabelle 6.1 dargestellten Einteilung ein Controller generiert, der eine Menge von Endpunkten spezifiziert.

Controller	Device	Group	Manufacturer	MetaData
Entity *(GET, POST, PUT, PATCH, DELETE)*	⇒/devices ⇒/devices/{id}	⇒/groups ⇒/groups/{id}	⇒/manufacturers ⇒/manufacturers/{id}	⇒/meta ⇒/meta/{id}
Search *(GET)*	⇒/devices/ /search/{foo}?{param}={value}	⇒/groups/ /search/{foo}?{param}={value}	⇒/manufacturers/ /search/{foo}?{param}={value}	⇒/meta/ /search/{foo}?{param}={value}
Property Reference *(GET, PUT, POST, DELETE)*	⇒/devices/{id}/{property}	Keine Relationen zu anderen Entitäten	Keine Relationen zu anderen Entitäten	Keine Relationen zu anderen Entitäten

Tabelle 6.1: Automatisch generierte REST-Endpunkte pro Entität

In Tabelle 6.1 sind die generierten Endpunkte respektive für alle Entitäten dargestellt. Der Entity-Controller stellt jeweils einen Endpunkt zur Interaktion mit einzelnen Entitäten und einen weiteren zur Interaktion allen verfügbaren Entitäten zur bereit. Um mit einzelnen Entitäten zu interagieren muss die ID dieser als Path-Parameter angegeben werden. Analog zur klassischen CRUD-Funktionalität können Entitäten durch Verwendung der unterschiedlichen HTTP-Verben erstellt, gelesen, geändert, ersetzt oder gelöscht werden.

Zur Ausführung von komplexeren Leseoperationen, stellt der *Search-Controller* Endpunkte zur Verfügung um Ergebnisse anhand von einem oder mehreren Attributen einer Entität zu filtern. Dabei setzt sich die URL aus dem Namen der zuvor im entsprechenden *JPARepository* definierten Methode als *Path-Parameter* und den benötigten Parametern als *Query-Parametern* zusammen. Da es sich bei den Endpunkten des Search-Controller ausschließlich um Leseoperationen handelt muss eine Anfrage ein GET-Request sein.

Der Property-Reference-Controller dient zur Abfrage von Assoziationen einer Entität. Die URL erfordert hierbei die Angabe der ID, sowie der Name der gewünschten Assoziation einer Entität als Path-Parameter. Auf diese Weise kann einfach auf Entitäten, wie beispielsweise *Group*, *Manufacturer* und *Meta*, die in Relation zur Entität Device stehen, zugegriffen werden.

6.3.2.3 Interaktion mit Peripheriegeräten

Zusätzlich zur automatisch generierten REST-Schnittstelle erfordert die Registrierung und Verwaltung von Peripheriegeräten das Senden von Nachrichten über den Message-Broker. Die Kommunikation erfolgt über die in Listing 5.2 vorgestellte, generische Payloadstruktur, dementsprechend obliegt die Verarbeitung der Nachrichten der Instanz, welche die Nachricht empfängt. Im Kontext des CiTe-Testfeldes können Nachrichten an den Raspberry Pi gesendet werden um Grove-Sensoren und -Aktoren zu initialisieren, entfernen oder zu konfigurieren. Die Funktionalität zur Kommunikation mit dem Raspberry-Gateway ergänzt den zu Grunde liegenden Device-Controller und stellt die folgenden Endpunkte zur Verfügung:

- **/devices/id/register** Dieser Endpunkt bezieht sich auf ein Gerät, dass als Gateway fungiert. Im Kontext des CiTe-Testfelds nimmt dieser Endpunkt die Konfiguration eines Sensors oder Aktors entgegen übergibt diese der zuständigen *DeviceManager*-Instanz. Die dabei benötigten Daten erfordern die Angabe des *Hardware-Pins*, der *Geräteart (sensor, actor)* und ein *Name*.

- **/devices/id/delete** Mit diesem Endpunkt kann ein durch ein Gateway verwaltetes Gerät von diesem abgemeldet werden. Hierfür wird die ID des Geräts übermittelt,

welche im Anschluss vom Gateway verarbeitet wird, indem dieses die *DeviceManager*-Instanz terminiert.

- **/devices/id/config** Über diesen Endpunkt können weitere Konfigurationsnachrichten verschickt werden insofern entsprechende Funktionen vom Empfänger bereitgestellt werden.

Die Struktur und Inhalt der Nachrichten wird im nachfolgenden Abschnitt 6.3.4 im Detail in Bezug auf die Verwaltung von Sensoren über die gegebene Raspberry Pi-Gateway-Konfiguration erläutert.

6.3.2.4 Interaktive Dokumentation mit Swagger

```
<dependency>
      <groupId>org.springdoc</groupId>
      <artifactId>springdoc-openapi-data-rest</artifactId>
      <version>1.6.8</version>
</dependency>
```

Listing 6.11: Maven Dependency zur Verwendung von Springdoc OpenAPI

Unter Verwendung von *Springdoc-Openapi* und der *Spring-Data-Rest*-Erweiterung wird eine interaktive Dokumentation zur Verfügung gestellt. Damit diese genutzt werden kann, muss diese, wie Listing 6.11 zu entnehmen, als Maven-Dependency angegeben werden. Nach Applikationsstart ist die Dokumentation über den URL-Pfad */swagger-ui/index.html* erreichbar und bietet eine Nutzeroberfläche an, unter der alle automatisch generierten Endpunkte für jede Entität angesprochen werden können. [75]

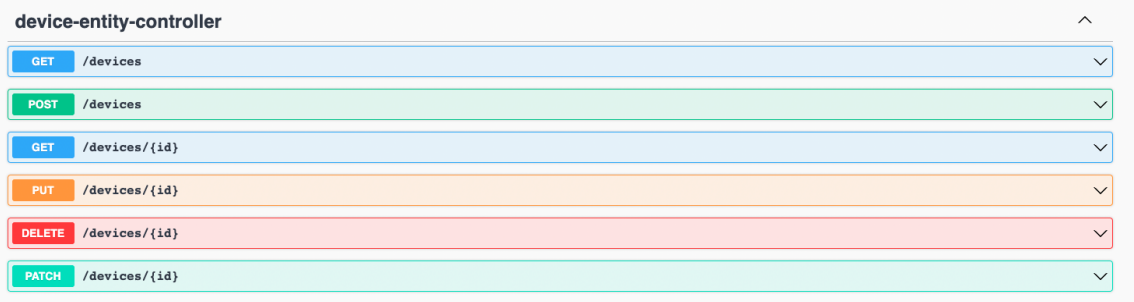


Abbildung 6.2: Swagger - Übersicht der *Device*-Endpunkte

Abbildung 6.2 zeigt exemplarisch die Übersicht der verfügbaren Endpunkte des Device-Entity-Controller. Jede verfügbare Methode wird farbig differenziert. Durch Wahl der zu testenden Option, klappt eine interaktive Nutzeroberfläche auf die die Eingabe von benötigten Parametern und die anschließende Ausführung des Requests erlaubt.

6.3.3 GrapQL - Schnittstelle

Die GraphQL-Schnittstelle zeichnet sich durch eine strenge schemabasierte Typisierung aus und erlaubt die gleichzeitige feingranulare Abfrage, aber auch die Erstellung von Daten. Ähnlich der REST-Schnittstelle wird die unterliegende Implementierung der Datenzugriffsschicht[1], genutzt, um mit der Datenbank zu interagieren. Die durchgeführte

[1] In der gegebenen Implementierung die vorliegenden JPA-Repositories.

Implementierung unterliegt jedoch der Einschränkung, dass das Datenmodell nicht vollständig abgebildet werden konnte. Aufgrund der strengen Typisierung durch das Schema kann der generische Datentyp in Java, welcher bei der Darstellung von optionalen Metadaten durch die Entität *OptMeta* verwendet wird sowie die verwendeten Datumsformate zur Repräsentation der Zeitstempel, nicht ohne weiteres abgebildet werden. Infolgedessen wurden diese in der vorliegenden Implementierung ausgelassen. Die Abfrage der Daten unter Ausschluss der genannten, ist dennoch möglich.

Zur Umsetzung der GraphQL-Schnittstelle werden, wie Listing 6.12 zu entnehmen, die Maven-Dependencies *spring-boot-starter-graphql* und *graphiql-spring-boot-starter* verwendet. Letztere stellt, ähnlich Swagger, eine interaktive Dokumentation zur Verfügung, über welche GraphQL-Operationen direkt im Browser ausgeführt werden können. Die *spring-boot-starter-graphql* Dependency stellt die Java-Implementierung der GraphQL-Spezifikation sowie weitere hilfreiche Funktionen zur Integration in Spring zur Verfügung.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-graphql</artifactId>
</dependency>

<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphiql-spring-boot-starter</artifactId>
    <version>5.0.2</version>
</dependency>
```

Listing 6.12: Maven Dependency zur Integration von GraphQL

6.3.3.1 Spezifikation des GraphQL-Schemas

Das GraphQL-Schema wird in einer Datei mit der Endung *.graphqls* angegeben. In dieser Datei werden analog zur Repräsentation der vorliegenden Entitäten mittels *POJOs*, diese als *GraphQL-Objekte* angegeben.

```
1  type Query {
2      groups: [Group]
3      group(id:ID!): Group!
4      manufacturers: [Manufacturer]
5      manufacturer(id: ID!): Manufacturer!
6      meta: [MetaData]
7      metaOne(id: ID!): MetaData!
8      devices: [Device]
9      device(id: ID!): Device!
10 }
11
12 type Mutation {
13     createGroup(input: GroupInput): Group
14     createManufacturer(input: ManufacturerInput): Manufacturer
15     createMetaData(input: MetaDataInput): MetaData
16     createDevice(input: DeviceInput): Device
17 }
```

```
18
19 type Manufacturer {
20     id: ID!
21     name: String!
22     url: String
23 }
24 type Group {
25     id: ID!
26     name: String!
27     description: String
28 }
29
30 type MetaData {
31     id: ID!
32     unit: String
33     dataType: String
34     kind: String
35 }
36
37 type Device {
38     id: ID!
39     uuid: String
40     group: Group!
41     manufacturer: Manufacturer
42     meta: MetaData
43     state: String
44     isNode: Boolean
45     associatedNode: Device
46 }
47 input GroupInput {
48     name: String!
49     description: String
50 }
51
52 input MetaDataInput {
53     unit: String
54     dataType: String
55     kind: String!
56 }
57
58 input ManufacturerInput {
59     name: String!
60     url: String
61 }
62
63 input DeviceInput {
64     uuid: String
65     group: GroupInput!
```

```
    manufacturer: ManufacturerInput
    meta: MetaDataInput
    state: String
    isNode: Boolean
    associatedNode: DeviceInput
}
```

Listing 6.13: Abbildung des Datenmodells im GraphQL-Schema

Die Typen *Query* und *Mutation* definieren hierbei alle verfügbaren Lese- und Schreiboperationen auf den entsprechenden Entitäten. Eine Besonderheit stellen die Objekte des Typs *input* dar. Diese reduzieren die Komplexität der Methodensignatur einer *Query*- oder *Mutation*-Operation dadurch, dass die Angabe jedes Attributs eines Objekt bei der Erstellung nicht als eigener Parameter definiert werden muss. Die Input-Objekte werden ebenfalls als *POJOs* umgesetzt.

6.3.3.2 GraphQL-Controller

Mittels der Annotation *@Controller* wird zu den folgenden Entitäten jeweils ein Controller umgesetzt, der als *Resolver* für eine Entität fungiert:

- **DeviceController** verarbeitet Anfragen die sich auf die Entität *Device* beziehen und implementiert die in Listing 6.13 dargestellten Lese- und Schreiboperationen.
- **GroupController** verarbeitet Anfragen die sich auf die Entität *Group* beziehen. Die Implementierung erfolgt analog zum *DeviceController*.
- **MetaDataController** verarbeitet Anfragen die sich auf die Entität *MetaData* beziehen. Die Implementierung erfolgt analog zum *DeviceController*.
- **ManufacturerController** verarbeitet Anfragen die sich auf die Entität *Manufacturer* beziehen. Die Implementierung erfolgt analog zum *DeviceController*.

Um Methoden als *Mutation* oder *Query* zu kennzeichnen werden die Annotationen *@Query* und *@Mutatation* verwendet.

6.3.3.3 Interaktive Dokumentation mit GraphiQl

Analog zur Swagger-Dokumentation der REST-API steht zur Dokumentation der GraphQL-API eine ähnliche Erweiterung zur Verfügung. Die GraphiQL-Anwendung ist über den URL-Pfad */graphiql* erreichbar. Über die Benutzeroberfläche können interaktiv *Mutation*- oder *Query*-Operationen ausgeführt werden. Weiterhin kann das unterliegende Schema eingesehen werden und die so definierten *Typen*, *Mutations* und *Queries*.

6.3.4 Netzwerkkommunikation

Als Grundlage zur Kommunikation mit dem RabbitMQ-Message-Broker dient die Java-Bibliothek Eclipse-Paho. Die Client-Implementierung wird ebenfalls als Maven-Dependency, wie in Listing 1.1 aufgeführt, eingebunden. Um eine nicht blockierende Kommunikation zur ermöglichen stellt Eclipse-Paho eine Client-Implementierung, namentlich *MqttAsyncClient*, zur Verfügung, die Aktionen wie das Empfangen und Senden von Nachrichten durch Auslagerung in separate Threads, im Hintergrund ausführt und somit die

Ausführung der Anwendung nicht blockiert. [60] Zur Bereitstellung einer im Kontext der Spring Anwendung, globale Client-Instanz wird die Initialisierung des MqttAsyncClient in einer Konfigurationsklasse ausgeführt. Dabei handelt es sich um eine mit *@Configuration* annotierte Klasse, welche somit alle in ihr definierten Methoden mittels Dependency-Injection als Singleton-Instanz bereitstellt. Die Initialisierung dieser Konfiguration erfolgt in der Klasse *MqttClientConfiguration* über die Methode *mqttClient()*. Durch Angabe des Klassenattributs *IMqttAsyncClient mqttClient;* in einer von Spring verwalteten Komponente kann direkt auf eine Client-Instanz zugegriffen werden.

```
import lombok.AllArgsConstructor;
import lombok.extern.apachecommons.CommonsLog;
import org.eclipse.paho.client.mqttv3.*;
import org.springframework.stereotype.Service;

@Service
@AllArgsConstructor
@CommonsLog
public class MessageService {

    private final IMqttAsyncClient mqttClient;

    public void publish(final String topic, final String payload) throws
        MqttException {

        MqttMessage mqttMessage = new MqttMessage(payload.getBytes());
        mqttMessage.setRetained(true);
        mqttMessage.setQos(1);
        mqttClient.publish(topic, mqttMessage);

        log.debug(String.format("Sent message on topic: %s", topic));
    }

    public void subscribe(final String topic, IMqttMessageListener listener) throws
        MqttException {
        log.debug(String.format("Initializing subscription to %s ...", topic));
        mqttClient.subscribe(topic, 1, listener);
    }

}
```

Listing 6.14: Implementierung des Message-Service

Grundsätzlich verwendet jegliche Client-Kommunikation über das MQTT-Protokoll das QoS-Level 1 und Nachrichten müssen mit dem gesetzten *retained*-Flag versendet werden, damit sichergestellt werden kann, dass diese beim Empfänger ankommen und auch nach einem Verbindungsabbruch noch verfügbar sind. Da bei jeder verschickten Nachricht das *retained*-Flag sowie das QoS-Level und bei jeder Initiierung einer Subscription das QoS-Level gesetzt werden muss, kapselt *MessageService* diese Funktionalität. Listing 6.14 zeigt die Implementierung des Service zur Broker-Kommunikation. Da es sich bei der Klasse *MessageService*, aufgrund der Annotation *@Service* um eine von Spring verwaltete Komponente handelt, ist eine Singleton-Instanz dieser Klasse im Anwendungskontext global verfügbar.

6.3.4.1 Empfang von Metadaten

Durch die Publikation von DBIRTH- und NBIRTH-Nachrichten stellen EoN-Node-Devices als auch EoN-Legacy-Devices Informationen zur Verfügung, um diese dem System bekannt zu machen. Um diese Daten zu konsumieren muss die Anwendung die notwendigen Subscriptions beim Start der Anwendung initialisieren. Für diese Funktionalität wurden die Klassen *SubscriptionConfiguration*, welche das Interface *ApplicationListener <ApplicationReadyEvent>* implementiert, umgesetzt. Das ApplicationListener-Interface erlaubt es auf Events, die durch Spring getriggert werden, zu reagieren. Das *ApplicationReadyEvent* wird getriggert sobald eine Spring-Applikation vollständig initialisiert wurde. [73] Um die Subscriptions zu initialisieren wird die *onApplicationEvent*-Methode implementiert, in der über den *MessageService* die in Tabelle 6.2 dargestellten Subscriptions gesetzt werden.

EoN-Node-Device	**EoN-Legacy-Device**
cite/+/NBIRTH/+	cite/+/DBIRTH/+/+
cite/+/NDEATH/+	cite/+/DDEATH/+/+
cite/+/NSTATE/+	cite/+/DSTATE/+/+

Tabelle 6.2: Subscription-Topics zum Empfang von Gerätedaten

6.3.4.2 Verarbeitung von Metadaten

Zur letztlichen Verarbeitung der empfangenen Daten müssen bei Initialisierung der Subscription entsprechende Message-Handler übergeben werden, welche spezifizieren wie eine Nachricht auf dem Topic, auf dem die Subscriptions gesetzt wurde, verarbeitet wird. Der Paho-Client stellt zur Implementierung das Interface *IMqttMessageListener* zur Verfügung. Eine Implementierung dieses Interface muss lediglich die Methode *messageArrived* implementieren, in welcher spezifiziert wird, wie mit einer Nachricht umzugehen ist. Respektive werden die Implementierungen *BirthMessageHandler*, *DeathMessageHandler* und *StateMessageHandler* umgesetzt, um eingehende Nachrichten zu prozessieren.

Birth Message Handler Die Aufgabe des *BirthMessageHandler* ist es eingehende Birth-Zertifikate zu verarbeiten und die relevanten Daten im Datenhaltungssystem zu persistieren. Dazu erfolgt zunächst per Dependency Injection die Angabe der relevanten Repositories für den Persistierungsschritt. Gemäß der in Tabelle 5.4 dargestellten Metriken wird geprüft, ob diese vorhanden sind. Entsprechend der vorliegenden Daten werden die folgenden Entitäten konstruiert:

- **Manufacturer** Diese Entität wird mit den Metadaten *manufacturerUrl* und *manufacturerName* instanziiert. Sind die Metriken *MetaData/manufacturerUrl* und *MetaData/manufacturerName* in der Nachricht spezifiziert, wird zunächst geprüft ob ein passender Datenbankeintrag existiert und dementsprechend eine Referenz geladen, andernfalls wird die Entität erstellt und persistiert.
- **Group** Die Entität *Group* wird mit der *group_id* instanziiert, welche implizit durch den Topicnamen gegeben ist und von diesem abgeleitet wird. Bevor die Entität persistiert wird, muss geprüft werden ob ein Eintrag mit dem Attribute *name* bereits

existiert. Existiert bereits ein Eintrag wird die Referenz geladen, andernfalls wird die Entität persistiert.

- **MetaData** Da die Entität *MetaData* in einer *OnToOne-Beziehung* zur Entität *Device* steht, wird diese in jedem Fall persistiert. Alle Metriken deren Name mit *MetaData* beginnt, werden auf der Entität gesetzt. Befinden sich in der Nachricht Metriken deren Name mit *OptMeta* beginnt, werden diese als Element des Typs *OptMeta* dem Attribut *opt_info* hinzugefügt.

- **D**ie Repräsentation eines EoN-Node-Devices, sowie eines *EoN-Legacy-Devices* hält Referenzen auf die zuvor aufgeführten Entitäten. Lediglich die Attribute *state* und *isNode* müssen unabhängig vom Nachrichteninhalt gesetzt werden. Da der Erhalt der Nachricht die Verfügbarkeit eines Gerätes im Netzwerk signalisiert, wird das Attribut *state* auf den Wert 'ONLINE' gesetzt. Der Wert des Attributs *isNode* lässt sich analog zur *group_id* vom Topic-Namen ableiten. Anhand des Nachrichtentyps (NBIRTH, DBIRTH) wird bestimmt ob es sich um ein *EoN-Node-Device* oder ein *EoN-Legacy-Device* handelt. Handelt es sich um ein *EoN-Node-Device* wird das Attribut *isNode* auf den Wert *true* gesetzt, andernfalls auf *false*. Im Falle eines *EoN-Legacy-Device* muss das zugehörige *EoN-Node-Device* über das Attribut *associatedNode* referenziert werden, welches wiederum über den Topic-Namen bestimmt wird.

Wurden alle Entitäten erstellt, werden diese über das jeweilige Repository mittels der *save*-Methode persistiert.

Death Message Handler Death-Zertifikate signalisieren, dass eine Gerät nicht mehr verfügbar ist. Entsprechend wird das Attribut *state* eines *Device* bei erhalt eines Death-Zertifikats auf den Status *'OFFLINE'* gesetzt. Hierfür wird lediglich das Repository der Entität *Device* integriert um diese Änderung zu persistieren.

State Message Handler Eingehende State-Nachrichten werden von der Klasse *StateMessageHandler* prozessiert. Analog zum Handling der Birth- und Death-Zertifikate werden benötigte Repositories zur Persistierung angegeben. In diesem Fall wird lediglich das zur Entität *Device* zugehörige Repository benötigt. Beim Empfang einer Nachricht wird der Name der Metriken auf den Wert *Info/state* überprüft. Der enthaltene Wert entspricht entweder dem Wert *'OFFLINE'* oder *'ERROR'*. Gemäß diesem Wert wird das Attribut *state* der Device-Entität gesetzt. Diese kann wiederum über den Topic-Namen und den darin enthaltenen Nachrichtentyp *(NSTATE, DSTATE)* identifiziert werden.

6.3.4.3 Remote Konfiguration des Raspberry-Pi

Die remote gesteuerte Konfiguration des Raspberry Pi umfasst in dieser Implementierung drei Funktionen. Die *Registrierung* von bereits physisch angeschlossenen Sensoren und Aktoren, das *Löschen* von angeschlossenen Sensoren und Aktoren und die nachträgliche *Konfiguration* von angeschlossenen und bereits im System bereitgestellten Geräten. Alle bereitgestellten Endpunkte werden mittels eines *POST-Request* angesprochen und enthalten alle benötigten Informationen, die die Gegenseite, wie in Abschnitt 5.4.2 beschrieben, zur Verarbeitung benötigt.

Registrierung von Peripheriegeräten - /devices/register Über den Endpunkt */devices/register* können über ein Raspberry Pi-Gateway angeschlossene *EoN-Legacy-Devices* remote initialisiert werden und somit im Netzwerk bereitgestellt werden. Die Registrierung von Grove-Sensoren setzt voraus, dass diese bereits physisch am jeweiligen Port angeschlossen wurden. Eine *Registrierungsnachricht* an den Raspberry-Pi, wird von diesem verarbeitet und löst die Initialisierung einer weiteren *DeviceManager*-Instanz auf Gateway-Seite aus. Eine solche Nachricht erfordert die Angabe der erforderlichen Informationen als Key-Value-Paare in der vorgestellten Payload-Struktur wie in Listing 5.2 bereits dargestellt. Das Senden einer Registrierungsnachricht erfordert keine weiteren manuellen Bearbeitungsschritte in Form von Anfragen an die REST-Schnittstelle. Ein so registriertes Gerät sendet bei erfolgreicher Initialisierung ein Birth-Zertifikat, welches wie in Abschnitt 5.4.2 beschrieben verarbeitet wird. Hierbei ist anzumerken, dass dieser Endpunkt nur für*EoN-Node-Devices* die als Gateways fungieren verwendet werden kann.

Löschen von Peripheriegeräten - /devices/delete Analog zur Registrierung von Geräten, können Geräte über diesen Endpunkt auch aus dem Netzwerk entfernt werden. Das Senden einer Nachricht zur Löschung eines Geräts, wird vom empfangenden *DeviceManager* oder *GatewayManager* verarbeitet. Dementsprechend wird die Metrik mit der Kennung *Config/delete* und der UUID des zu entfernenden Gerätes übermittelt. Die Verarbeitung der Nachricht erfolgt auf Seite des *EoN-Devices* respektive durch die Instanz des *DeviceManager* oder *GatewayManager*.

Konfiguration von Peripheriegeräten - /devices/config Der Endpunkt */devices/config* kapselt das Senden einer MQTT-Nachricht um konfigurationsrelevante Daten an ein *EoN-Node oder -Legacy-Device* zu übermitteln. In der gegebenen Implementierung steht lediglich die Funktionalität zur Verfügung die *group_id* eines Geräts zu ändern. Wie bereits in Absatz 5.4.2.2 beschrieben, wird eine Metrik mit der Kennung *Config/changeGroup* und der gewünschten Bezeichnung der *group_id* als Wert übermittelt.

6.3.5 Konfiguration

Die Konfiguration eines Spring Microservices kann auf verschiedene Weise erfolgen. Programmbibliotheken die nativ durch entsprechende Abhängigkeiten in eine Spring-Anwendung integrierbar sind, erlauben eine einfache Konfiguration durch Spring Application Properties. Diese werden auf Root-Ebene der Anwendung im Ordner resources in einer separaten Datei definiert. Alle Konfigurationsparameter werden im Yaml-Format in einer application.yaml Datei angegeben, da dieses Format gerade bei einer umfangreicheren Konfiguration durch seine hierarchischen Aufbau eine bessere Übersicht bietet.

Die Konfiguration der *Spring-Data-Jpa*, der *Spring-Data-REST*-Erweiterung und des Spring-Servers erfolgt ausschließlich über die dafür vorgesehenen Schlüsselwerte. Das umfasst unter anderem den Connection-String der Datenbank, der verwendete Datenbanktreiber, Username und Passwort sowie den Server-Port unter dem der Server letztlich ansprechbar ist. Spring erfasst diese Konfiguration automatisch und übernimmt bei Applikationsstart die so spezifizierte Konfiguration. Bibliotheken, welche nicht nativ in eine Spring-Anwendung integrierbar sind, können benötigte Konfigurationsparameter ebenfalls als Application Properties unter selbst definierten Schlüsselwerten angeben, benötigen aber weitere Schritte um diese in der Anwendung zu nutzen. Hierzu werden

POJOs mittels der Annotation *@Configuration* markiert um sie im Spring Kontext verfügbar zu machen. Weiterhin können Klassenattribute mit der *@Value*-Annotation, den in der Application Properties Datei definierten Schlüsselwerten zugeordnet werden. Zur Konfiguration des *Eclipse-Paho*-Client implementiert der Microservice die Klasse *MqttClientConfiguration*. Die in der Implementierung genutzte Konfiguration kann der Darstellung in Anhang A.1 entnommen werden. Sensible Daten wurden entsprechend ausgelassen.

Im Rahmen des Cluster-Deployments der Anwendung ist eine externalisierte Konfiguration wünschenswert. Damit nicht jede einzelne Replik der Anwendung manuell konfiguriert werden muss, werden sensible Daten wie Datenbank- und Broker-Verbindungsdaten in Kubernetes-*Secrets* ausgelagert. Die allgemeine Konfiguration der Spring-Anwendung analog in eine *ConfigMap*. Alle Zertifikatsdateien, die für die SSL-Verbindung des Brokers und der Datenbank benötigt werden, werden als Dateien unter dem Pfad */etc/broker* und */etc/db* gemounted. Wohingegen alle weiteren Konfigurationsobjekte, alle Schlüssel-Wert-Paare als Umgebungsvariablen im Pod der Anwendung zur Verfügung stellen. Hierbei wird die durch Spring Boot zur Verfügung gestellte Funktionalität ausgenutzt, jegliche in application.yaml spezifizierten Parameter durch gleichnamige Umgebungsvariablen zu überschreiben. Zu beachten ist jedoch, dass bei Trennzeichen oder über eine Hierarchieebene hinaus das in Tabelle 6.3 dargestellte Benennungsschema zu befolgen ist.

Property	**Umgebungsvariable**
foo	FOO
foo.bar	FOO_BAR
foo.barBaz	FOO_BAR_BAZ
foo.bar-baz	FOO_BAR-BAZ

Tabelle 6.3: Überschreiben der Properties durch Umgebungsvariablen

Ebenso erfordert die Bereitstellung von Metriken keinen zusätzlichen Implementierungsaufwand, sondern lediglich wenig Konfigurationsaufwand, aufgrund verfügbarer Spring-Integrationen. Lediglich die Einbindung der in Listing 6.15 dargestellten Maven Dependencies ermöglichen die Nutzung von Spring Actuator. Verschiedene Metriken können anschließend über den bereitgestellten Endpunkt */actuator/prometheus* von Anwendungen wie *Prometheus* abgerufen und durch weitere Tools wie *Grafana* visualisiert werden.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Listing 6.15: Einbinden von Spring Actuator und Micrometer

6.3.6 Deployment

Um ein Kubernetes-Deployment der Anwendung zu erstellen, muss im ersten Schritt eine *Java Archive (JAR)-Datei* erstellt werden. Mithilfe dieser Datei wird im Anschluss eine containerisierte Anwendung bereitgestellt. Die Containerisierung erfolgt anhand von, in einem *Dockerfile*, spezifizierten Anweisungen. Nach erfolgreichem Erzeugen eines Container-Image wird dieses über *Docker-Hub* bereitgestellt. Bei der Bereitstellung über *Docker-Hub* wird einem Image ein eindeutiges *Tag* zugewiesen. Dadurch kann das Docker-Image über das global verfügbare Docker-Hub-Repository abgerufen werden. Kubernetes nutzt dieses Image um ein Deployment zu erstellen. Das Deployment kapselt pro Pod eine Anwendungsinstanz des Microservice. Dabei wird ein *ReplicaSet* verwendet um sicherzustellen, dass zu jeder Zeit drei Instanzen des Microservice bereitstehen. Weiterhin wird für die interne Kommunikation ein Service des Typs *ClusterIP* erstellt. So sind andere Microservices in der Lage innerhalb des Clusters mit der Anwendung über die bereitgestellten Schnittstellen zu kommunizieren.

Da für die spätere Entwicklung von möglichen Frontendanwendungen, der Zugriff von außerhalb des Clusters erforderlich sein kann, wird zusätzlich ein Service des Typs NodePort erstellt. Dieser ermöglicht den Zugriff auf einem statischen Port, einer jeden Node im Cluster und wird mithilfe der *expose*-Funktionalität des Tools *kubectl* erstellt. Der NodePort-Service ist außerhalb des Clusters über <*NodeIP*>:<*NodePort*> erreichbar. [29]

6.4 Monitoring & Logging

Nach dem in Abschnitt 5.2.3 vorgestellten Konzept soll eine Monitoring-Anwendung zur Überwachung des Microservice-Deployments, sowie eine Logging-Anwendung zur Aggregation aller Log-Nachrichten auf dem Kubernetes-Cluster bereitgestellt werden. Zur Umsetzung dieser Anwendungen werden Prometheus zur *Aggregation von Metriken* und Grafana-Loki zur *Aggregation von Log-Nachrichten* verwendet. Beide Tools ermöglichen eine einfache Bereitstellung in einer Clusterumgebung und können auf einfache Weise skaliert werden um auch größere Datenmengen zu verarbeiten.

Metriken über die Ressourcenauslastung der JVM und weitere Spring-spezifische Metriken werden über den durch die *Spring-Actuator*-Erweiterung bereitgestellten Endpunkt zur Verfügung gestellt. Diese Metriken können über eine entsprechende Konfiguration von Prometheus abgerufen werden.

Analog zur Aggregation von Metriken, aggregiert Loki alle auf den Standardkanälen *stdout* und *stderr* produzierten Log-Nachrichten. Im Gegensatz zu Prometheus ist der Konfigurationsaufwand für ein Loki-Deployment sehr gering, da dies durch ein einfaches Helm-Chart ohne viele Konfigurationsparameter bereitgestellt werden kann.

Um die so aggregierten Daten in einer strukturierten Ansicht bereitzustellen wird *Grafana* zur Visualisierung der Daten verwendet. Grafana bietet eine grafische Benutzeroberfläche mit der Dashboards erstellt werden können, womit sich die Metriken und Log-Nachrichten visualisieren lassen. Damit die durch Prometheus und Loki aggregierten Daten in Grafana einsehbar sind, müssen beide Anwendungen als Datenquellen in Grafana verfügbar gemacht werden. Die Bereitstellung als Datenquelle erfolgt im Rahmen des Deployments und der anschließenden Konfiguration.

6.4.1 Deployment

Das Deployment des gesamten Monitoring-Stacks erfolgt über die Verwendung von Helm-Charts. Das von Grafana-Labs bereitgestellte Chart *loki-stack* ermöglicht ein einfaches Deployment einer Grafana-Instanz mit jeweils einem vorkonfigurierten Deployment von Loki und Prometheus, welche bereits als Datenquelle verfügbar sind. Um jedoch auch die Metriken der Spring-Anwendung verfügbar zu machen erfordert Prometheus eine erweiterte Konfiguration die Informationen zur Verfügung stellt, an welcher Stelle die respektiven Metriken zu finden sind. Diese Konfiguration der Prometheus-Instanz ist über sogenannte *ScrapeConfigs* möglich. Eine *ScrapeConfig* wird in der Regel als zusätzliche Konfiguration in der Deployment-Konfiguration angegeben. Je nach Anforderung können solche Konfigurationen sehr umfangreich und komplex werden. Um diese Komplexität zu reduzieren wird sowohl die Prometheus-, sowie die Grafana-Instanz nicht über das *loki-stack* Helm-Chart, sondern über ein Deployment des *Prometheus-Operator* über ein entsprechendes Chart realisiert. Das Operator-Deployment inkludiert ein Prometheus-Deployment, sowie ein Grafana-Deployment, wobei Prometheus also Datenquelle zur Verwendung in Grafana vorkonfiguriert ist.

Ziel des *Prometheus-Operator* ist die Simplifizierung der Deployment-, und Target-Konfiguration. Letztere bezieht sich auf die Konfiguration der zu aggregierenden Anwendungsmetriken, welche normalerweise über *Scrape-Konfigurationen* definiert werden. Der *Prometheus-Operator* stellt über eine *CustomResourceDefinition*, eigene Kubernetes-Objekte zur Verfügung, über welche die Ziele zum Abruf von Metriken spezifiziert werden können. In diesem Fall wird ein *ServiceMonitor*-Objekt definiert. Dieses wird analog zu anderen Kubernetes-Objekten im YAML-Format angegeben und mittels *kubectl* bereitgestellt. Ein *ServiceMonitor* bezieht sich auf einen konkretes *Service*-Objekt eines *Deployments*. Damit Prometheus in der Lage ist über den entsprechenden *Service* die Metriken aller zugehörigen *Pods* abzurufen, muss ein *Service* ein Label für den Port besitzen auf dem die Anwendung läuft. In der Konfiguration des *ServiceMonitors* wird dieses Label sowie der relative URL-Pfad */actuator/prometheus* angegeben unter dem die Metriken der Spring-Anwendung abrufbar sind. Durch einen Abgleich der Labels ist Prometheus so in der Lage alle Anwendungen mit dem angegeben Label als *Target* zu definieren. [64]

Analog zum Deployment des Prometheus-Operators erfolgt das Loki-Deployment über das bereits genannte *loki-stack* Helm-Chart. Anstatt des gesamten Stacks wird jedoch lediglich das Loki-Deployment ausgeführt. Die erneute Installation von Prometheus und Grafana wird durch entsprechende Parameter bei der Ausführung des Helm-Chart unterbunden. Da die so bereitgestellte Loki-Instanz nicht zur Verwendung mit Grafana vorkonfiguriert wurde, muss diese noch als Datenquelle hinzugefügt werden. Über die grafische Benutzeroberfläche von Grafana, kann Loki mit wenigen Klicks als Datenquelle hinzugefügt werden. Die Konfiguration erfordert lediglich die Angabe der URL *http://loki:3100* unter welcher Loki innerhalb des Clusters erreichbar ist.

Die Installation der Helm Charts kann in Anhang A.4 und die Prometheus konfiguration über das *ServiceMonitor*-Objekt in Anhang A.3.3, nachvollzogen werden.

7 Evaluation

Zur Evaluation der Eignung der vorgestellten Lösung, wird im Folgenden die Spezifikation der Anforderungen mit der tatsächlichen Umsetzung des entwickelten Konzepts verglichen. Durch dieses Vorgehen wird ein Fertigstellungsgrad bestimmt. Anschließend werden die verwendeten Technologien anhand qualitativer Merkmale in Bezug auf die gegebenen Anforderungen bewertet. Weiterhin erfolgt eine Evaluation inwieweit die vorgestellte Kommunikationsstrukur kompatibel mit bereits durchgeführten Arbeiten im Rahmen des CiTe-Testfeldes kompatibel ist und wie eine Anpasssung umsetzbar wäre.

7.1 Erfüllung der Anforderungen

Eine Berechnung des Fertigstellungsgrades erfolgt nach der 0/50/100-Methode, dabei werden angefangene, aber nicht abgeschlossene, mit 50%, nicht angefangene Arbeitspaket mit einem Fertigstellungsgrad von 0% bewertet. Jedes abgeschlossene Arbeitspaket fließt mit einem Fertigstellungsgrad von 100% in die Berechnung ein. Der letztliche Fertigstellungsgrad wurde anhand der Einteilung der Arbeitspakete in die Folgenden, wie bereits in Abschnitt 4.2 dargestellten, übergeordneten Kategorien unter Berücksichtigung der funktionalen und nicht-funktionalen Anforderungen gemessen:

- **Broker** Alle die Kommunikation mit dem Message-Broker betreffenden Arbeitspakete wurden unter dieser Kategorie aggregiert. Darunter die Spezifikation der Payload-Struktur, der Nachrichtentypen und der Topic-Struktur.
- **Datenhaltungssystem** Die Arbeitspakete bezüglich des Datenhaltungsystems enthalten die Bereitstellung und Konfiguration der CockroachDB-Datenbank im Cluster, sowie die Erstellung des Datenbankschemas.
- **Microservice** Die Implementierung der Datenzugriffsschicht, der universellen Schnittstelle, der MQTT-Client-Implementierung, des Datenmodells sowie die gesamte Konfiguration als auch die externalisierte Konfiguration, wurde unter der Kategorie Microservice zusammengefasst.
- **Raspberry-Pi** Das Arbeitspaket *Raspberry-Pi* fasst die Anbindung der Sensoren, die Konfiguration der Anwendung sowie die Implementierung des Message-Broker-Clients zusammen.

Durch Anwendung der 0/50/100-Methode ergibt sich ein kumulierter Fertigstellungsgrad von 94%.

7.2 Message Broker - Kommunikationsstandard

Anforderung	Umsetzung	Erfüllt
FR.BK.001	Entwurf eines einheitlichen Topic-Schemas	Erfüllt
FR.BK.002	Entwurf eines einheitlichen Topic-Schemas	Erfüllt
FR.BK.003	Keine eigene Implementierung notwendig.	Erfüllt
FR.BK.004	Aggregation durch Loki.	Erfüllt
FR.BK.005	Keine eigene Implementierung notwendig.	Erfüllt
NFR.BK.001-007	Keine eigene Implementierung notwendig.	Erfüllt

Tabelle 7.1: Message-Broker - Erfüllung der Anforderungen

Tabelle 7.1 listet die zuvor spezifizierten Anforderungen an den Message-Broker auf und deren Fertigstellungsgrad.

7.2.1 Standardisierte Kommunikation

Die Spezifikation eines universell verwendbaren Topic-Namespaces stellt einen wesentlichen Bestandteil der Anwendung dar. Durch die Verwendung des Konzepts von Birth- und Death-Zertifikaten sowie der Übermittlung von Statusnachrichten, konnte die automatische Bereitstellung, Abmeldung und Benachrichtigung im Fehlerfall umgesetzt werden. Es wurden festgelegte Topic zur Kommunikation von erfassten Daten *(NDATA, DDATA)*, zum Empfang von Steuerungsbefehlen *(NCMD,DCMD)* und Konfigurationsnachrichten *(NCONFIG, DCONFIG)* entworfen. Diese Struktur kann für zukünftige Projekte genutzt werden, um eine einheitliche projektübergreifende Kommunikation zu ermöglichen. Wodurch eine einfachere Einbindung von verwandten Arbeiten möglich ist, da diese auf der selben Kommunikationsbasis beruhen. Bestehende Arbeiten, wie in Abschnitt 3.1.2 und Abschnitt 3.1.3 beschrieben, müssen dennoch angepasst werden.

Die Arbeit von Studer verwendet einfache Topics zum Senden von erfassten Sensormetriken. Diese müssen zur Gewährleistung der Kompatibilität angepasst werden, so dass Grove-Sensoren ihre Daten auf entsprechenden Topics mit Nachrichtentyp *DDATA* senden. In Studers Implementierung werden erfasste Daten über das MQTT-Protokoll versendet und über AMQP-Queues weitergeleitet. Microservices greifen diese Nachrichten ab und prozessieren diese weiter. Die Weiterleitung wird durch einen *Routing-Key* spezifiziert, welche im Subscription-Client des Service angepasst werden muss, so dass der *DDATA-Topic* verwendet werden kann.

Altmeyers Implementierung einer Roboter-Steuerung verwendet im Gegensatz zu Studers Implementierung weitere Topics, die die Brücke zwischen Sensorhardware und Backendanwendung bilden. Im Wesentlichen werden drei Topics zur Bereitstellung, zum Empfang von Steuerungsbefehlen und zum Senden von Sensordaten verwendet. Diese können respektive angepasst werden, indem Birth-Zertifikate zur Bereitstellung, DCMD- oder NCMD-Nachrichten zum Empfang von Steuerungsbefehlen und DDATA- oder NDATA-Nachrichten zum Senden von Sensordaten verwendet werden.

7.2.2 Cluster-Deployment

Das RabbitMQ Cluster-Deployment wurde als bereits bestehendes Deployment nicht im Rahmen dieser Arbeit umgesetzt. Die gegebenen Anforderungen wurden jedoch analog als Voraussetzung von Cluster-Anwendungen festgelegt. Das Deployment stellt grundsätzliches Logging zur Verfügung, welches über das Loki-Deployment eingesehen und analysiert werden kann. RabbitMQ stellt über ein entsprechendes Plugin einen Endpunkt zur Verfügung der von Prometheus angesprochen werden kann.

Die in den nicht-funktionalen Anforderungen spezifizierten Punkte beziehen sich auf den Deployment-Prozess und die Performance des RabbitMQ-Deployments. Da dieses umfangreich durch das Team des DSL getestet wurde, werden diese Anforderungen als erfüllt betrachtet.

7.3 Datenhaltungssystem

Anforderung	Umsetzung	Erfüllt
FR.DB.001	Abfrage über PostgresSQL-Syntax.	Erfüllt
FR.DB.002	Anwendung betreibt Logging.	Erfüllt
FR.DB.003	Aggregation durch Loki.	Erfüllt
FR.DB.004	Stellt Metriken über Endpunkt bereit.	Erfüllt
NFR.DB.001/002	Operator oder Helm-Chart.	Erfüllt
NFR.DB.003	Vorliegende Benchmarks von CockroachLabs.	Erfüllt
NFR.DB.004/005	Bedingt durch CockroachDB-Architektur.	Erfüllt
NFR.DB.006	Skalierung über Deployment Konfiguration.	Erfüllt
NFR.DB.007	Bedingt durch CockroachDB-Architektur.	Erfüllt

Tabelle 7.2: Datenhaltungssystem - Erfüllung der Anforderungen

Das vorliegende Deployment der CockroachDB-Datenbank wurde auf dem DSL-Cluster durchgeführt. Dabei konnte mit 2 CPUs und 8Gi Arbeitsspeicher als allokierten Ressourcen per Container, bereits ein zufriedenstellendes Ergebnis erzielt werden. Tabelle 7.2 listet alle spezifizierten Anforderungen und Status in Bezug auf deren Erfüllung auf.

7.3.1 Flexibilität

Unter Verwendung der klassischen SQL-Syntax bietet CockroachDB leichten Zugang und eine einfache, aber umfassende Möglichkeit um Abfragen zu erstellen. CockroachDB orientiert sich eng an der Datenbanklösung Postgres und der adaptiert das Postgres Netzwerkprotkoll *pgwire*, sowie den Postgres eigenen *Syntax Parsers*, wodurch eine große Bandbreite der SQL-Syntax und SQL-Funktionen von PostgreSQL nutzbar sind. Wie in der Umsetzung verwendet, können so SQL-untypische Datentypen wie JSON verwendet werden. Dadurch werden Ansätze eines dokumentenbasierten Speichers eingeführt, was einen

flexiblen Umgang mit unstrukturierten Daten in einem relationalen Datenbanksystem erlaubt. [52]

Weiterhin stellt CockroachDB im Hinblick auf zukünftige Anwendungszenarien im gegebenen Kontext eine gute Lösung dar, da die Persistierung von geographischen Daten in speziell dafür entwickelten Datenstrukturen möglich ist. Hierbei orientiert sich die Umsetzung an der populären PostgreSQL-Erweiterung Post-GIS, welche speziell für den Umgang mit geographischen Daten entwickelt wurde. [82] Bisher unterliegt diese Implementierung jedoch einigen Limitationen. So ist beispielsweise nur ein Subset der von Post-GIS zur Verfügung gestellten Funktionen verfügbar. [54]

7.3.2 Konsistenz

Wie bereits in Abschnitt 2.2.5 beschrieben gewährleistet CockroachDB eine Garantie für eine starke Konsistenz. Um diese Garantie zu untermauern greift CockroachLabs in der Entwicklung auf das Testing-Framework *Jepsen* zurück. Jepsen ist eine Open-Source-Softwarebibliothek zum Testen von verteilten Datenbanken und dient der Verbesserung der Sicherheit von verteilten Datenbanken, Queues und Konsenssystemen. [53] Mithilfe der Softwarebibliothek und der Unterstützung des Jepsen-Entwicklers, bindet CockroachDB diese Tests in die Entwicklung ein und kann so prüfen, ob die ausgesprochenen Garantien eingehalten werden können. Durch nachweislich gute Ergebnisse in den so erstellten Tests, eignet sich CockroachDB für das vorliegende Anwendungsszenario, um eine konsistente Sicht auf gespeicherte Daten zu gewährleisten. [55][6]

7.3.3 Performance

Die Performance der Datenbank wurde in dieser Arbeit lediglich in Kombination mit der universellen Schnittstelle getestet. Ein unabhängiges Testen konnte im zeitlichen Rahmen nicht mehr umgesetzt werden. Für den Test vorgesehen war das TPC-C-Benchmark. Dabei handelt es sich um ein OLTP-Benchmark, welches mithilfe eines vorgefertigten Datensatzes eine Vielzahl an Operationen ausführt um so Latenz und Durchsatz zu messen. [62]

Eine allgemeine Übersicht unter Verwendung verschiedener Benchmarks stellt CockroachLabs selbst im Rahmen der Dokumentation zur Verfügung. Abbildung 7.1 zeigt das Ergebnis eines Benchmarks zur Beobachtung der Latenz von Lese- und Schreiboperationen bei einer linearen Skalierung von 0 bis hin zu 256 Knoten. Dabei ist zu beobachten, dass der Durchsatz linear zur Knotenanzahl steigt und die Latenz der Lese- und Schreiboperationen nahezu konstant bei einer Latenz von deutlich unter 100ms liegt. Ausgeführt wurden diese Tests auf *AWS c5d.4xlarge* Instanzen. [63]

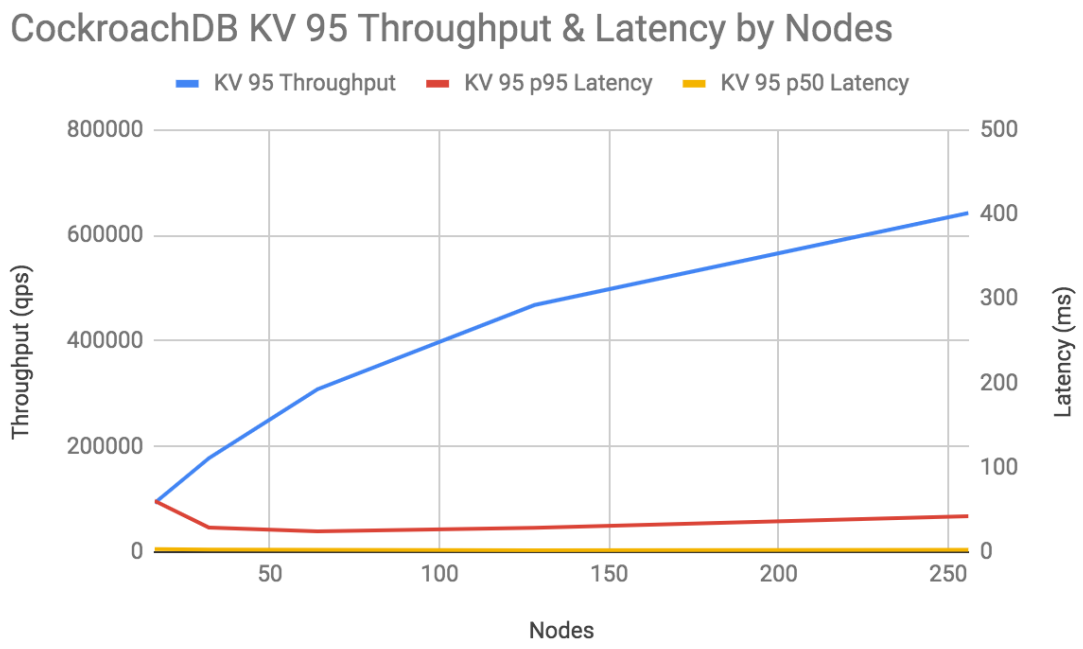


Abbildung 7.1: Lineare Skalierung - Durchsatz und Latenz bei steigender Node-Anzahl [63]

Um die durchschnittliche Latenz zu testen, steht ein Sysbench-Benchmark zu Verfügung, welches auf einem Cluster mit drei Nodes auf einer *AWS c5d.9xlarge*-Instanz in der AWS-Region *us-east-1* über die *Availability-Zonen* a,b und c ausgeführt wurde. In Abbildung 7.2 ist das Ergebnis des Benchmarks dargestellt. Die Operationen *oltp_insert* und *oltp_select* repräsentieren jeweils ein Set von Schreib- und Leseoperationen die durch das Sysbench-Benchmark ausgeführt wurden. Zu beobachten ist, dass auch hier die Latenz sich in einem sehr geringen Spektrum von unter 5ms bewegt. Anhand der vorliegenden Benchmarks stellt Cockroach in Bezug auf geringe Antwortzeiten und Skalierung eine geeignete Lösung dar. Weiterhin wäre theoretisch eine beliebige weitere Skalierung eines CockroachDB-Clusters möglich.

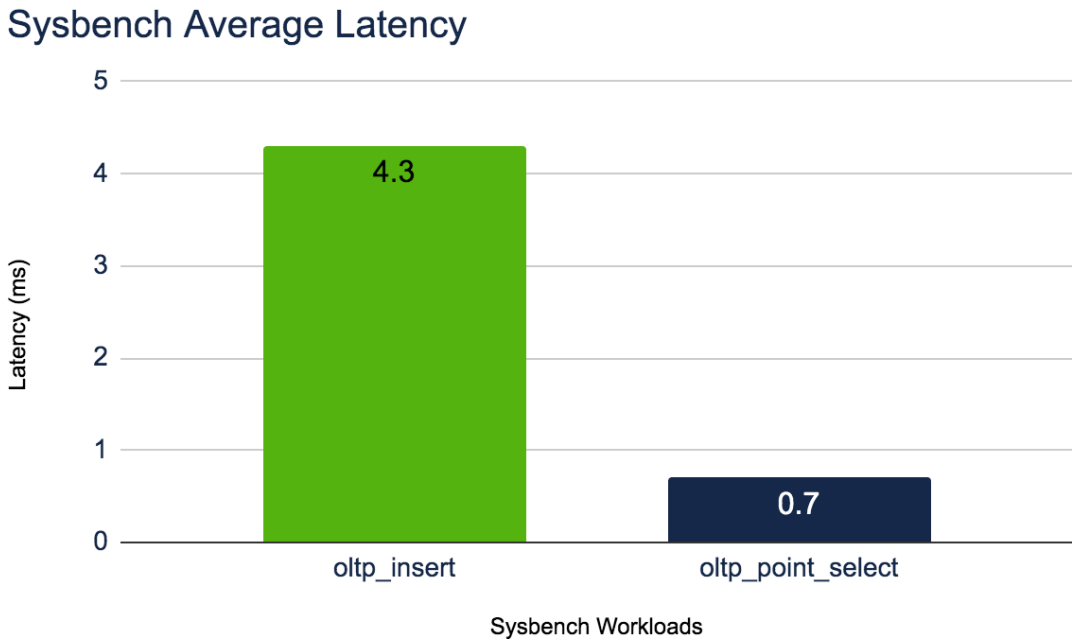


Abbildung 7.2: Durchschnittliche Latenz anhand des Sysbench-Benchmark [63]

7.3.4 Cluster-Deployment

Das Deployment von Cockroach auf Kubernetes wurde mittels des Cockroach-Operator durchgeführt. Dieser übernimmt einen großen Teil der Konfiguration, erlaubt aber dennoch Anpassungen, falls notwendig. Alternativ ist ebenfalls ein Deployment über Helm-Charts oder eine *StatefulSet*-Konfiguration möglich. Anzumerken ist jedoch, dass das bereitgestellte Helm-Chart offiziell lediglich Kubernetes Version 1.22 oder früher unterstützt. Für weitere Versionen wird das Helm-Chart nicht mehr weiterentwickelt. [8] Durch diese Möglichkeiten ist ein unkompliziertes Deployment auf einem Cluster möglich. Die horizontale Skalierung erfolgt manuell über Anpassung der Deployment-Konfiguration und ist auf beliebig viele Nodes erweiterbar, wodurch eine hohe Verfügbarkeit und Resilienz der Anwendung gegeben ist. Weiterhin stellt jede Cockroach Instanz Endpunkte zur Kommunikation von Metriken zur Verfügung, so dass diese von Anwendungen wie Prometheus verarbeitet werden können. Diese können ebenfalls über ein von Cockroach bereitgestelltes Dashboard eingesehen werden, welches über den Service *cockroachdb-public* erreichbar ist. [59]

7.4 Spring Microservice

Anforderung	Umsetzung	Erfüllt
FR.MS.001	REST-Schnittstelle wurde umgesetzt. GraphQL-Schnittstelle, konnte nur zum Teil umgesetzt werden.	Teilweise
FR.MS.002/003	Registrierung/Abmeldung von Sensoren über HTTP-Endpunkt.	Erfüllt
FR.MS.003	Topic-Schema wird verwendet. Eclipse Paho zur Kommunkation.	Erfüllt
FR.DB.004	Umsetzung durch Spring Data JPA.	Erfüllt
FR.MS.005	MessageHandler prozessieren MQTT-Nachrichten.	Erfüllt
FR.MS.006	Implementierung von Entities.	Erfüllt
FR.MS.007	Leader Election wurde nicht umgesetzt.	Nicht erfüllt
FR.MS.008	Zugriffskontrolle wurde nicht umgesetzt.	Nicht erfüllt
FR.MS.009	Logging via Apache Commons Logging.	Erfüllt
FR.MS.010	Metriken werden über Actuator Endpunkt bereitgestellt.	Erfüllt
FR.MS.011	Aggregation durch Loki.	Erfüllt
NFR.MS.001	Konfiguration über ConfigMaps und Secrets.	Erfüllt
NFR.MS.002	Modularer Aufbau mittels Repositories.	Erfüllt
NFR.MS.003/004	Bereitstellung über Deployment-Konfiguration	Erfüllt
NFR.MS.005/006	Microservices sind stateless.	Erfüllt
NFR.MS.007	Skalierbar über Deployment Konfiguartion.	Erfüllt

Tabelle 7.3: Datenhaltungssystem - Erfüllung der Anforderungen

Die Anwendung des Microservices zur Verwaltung der Metadaten wurde unter Verwendung des Spring-Frameworks und mehreren Erweiterungen dessen umgesetzt. Eine Zusammenfassung des Status der Anforderungserfüllung kann Tabelle 7.3 entnommen werden.

7.4.1 Datenzugriff

Mit *Spring Data JPA* konnte ein modularer Aufbau der Datenzugriffschicht umgesetzt werden. Dabei wurde das Datenmodell durch Implementierung von *Entity*-Objekten umgesetzt. Die Implementierung kann auf einfache Weise durch Hinzufügen weiterer Entitäten und Repositories erweitert werden. Das unterliegende Framework Hibernate, ermöglicht weiterhin auch die Verwendung verschiedener Datenquellen und unterstützt selbst diverse Erweiterungen. Für zukünftige Anwendungszenarien in denen beispielsweise geographische Daten verarbeitet werden müsssen, kann die Hibernate-Erweiterung *Hibernate Spatial* verwendet werden, welche notwendige Datentypen zu Verfügung stellt. Dabei ist *Hibernate Spatial* ebenfalls kompatibel mit der in dieser Arbeit verwendeten Datenbanklösung *CockroachDB*. [47]

7.4.2 Broker-Kommunikation

Die zum Empfang von Gerätedaten notwendige MQTT-Kommunikation wurde durch die Integration des Eclipse-Paho-Client umgesetzt. Dieser repräsentiert quasi den De-Facto-Standard zur Kommunikation über das MQTT-Protokoll und bietet durch die ständige Weiterentwicklung eine zuverlässige Lösung. Neu im System bereitgestellte Geräte kommunizieren ihre Daten über entsprechende Topics, auf welchen der Microservice Subscriptions hält. Die empfangenen Daten liegen in einem standardisierten Nachrichtenformat

vor, was in der Theorie neben der Anbindung von Grove-Sensoren, auch die Anbindung von Geräten anderer Hersteller ermöglicht.

Durch die Datenzugriffschicht werden die empfangenen Nachrichten prozessiert und die relevanten Daten persistiert. Aufgrund der des MQTT-Protokolls zugrunde liegenden Broadcast-Kommunikation im Kontext eines Topics, erhalten alle Instanzen des Microservice zeitgleich eine Birth-, Death- oder State-Nachricht. An dieser Stelle wäre eine Synchronsierung der Instanzen notwendig um parallele Schreiboperationen der gleichen Daten zu vermeiden. Für dieses Problem wurde in der Konzeption bereits ein Lösungsansatz vorgestellt, um einen Koordinator zu bestimmen der die Schreiboperation ausführen darf. Eine Umsetzung dieser vorgestellten Lösung konnte im Rahmen der Implementierung nicht mehr umgesetzt werden.

Weiterhin kapseln bereitgestellte HTTP-Endpunkte Funktionalität um Sensoren und Aktoren über MQTT-Nachrichten anzusteuern. Dadurch ist eine ortsunabhängige, nachträgliche Konfiguration der Geräte über das Backendsystem möglich.

7.4.3 Universelle Schnittstelle

Damit persistierte Metadaten auch anderen Services zur Verfügung stehen, wurde eine REST-Schnittstelle umgesetzt, welche aufbauend auf der Datenzugriffsschicht CRUD-Operationen zur Verfügung stellt. Über diese können andere Anwendungen auf einfache Weise mit der Anwendung interagieren. Eine Erweiterung der Schnittstelle ist ebenso unkompliziert, da jedes verfügbare JPA-Repository eine REST-Ressource bereitstellt. Dementsprechend muss zur Erweiterung lediglich ein neu erstelltes Repository mit der *RestRepositoryResource* Annotation markiert werden. Dadurch muss jedoch auf die Funktionalität der *Spring-Data-Rest*-Erweiterung zurückgegriffen werden. Insofern weitere Funktionalität gewünscht ist, kann dies durch eigens implementierte Controller umgesetzt werden. Aufgrund der besseren Verständlichkeit, basiert die REST-Schnittstelle auf dem HATEOAS-Prinzip, wobei Relationen durch Links auf weitere Ressourcen dargestellt werden, was jedoch den Nachteil hat, dass dadurch ein zusätzlicher Kommunikationsaufwand entsteht.

Neben der REST-Schnittstelle wurde zusätzlich eine GraphQL-Schnittstelle umgesetzt. Diese steht im Gegensatz zum hohen Kommunikationsaufwand einer REST-Schnittstelle. GraphQL unterscheidet sich von REST darin, dass Anfragen die bei einer REST-Schnittstelle mehrere Anfragen erfordern, in einer einzelnen Anfrage gebündelt werden können. Hierzu stellt der GraphQL-Server einen einzelnen Endpunkt zur Verfügung der alle Anfragen serverseitig verarbeitet und über die Datenzugriffsschicht die angeforderten Daten bezieht und in einer Serverantwort an den Client zurückschickt. Da GraphQL eine strenge Typisierung mittels eines Schemas umsetzt, konnte nicht das gesamte Datenmodell umgesetzt werden. Weiterhin konnte nur eine rudimentäre Implementierung umgesetzt werden, welche lediglich einfache Lese- und Schreiboperationen erlauben. Diese beschränken sich auf die Abfrage von allen Elementenen einer Entität oder eine spezifische Entität, welche Anhand ihrer ID abgerufen wird. Neue Entitäten können über die spezifizierten Mutation-Operationen erstellt oder gelöscht werden. Eine Erweiterung der GraphQL-Schnittstelle kann über die Angabe der neuen Entitäten innerhalb des Schemas, sowie der Implementierung eines GraphQL-Controllers erfolgen. Die vorliegende Implementierung ist aufgrund der einfachen Umsetzung eher exemplarischer Natur, um die wesentlichen Konzepte darzustellen. Um Overhead in der Spring Anwendung zu vermeiden, sollte evaluiert werden, ob an dieser Stelle eine eigenständige Serverlösung, welche ausschließlich

GraphQL-Anfragen verarbeitet, sinnvoller ist.

Aufgrund einer scheinbaren Inkompatibilität mit der verwendeten *Spring-Boot-Version 2.7.0-M1* konnte die GraphQL-Schnittstelle nicht ausreichend getestet werden, da der Endpunkt */graphql* nicht erreichbar war. Lediglich das mitgelieferte Tool *GraphiQL* zum interaktiven Testen von GraphQL-Queries konnte verwendet werden und war unter dem Endpunkt */graphiql* zu erreichen.

Um Restriktionen für den Zugriff auf Ressourcen über die Schnittstellen festzulegen, war die Integration von *Spring-Security* angedacht. Die Umsetzung der Integration konnte im zeitlichen Rahmen nicht mehr durchgeführt werden.

7.4.4 Cluster-Deployment

Aufgrund der Entwicklung der Anwendung im Kontext des CiTe-Testfelds, stellt die Bereitstellung auf dem Kubernetes Cluster eine zentrale Anforderung dar. Zur Umsetzung wurde eine Deployment-Konfiguration sowie mehrere Konfigurationsobjekte erstellt um der Anwendung alle notwendigen Daten zur Verfügung zu stellen. Durch die Bereitstellung der Konfigurationsobjekte ist eine externalisierte Konfiguration der Spring-Boot-Anwendung möglich, indem Key-Value-Paare als Konfigurationsparamter gespeichert werden und den Instanzen als Umgebungsvariablen zur Verfügung gestellt werden. So können über die Deployment-Konfiguration auf einfache Weise mehrere Instanzen der Anwendung auf dem Cluster bereitgestellt werden. Da der Microservice *stateless* ist, ist dadurch implizit eine hohe Verfügbarkeit und Resilienz gewährleistet. Fällt ein einzelner Service aus, sind die anderen Services immer noch in der Lage die eingehende Nachrichten zu verarbeiten. Eine Skalierung des Service ist durch eine manuelle Anpassung des Deployments möglich, eine automatische Skalierung wurde nicht umgesetzt.

7.5 Sensoranbindung

Anforderung	Umsetzung	Erfüllt
FR.PI.001	Speicherung der Konfiguration in lokaler Datei.	Erfüllt
FR.PI.002	Senden von Birth- und Death-Zertifikaten	Erfüllt
FR.PI.003	Verarbeiten von CONFIG-Nachrichten.	Erfüllt
FR.PI.004	Laden der Konfiguration aus einer Datei.	Erfüllt
FR.PI.005/006	Senden und Empfangen über DATA- und CMD-Nachrichten	Erfüllt
FR.PI.007	Logging durch Python Logging Modul.	Erfüllt
FR.PI.008	Aufgrund technischer Limitierung nicht umgesetzt.	Nicht erfüllt
NFR.PI.001	Erweiterbar durch Interfaces und modularen Aufbau.	Erfüllt
NFR.PI.002	Lauffähig auf Raspberry Pi.	Erfüllt

Tabelle 7.4: Sensoranbindung - Erfüllung der Anforderungen

Eine Übersicht des Fertigstellungsgrad der spezifizierten Anforderungen ist Tabelle 7.4 zu entnehmen. Die Anwendung zur Anbindung von Sensoren basiert auf einem modu-

laren Aufbau, wodurch sie für beliebige Geräte des Grove-Ökosystems erweitert werden kann, indem gegebenenfalls weitere Controller und Callbacks implementiert werden. Lediglich die Anforderung zur automatischen Erfassung der Sensoren konnte nicht umgesetzt werden. Da diese aber auf technischen Limitierungen beruht, dementsprechend nicht umgesetzt werden konnte, wurde diese Anforderung nicht in den Fertigstellungsgrad miteinbezogen. Durch die Integration der konzipierten Topic-Struktur ist es möglich bereits physisch angeschlossene Geräte über die vorgestellten Topics zu konfigurieren. Alternativ können diese durch eine Konfigurationsdatei lokal konfiguriert werden. In dieser wird zeitgleich auch die konkrete Hardwarekonfiguration von angeschlossenen Sensoren verwaltet. Im Kern der Arbeit stand hauptsächlich die Verwaltung der Sensoren, was die Bereitstellung, Konfiguration und die Persistierung der Metadaten im Backend einschließt. Aufgrund der vorgestellten Topic-Struktur konnte jedoch auch eine Möglichkeit zur standardisierten Kommunikation von Steuerungsbefehlen und erfassten Sensordaten umgesetzt werden. Zur Überwachung der Anwendungsausführung wurde rudimentäres Logging mittels der Python eigenen Module umgesetzt, als auch die Kommunikation von Statusnachrichten über den Message-Broker.

Weiterhin ist die Anwendung auf einem Raspberry-Pi ausführbar. Dafür muss lediglich eine Python Version > 3.6 installiert werden, sowie die benötigten Abhängigkeiten, welche in einem Requirements-File hinterlegt sind. Eine weitere Limitierung die während der Implementierung beobachtet werden konnte, beruht auf der Verwendung der Python Implementierung von Threads. Jede *DeviceManager*-Instanz initialisiert die Routine zur Erfassung und Versenden von Sensordaten in einem eigenen Thread. Das *Global Interpreter Lock* verhindert dabei die echte parallele Ausführung von Python-Bytecode. Dadurch werden in Threads ausgeführte Operationen lediglich nebenläufig und nicht parallel ausgeführt. Für eine echte parallele Ausführung müsste die Implementierung auf das *multiprocessing*-Modul umgestellt werden. Dadurch wird für jeden so erstellten Prozess eine eigene Python-Interpreter-Instanz gestartet, wodurch jedoch ein entsprechender Overhead entsteht, der gerade bei leistungsschwacheren Geräten beachtet werden sollte. Da die Zahl der Sensoren, die an das GrovePi+-Board angeschlossen werden können, aufgrund der verfügbaren Anschlussmöglichkeiten sehr überschaubar ist, wurde die Verwendung des Python-Threading-Moduls als ausreichend befunden.

7.6 Monitoring & Logging

Mit der Bereitstellung von Grafana, Prometheus und Loki stehen Tools zum Erfassen von Metriken, zur Aggregation von Logs zur Verfügung. Die Verwendung beschränkt sich hierbei nicht auf die Überwachung der in dieser Arbeit konzipierten Anwendungen, sondern kann für jegliche im Cluster bereitgestellten Anwendungen verwendet werden. Durch Bereitstellung von *ServiceMonitor*-Objekten können weitere Targets zur Prometheus-*ScrapeConfig* hinzugefügt werden. Da Loki über die Kubernetes-API in der Lage ist alle Log-Nachrichten abzugreifen, benötigt das Deployment an dieser Stelle keine weitere Konfiguration und kann für alle Anwendungen universell genutzt werden. Die Visualisierung erfordert lediglich den Zugriff über einen Service. Diese kann entweder während der Entwicklung einer Anwendung durch die *Port Forwarding*-Funktion von *kubectl* genutzt werden oder alternativ kann ein Service über einen *NodePort* zur Verfügung gestellt werden.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Die Arbeit befasst sich mit einem wesentlichen Aspekt von modernen IoT-Systemen — der Verwaltung von Geräten, welche im Rahmen einer Smart City Daten zur Verfügung stellen oder Daten zur Steuerung von Aktoren entgegennehmen. Im Fokus steht die Bereitstellung von Sensoren und deren Metadaten im Gesamtsystem und der Zugriff und die Manipulation dieser Daten über eine zentrale Schnittstelle.Dabei wurde evaluiert, welche Art der Datenhaltung und welche Technologie zum Abruf von Daten am besten geeignet ist.

Da eine hohe Konsistenz der Daten sowie strukturierte Datenabfrage eine große Rolle spielen, wurde CockroachDB als hoch verfügbares und horizontal skalierbares Datenhaltungssystem ausgewählt. Außerdem wurde ein Microservice entwickelt, welcher ermöglicht Metadaten über eine universelle Schnittstelle zu verwalten. Dabei wurde ein Datenmodell, bestehend aus mehreren Entitäten entwickelt, welche die relevanten Metadaten, die zur Verwaltung notwendig sind, abbilden. Neu eingebundene Sensoren und Aktoren werden automatisch anhand der über Birth-Zertifikate bereitgestellten Informationen in der Datenbank persistiert. Weiterhin besteht die Möglichkeit Sensoren und Aktoren über den Microservice anzusteuern. Dabei können Geräte an den Raspberry Pi angeschlossene Geräte über Konfigurationsnachrichten verwaltet werden. Als universelle Schnittstelle zur Datenabfrage wurde sowohl eine REST-Schnittstelle als auch eine GraphQL-Schnittstelle implementiert. Da es sich bei GraphQL um eine Technologie handelt, die immer mehr an Popularität gewinnt, wurde diese ergänzend zur REST-Schnittstelle implementiert um auch für zukünftige Arbeiten eine Grundlage zu bieten diese Technologie und die potentiellen Anwendungsmöglichkeiten zu erforschen. Beide Schnittstellen bauen auf Repositories auf, die jeweils einen Entitätentyp verwaltet. Dadurch ergibt sich ein modularer Aufbau der eine einfache Erweiterung des Systems ermöglicht.

Um eine einheitliche und eine standardisierte Hersteller - und projektübergreifende Kommunikation zwischen Sensoren und Backend zu ermöglichen, wurde eine Kommunikationsstruktur konzipiert, die universell mit allen MQTT-fähigen Geräten verwendbar ist. Dabei orientiert sich diese nah an der bereits existierenden Eclipse-Sparkplug-Spezifikation, simplifiziert jedoch wesentliche Aspekte, um diese einfacher im gegebenen Kontext nutzen zu können. Es werden Kommunikationskanäle zur Bereitstellung und Abmeldung, sowie zum Senden von Sensordaten und zum Empfang von Steuerungsbefehlen und Konfigurationsnachrichten bereitgestellt.

Diese festgelegten Kommunikationskanäle nutzt die in Python implementierte Anwendung zur Anbindung der Sensoren und Aktoren. Die Umsetzung der Anwendung folgt ebenso einem modularen Aufbau. Es werden jeweils Interfaces für die Broker-Kommunikation (Client-Interface) und Hardwareinteraktion (Controller-Interface) mit Sensoren und Aktoren erstellt. Respektive erfolgte eine Interface-Implementierung der Broker-Kommunikation für den Raspberry-Pi als Gateway und für anzubindende Geräte als Devices. Da sich Sensoren und Aktoren in ihrer Funktionsweise unterscheiden, wur-

den Interfaces für Sensor-und Aktor-Controller erstellt und jeweils eine generische Version implementiert, die lediglich rudimentäre Lese- und Schreiboperationen unterstützt. Eine Erweiterung ist durch Implementierung weiterer Controller möglich. Damit Geräte auch nach Ausfall verfügbar sind, wurde ein einfacher Backup-Mechanismus umgesetzt, indem die aktuelle Konfiguration gespeichert und bei Neustart geladen wird.

Damit zu jeder Zeit geprüft werden kann, ob Deployments im Cluster fehlerfrei laufen wurden zur Überwachung Prometheus zur Erfassung von Metriken, Loki zur Aggregation von Log-Nachrichten und Grafana auf dem Cluster deployed. Mittels Grafana werden Dashboards erstellt, um Anwendungen zu überwachen. Dabei beschränkt sich die Überwachung nicht auf die in dieser Arbeit konzipierten Anwendungen, sondern ermöglicht die Überwachung für alle Anwendungen im Cluster.

Der Backend-Service wurde so umgesetzt, dass sich dieser leicht skalieren lässt. Broker-Nachrichten werden grundsätzlich per Broadcast-Routing an alle Konsument eines Topics verteilt. Dadurch erhält jede Instanz des Service bereitgestellte Metadaten. Auch erst nachträglich initialisierte Instanzen erhalten Nachrichten durch die *retained* Funktion des Brokers. Dadurch sind die Microservices zustandslos, da sie ihre Subscriptions nicht selbst verwalten müssen. Der Service kann beliebig im Cluster-Kontext über die zur Verfügung gestellte Deployment-Konfiguration skaliert werden. So ermöglicht das erarbeitete Konzept eine flexible Möglichkeit zu Verwaltung von Sensoren und Aktoren im CiTe-Testfeld.

8.2 Ausblick

Die GraphQL-Schnittstelle wurde lediglich rudimentär umgesetzt und bildet aufgrund technischer Limitierungen nicht das gesamte Datenmodell ab. Für zukünftige Arbeiten und Anwendungszenarien wäre eine Evaluation einer eigenständigen, skalierbaren GraphQL-Serverlösung im CiTe-Testfeld in Betracht zu ziehen. Hier könnten beispielsweise Serverlösungen wie Apollo verwendet werden. [50] Weiterhin ist die Umsetzung einer Zugrifsskontrolle für die universellen Schnittstellen angedacht, die im Rahmen der Arbeit nicht umgesetzt werden konnte. Durch Konfiguration der Schnittstellen soll der Zugriff beispielsweise auf User-Ebene für einzelne Ressourcen eingeschränkt werden. Um eine allgemeine Performance-Steigerung zu erzielen kann die Anwendung durch Integration von Caching-Mechanismen erweitert werden. Dadurch kann speziell bei häufig gestellten Anfragen eine kürzere Antwortzeit erzielt werden. Ein weiterer Punkt stellt die Umsetzung der in Abschnitt 5.5.2 vorgestellten Leader-Election dar. Diese dient zur Auswahl eines koordinierenden Microservices, der berechtigt ist neu bereitgestellte Metadaten zu persistieren.

Da im speziellen Java-Docker-Images meist sehr ressourcenintensiv sind und oft eine lange Startup-Zeit besitzen, stellt die Evaluation von Alternativen eine Möglichkeit zur Verbesserung der Performance dar. Im speziellen arbeitet das Projekt Quarkus an einer einfachen und ressourcenschonenden Möglichkeit Java-Applikationen performanter in einer Cluster-Umgebung zu betreiben. [67] Quarkus verspricht hierbei eine wesentlich schnellere Startzeit, sowie eine wesentlich geringerer Speicherauslastung, was speziell für Spring Boot Anwendungen eine leichtgewichtigere Alternative sein könnte.

Literatur

[1] *A testbed for smart city applications and architectures*. URL: https://dsl.htwsaar.de/Research.html. (accessed: 05.03.2022).

[2] *Abstract Base Classes*. URL: https://docs.python.org/3/library/abc.html. (accessed: 24.04.2022).

[3] Julian Altmeyer. *Konzeption und Implementierung einer Architektur zur Verteilung von Steuerungs-und Navigationsaufgaben an Roboter in einem Smart-City-Umfeld*. Nov. 2021.

[4] Eric Brewer. „CAP twelve years later: How the "rules"have changed“. In: *Computer* 45 (2 Jan. 2012), S. 23–29. ISSN: 0018-9162. DOI: 10.1109/mc.2012.37.

[5] *Callbacks*. URL: https://www.eclipse.org/paho/index.php?page=clients/python/docs/index.php#callbacks. (accessed: 13.02.2022).

[6] *CockroachDB Beta Passes Jepsen Testing*. URL: https://www.cockroachlabs.com/blog/cockroachdb-beta-passes-jepsen-testing. (accessed: 06.02.2022).

[7] George Coulouris, Jean Dollimore, Tim Kindberg und Blair Gordon. *Distributed Systems - Concepts and Design*. 2012.

[8] *Deploy CockroachDB in a Single Kubernetes Cluster*. URL: https://www.cockroachlabs.com/docs/stable/deploy-cockroachdb-with-kubernetes.html. (accessed: 19.03.2022).

[9] *Derived Data Queries*. URL: https://www.baeldung.com/spring-data-derived-queries. (accessed: 17.02.2022).

[10] Apache Commons Documentation. *Apache Commons - Best Practices*. URL: https://commons.apache.org/proper/commons-logging/guide.html#Best_Practices_General. (accessed: 19.01.2022).

[11] Cassandra Documentation. *Cassandra - Tunable Consistency*. URL: https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html#tunable-consistency. (accessed: 20.12.2021).

[12] Cockroach Documentation. *CockroachDB FAQs*. URL: https://www.cockroachlabs.com/docs/stable/frequently-asked-questions.html. (accessed: 24.01.2022).

[13] Google Protobuf Documentation. *Overview*. URL: https://developers.google.com/protocol-buffers/docs/overview.

[14] Kubernetes Documentation. *Externalizing config using MicroProfile, ConfigMaps and Secrets*. URL: https://kubernetes.io/docs/tutorials/configuration/configure-java-microservice/configure-java-microservice/. (accessed: 11.01.2022).

[15] Kubernetes Documentation. *Kubernetes - ConfigMaps*. URL: https://kubernetes.io/docs/concepts/configuration/configmap/. (accessed: 11.01.2022).

[16] Kubernetes Documentation. *Kubernetes - Configuration Best Practices*. URL: https://kubernetes.io/docs/concepts/configuration/overview/. (accessed: 04.02.2022).

[17] Kubernetes Documentation. *Kubernetes - Containers*. URL: https://kubernetes.io/docs/concepts/containers/. (accessed: 28.12.2021).

[18] Kubernetes Documentation. *Kubernetes - Deployment*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. (accessed: 22.12.2021).

[19] Kubernetes Documentation. *Kubernetes - Images*. URL: https://kubernetes.io/docs/concepts/containers/images/. (accessed: 28.12.2021).

[20] Kubernetes Documentation. *Kubernetes - Namespaces*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. (accessed: 09.01.2022).

[21] Kubernetes Documentation. *Kubernetes - Networking*. URL: https://kubernetes.io/docs/concepts/services-networking/. (accessed: 09.01.2022).

[22] Kubernetes Documentation. *Kubernetes - Pods*. URL: https://kubernetes.io/de/docs/concepts/workloads/pods/. (accessed: 05.01.2022).

[23] Kubernetes Documentation. *Kubernetes - ReplicaSet*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. (accessed: 08.01.2022).

[24] Kubernetes Documentation. *Kubernetes - Secrets*. URL: https://kubernetes.io/docs/concepts/configuration/secret/. (accessed: 11.01.2022).

[25] Kubernetes Documentation. *Kubernetes - Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[26] Kubernetes Documentation. *Kubernetes - Volumes*. URL: https://kubernetes.io/docs/concepts/storage/volumes/. (accessed: 10.01.2022).

[27] Kubernetes Documentation. *Kubernetes - What is Kubernetes*. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (accessed: 22.12.2021).

[28] Kubernetes Documentation. *Kubernetes Logging at the node level*. URL: https://kubernetes.io/docs/concepts/cluster-administration/logging/#logging-at-the-node-level. (accessed: 17.01.2022).

[29] Kubernetes Documentation. *expose*. URL: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#expose. (accessed: 28.04.2022).

[30] Maven Documentation. *Introduction to POM*. URL: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html. (accessed: 16.11.2021).

[31] Maven Documentation. *Introduction to the Build Lifecycle*. URL: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. (accessed: 16.11.2021).

[32] Microsoft Documentation. *Transaktionsisolationsstufen (ODBC)*. URL: https://docs.microsoft.com/de-de/sql/odbc/reference/develop-app/transaction-isolation-levels?view=sql-server-ver15. (accessed: 18.01.2022).

[33] RabbitMQ Documentation. *AMQP 0-9-1 Model Explaines*. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html. (accessed: 08.11.2021).

[34] RabbitMQ Documentation. *MQTT Plugin*. URL: https://www.rabbitmq.com/mqtt.html. (accessed: 08.11.2021).

[35] Spring Documentation. *Spring - Boot*. URL: https://spring.io/projects/spring-boot. (accessed: 14.02.2022).

[36] Spring Documentation. *Spring - Core Technologies*. Jan. 2022. URL: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html. (accessed: 14.02.2022).

[37] Spring Documentation. *Spring - Overview*. URL: https://spring.io/projects/spring-framework. (accessed: 14.02.2022).

[38] *Eclipse Paho*. URL: https://www.eclipse.org/paho/index.php?page=downloads.php. (accessed: 20.11.2021).

[39] Eric Evans. *Domain Driven Design*.

[40] Roy Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. (accessed: 13.12.2021).

[41] Eclipse Foundation. *Sparkplug™ Specification Version 2.2*. 2019.

[42] Martin Fowler. *Richardson Maturity Model - steps toward the glory of REST*. März 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html. (accessed: 13.12.2021).

[43] *Grafana*. URL: https://grafana.com. (accessed: 18.02.2022).

[44] *Grove System*. URL: https://wiki.seeedstudio.com/Grove_System/. (accessed: 07.11.2021).

[45] *Grovepy Python Library*. URL: https://github.com/DexterInd/GrovePi/tree/master/Software/Python. (accessed: 17.12.2021).

[46] Red Hat. *Virtuelle Maschinen (VM) - Funktionsweise, Typen, Anwenung*. Sep. 2019. URL: https://www.redhat.com/de/topics/virtualization/what-is-a-virtual-machine#:~:text=Mithilfe%20von%20Virtualisierungstechnologien%20kÃűnnen%20Sie,Umgebung%20an%20die%20VMs%20partitioniert. (accessed: 30.11.2021).

[47] *Hiberante Spatial*. URL: http://www.hibernatespatial.org/. (accessed: 15.01.2022).

[48] *Homie Convention*. URL: https://homieiot.github.io/specification/. (accessed: 07.12.2021).

[49] *I2C And I2C Address of Seeed Product*. URL: https://wiki.seeedstudio.com/I2C_And_I2C_Address_of_Seeed_Product/. (accessed: 27.12.2021).

[50] *Introduction to Apollo Server*. URL: https://www.apollographql.com/docs/apollo-server/. (accessed: 13.03.2022).

[51] *Introspection*. URL: https://graphql.org/learn/introspection/. (accessed: 10.12.2021).

[52] *JSON comes to cockroach*. URL: https://www.cockroachlabs.com/blog/json-coming-to-cockroach/. (accessed: 27.01.2022).

[53] *Jepsen*. URL: https://jepsen.io/. (accessed: 06.02.2022).

[54] *Known Limitiations*. URL: https://www.cockroachlabs.com/docs/v21.2/known-limitations#spatial-support-limitations. (accessed: 24.01.2022).

[55] *Lessons Learned from 2+ Years of Nightly Jepsen Tests*. URL: https://www.cockroachlabs.com/blog/jepsen-tests-lessons/. (accessed: 06.02.2022).

[56] *Loki*. URL: https://grafana.com/oss/loki/#:~:text=Loki%20is%20a%20horizontally%20scalable,labels%20for%20each%20log%20stream. (accessed: 04.03.2022).

[57] *MQTT & MQTT 5 Essentials A comprehensive overview of MQTT facts and features for beginners and experts alike eBook*.

[58] *MoSCoW*. URL: https://www.projektmagazin.de/glossarterm/moscow#:~:text=MoSCoW%20ist%20ein%20Akronym%20f%C3%BCr,S%3A%20Should%20(Soll). (accessed: 27.11.2021).

[59] *Monitoring and Alerting*. URL: https://www.cockroachlabs.com/docs/stable/monitoring-and-alerting.html. (accessed: 19.03.2022).

[60] *MqttAsyncClient*. URL: https://www.eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/MqttAsyncClient.html. (accessed: 13.02.2022).

[61] Sam Newman und an O'Reilly Media Company. Safari. *Building Microservices, 2nd Edition*. 2021, S. 400. ISBN: 9781492034025.

[62] *Performance Benchmarking with TPC-C*. URL: https://www.cockroachlabs.com/docs/v21.2/performance-benchmarking-with-tpcc-small. (accessed: 25.03.2022).

[63] *Performance*. URL: https://www.cockroachlabs.com/docs/stable/performance.html. (accessed: 27.11.2021).

[64] *Prometheus Operator*. URL: https://github.com/prometheus-operator/prometheus-operator. (accessed: 16.04.2022).

[65] *Prometheus*. URL: https://github.com/prometheus/prometheus. (accessed: 23.02.2022).

[66] Hannah Ritchi und Max Roser. *Urbanization - Our World in Data*. 2018. URL: https://ourworldindata.org/urbanization#by-2050-more-than-two-thirds-of-the-world-will-live-in-urban-areas.

[67] *SUPERSONIC/SUBATOMIC/JAVA*. URL: https://quarkus.io/. (accessed: 26.02.2022).

[68] Mayank Shah. *Kubernetes - Leader Election*. 2021. URL: https://itnext.io/leader-election-in-kubernetes-using-client-go-a19cbe7a9a85. (accessed: 05.04.2022).

[69] Shree Krishna Sharma, Dushantha K Nalin Jayakody, Symeon Chatzinotas und Alagan Anpalagan. *Communication Technologies for Networked Smart Cities*. 2021.

[70] RFC Specification. *A Universally Unique IDentifier (UUID) URN Namespace*. 2005. URL: https://datatracker.ietf.org/doc/html/rfc4122.

[71] RFC Specification. *The JavaScript Object Notation (JSON) Data Interchange Format*. 2017. URL: https://datatracker.ietf.org/doc/html/rfc8259#section-3.

[72] GraphQL Spezifikation. *GraphQL*. 2018. URL: https://spec.graphql.org/June2018/. (accessed: 09.12.2021).

[73] *Spring ApplicationListener*. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationListener.html. (accessed: 03.04.2022).

[74] *Spring Data rest*. URL: https://docs.spring.io/spring-data/rest/docs/current/reference/html/#reference. (accessed: 17.02.2022).

[75] *Springdoc Openapi*. URL: https://springdoc.org/. (accessed: 15.04.2022).

[76] Maverick Studer. *Konzeption und Implementierung einer Microservices-Architektur zur Sensordatenverarbeitung und -aggregation für IoT-Anwendungen*. Hochschule für Technik und Wirtschaft des Saarlandes, Sep. 2021.

[77] *Swagger UI*. URL: https://swagger.io/tools/swagger-ui/.

[78] *Thingsboard Documentation*. URL: https://thingsboard.io/docs/. (accessed: 13.12.2021).

[79] Raphaela Wagner. *Dynamische Anbindung von Sensoren und Aktoren an ein Cloud-Backend*. Nov. 2020.

[80] *What is I2C*. URL: https://wiki.seeedstudio.com/I2C_And_I2C_Address_of_Seeed_Product/. (accessed: 07.11.2021).

[81] Eberhard Wolff. *Microservices: Flexible Software Architecture*. 2017.

[82] *Work with spatial data*. URL: https://www.cockroachlabs.com/docs/stable/spatial-data.html. (accessed: 15.01.2022).

[83] Chuan-Kun Wu. *Internet of Things Security*. DOI: https://doi.org/10.1007/978-981-16-1372-2. URL: http://www.springer.com/series/13197.

[84] Bernd Leukert et al. „IoT 2020: Smart and secure IoT platform“. In: (2020). (accessed: 16.12.2021).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

HATEOAS	Hypermedia as the Engine of Application State
CRUD	Create Read Update Delete
API	Application Programming Interface
REST	Representational State Transfer
EoN	Edge of Network
JSON	Java Script Object Notation
UUID	Universally Unique Identifier
ID	Identifier
HTTP	Hypertransfer Transmission Protocol
XML	Extensible Markup Language
YAML	Yet Another Markup Language
SQL	Structured Query Language
NOSQL	Not only SQL
CLI	Commande Line Interface
MQTT	Message Queuing Telemetry Transport
SDO	Standard Defining Organization
IEC	International Electrotechnical Commission
IoT	Internet of Things
IIoT	Industrial Internet of Things
AISEC	Applied and Integrated Security
QoS	Quality of Service
LWT	Last Will and Testament
SSL	Secure Sockets Layer
TLS	Transport Layer Security
CPU	Central Processing Unit
POJO	Plain Old Java Object
JVM	Java Virtual Machine
ACID	Atomicity Consistency Isolation Durability
CAP	Consistency Availability Partition Tolerance
HAL	Hypertext Application Language
URI	Uniform Resource Identifier
POM	Project Object Model

MOM	Message Oriented Middleware
M2M	Machine to Machine
SOP	Single Point of Failure
DSL	Distributed Systems Lab
AMQP	Advanced Message Queing Protocol
CoAP	Constrained Application Protocol
SNMP	Simple Network Monitoring Protocol
LwM2M	Ligthweight Machine to Machine Communication
DNS	Domain Name System
IP	Internet Protocol
PV	Persistent Volum
PVC	Persistent Volume Claim
ORM	Object Relational Mapping
JPA	Java Persistence API
URL	Uniform Resource Locator
JAR	Java Archive

Anhang

A Weiterführende Informationen zur Implementierung

A.1 Spring Microservice - Application Properties

```
spring:
  datasource:
    url: jdbc:postgresql://HOST:PORT/DATABASE?PARAMETERS
    username: USER
    password: PASSWORD
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: update
    open-in-view: false
    show-sql: true
  data:
    rest:
      detection-strategy: annotated
  graphql:
    graphiql:
      enabled: true
      path: /graphiql
    path: /graphql
    schema:
      printer:
        enabled: false
server:
  port: 8080
broker:
  hostname: localhost
  port: 1883
  use_ssl: false
  ssl:
    username: USER
    user_pwd: PASSWORD
    ca_file: NOSSL
    client_crt_file: NOSSL
    client_key_file: NOSSL

management:
  endpoints:
```

```
38      enabled-by-default: true
39      web:
40        discovery:
41          enabled: true
42        exposure:
43          include: "health,info,metrics,loggers,env,prometheus"
```

Listing A.1: Konfigurationsdatei application.yaml der Spring Anwendung

A.2 Spring Microservice - Konfigurationsobjekte

A.2.1 Externalisierte Konfiguration

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: application-config
5    namespace: pgoelter
6  data:
7    SPRING_DATASOURCE_URL: "jdbc:postgresql://HOST/DB-NAME?PARAMETER"
8    SPRING_DATASOURCE_USERNAME: "USER"
9    SPRING_DATASOURCE_PASSWORD: "PASSWORD"
10   SPRING_DATASOURCE_DRIVER-CLASS-NAME: "org.postgresql.Driver"
11   SPRING_JPA_HIBERNATE_DDL-AUTO: "update"
12   SPRING_JPA_OPEN-IN-VIEW: "false"
13   SPRING_JPA_SHOW-SQL: "false"
14   SPRING_DATA_REST_DETECTION-STRATEGY: "annotated"
15   SERVER_PORT: "8080"
16   BROKER_HOSTNAME: "IP_OR_DNS-NAME"
17   BROKER_PORT: "8883"
18   BROKER_USE_SSL: "true"
19   BROKER_SSL_USERNAME: "USER"
20   BROKER_SSL_USER_PWD: "PASSWORD"
21   BROKER_SSL_CA_FILE: "/etc/broker/ca.crt"
22   BROKER_SSL_CLIENT_CRT_FILE: "/etc/broker/tls.crt"
23   BROKER_SSL_CLIENT_KEY_FILE: "/etc/broker/tls.key"
24   MANAGEMENT_ENDPOINTS_WEB_DISCOVERY_ENABLED: "true"
25   MANAGEMENT_ENDPOINTS_WEB_EXPOSURE: "health,metrics,prometheus,env,
       loggers,info"
```

Listing A.2: Application Properties werden durch diese Konfiguration überschrieben

A.3 Spring Microservice - Deployment

A.3.1 Deployment-Objekt

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: device-management-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: device-management-ms
  template:
    metadata:
      labels:
        app: device-management-ms
    spec:
      containers:
        - name: device-management-ms
          image: pgstudy/cite-management:dev4
          volumeMounts:
            - name: broker
              mountPath: "/etc/broker"
            - name: db
              mountPath: "/etc/db"
          envFrom:
            - configMapRef:
                name: application-config
            - secretRef:
                name: broker-credentials
            - secretRef:
                name: db-credentials
          ports:
            - containerPort: 8080
              name: http
          imagePullPolicy: Always
      volumes:
        - name: broker
          secret:
            secretName: rmq-certs
            optional: false
        - name: db
          secret:
            secretName: db-certs
            optional: false
```

Listing A.3: Kubernetes Deployment

A.3.2 Service-Objekt

,

```yaml
apiVersion: v1
kind: Service
metadata:
  name: device-management-service
  labels:
    app: device-management-ms
  namespace: pgoelter
spec:
  type: ClusterIP
  selector:
    app: device-management-ms
  ports:
    - port: 8080
      name: http-traffic
```

A.3.3 ServiceMonitor-Objekt

,

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: device-management-service-monitor
spec:
  selector:
    matchLabels:
      app: device-management-ms
  endpoints:
  - port: http-traffic
    path: "/actuator/prometheus"
```

A.4 Deployment - Monitoring Stack

1. Zunächst wird der Namespace *monitoring* erstellt:
   `kubectl create namespace monitoring`
2. Wechseln in den erstellten Namespace:
   `kubectl config set-context --current --namespace=monitoring`
3. Zur Installation des Prometheus-Operator muss das entsprechende Helm-Repository hinzugefügt werden, damit das Helm-Chart verfügbar ist:

```
helm repo add prometheus-community.github.io/helm-charts
helm repo update
```

4. Durch Installation des *kube-prometheus-stack* wird der Prometheus-Operator, eine betriebsbereite Prometheus-Instanz, sowie eine betriebsbereite Grafana-Instanz auf dem Cluster deployed:
```
helm install prometheus prometheus-community/kube-prometheus-stack
```

5. Beide Instanzen können über ihre entsprechenden Dashboards angesprochen werden. Zum Testen ob die Instanzen erreichbar sind kann auf diese, mittels der Port-Forwarding-Funktion zugegriffen werden.
Grafana-Dashboard:
```
kubectl port-forward --namespace monitoring service/prometheus-
   grafana 3000:80
```
Unter http://localhost:3000 ist die Grafana-Instanz erreichbar. Der durch *Prometheus-Operator* spezifizierte User ist *admin* und das Passwort lautet *prom-operator*.

Prometheus-Dashboard:
```
kubectl port-forward --namespace monitoring svc/prometheus-kube-
   prometheus-prometheus 9090:9090
```
Unter http://localhost:9090 ist das Prometheus-Dashboard erreichbar. Logins sind hier nicht weiter erforderlich.

6. Im nächsten Schritt wird Loki als Log-Aggregations-Anwendung installiert. Dazu muss wiederum das entsprechende Helm-Repository hinzugefügt werden:
```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
```

7. Die Installation erfolgt durch das entsprechende Helm-Chart:
```
helm upgrade --install loki grafana/loki-stack --set grafana.
   enabled=false,promtail.enabled=true, prometheus.enabled=false
```
Loki kann im Anschluss über die Grafana-Benutzeroberfläche als *DataSource* hinzugefügt werden. In der Konfiguration muss hierzu die Server-Url http://loki:3100 angegeben werden.

A.5 Deployment - CockroachDB

1. Zunächst muss die benötigte *CustomResourceDefinition* bereitgestellt werden:
```
kubectl apply -f https://raw.githubusercontent.com/cockroachdb/
   cockroach-operator/v2.6.0/install/crds.yaml
```

2. Im nächsten Schritt wird der *Operator* bereitgestellt, welcher im Anschluss das Deployment durchführt:

```
kubectl apply -f https://raw.githubusercontent.com/cockroachdb/
   cockroach-operator/v2.6.0/install/operator.yaml
```

Sind Änderungen in der Konfiguration gewünscht, kann die Datei heruntergeladen werden und angepasst werden. So kann Beispielsweise der vorkonfigurierte Namespace *cockroach-operator-system* angepasst werden.

3. Zuletzt muss der Cluster initialisiert werden. Hierzu wird zunächst die Konfiguration heruntergeladen:

```
curl -O https://raw.githubusercontent.com/cockroachdb/
   cockroach-operator/v2.6.0/examples/example.yaml
```

Hier können die verwendeten Ressourcen wie die Zahl der CPUs, der verwendete Arbeitsspeicher, sowie der verwendete Festplattenspeicher spezifiziert werden.

4. Zur Initialisierung wird die Konfiguration auf dem cluster bereitgestellt:
`kubectl apply -f example.yaml`

5. Damit User und Datenbanken erstellt werden können wird der mitgelieferte SQL-Client initialisiert:

```
kubectl create -f https://raw.githubusercontent.com/cockroachdb
   /cockroach-operator/master/examples/client-secure-operator.
   yaml
```

6. Mithilfe des folgenden Befehls kann wird eine Shell im entsprechenden Pod geöffnet und der SQL-Client gestartet:

```
kubectl exec -it cockroachdb-client-secure -- ./cockroach
   sql --certs-dir=/cockroach/cockroach-certs --host=
   cockroachdb-public
```

Nun können über die entsprechenden SQL-Befehle User und Datenbanken auf dem Cluster erstellt werden.

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.22, November 2021). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt