Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Verarbeitung komplexer IoT-Daten in der IoT-Plattform MBP

Tim Schneider

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Bernhard Mitschang
Betreuer/in:	Dr. Pascal Hirmer
Beginn am:	8. Dezember 2020
Beendet am:	8. Juni 2021

Kurzfassung

Die Multi-purpose Binding and Provisioning Platform (MBP) ist eine an der Universität Stuttgart entwickelte Internet of Things (IoT)-Plattform. Sie unterstützt eine Anbindung von IoT-Objekten, die als Sensorwerte jeweils einzelne Fließkommazahlen an die Plattform senden. Diese Daten werden von der MBP hauptsächlich zur Visualisierung und der Auswertung von IoT-Regeln verwendet. In dieser Arbeit wird das bestehende Datenmodell der MBP so erweitert, dass auch komplexe Sensordaten von der Plattform unterstützt werden können. Dazu wird ein neues Datenmodell-Konzept eingeführt, das die Verschachtelung von Sensordaten mittels Objekt- und Arraystrukturen ermöglicht sowie verschiedene primitive Datentypen als Datenfelder erlaubt. Es wird gezeigt, wie dieses Konzept in die MBP integriert werden kann und wie die notwendigen Anpassungen für die verschiedenen vom Datenmodell abhängigen Funktionalitäten der Plattform umgesetzt werden. Die Änderungen betreffen dabei alle architektonischen Schichten der Plattform. Die Anwendungslogik muss sowohl Datenmodelle als auch ihre konkreten Dateninstanzen verarbeiten können, was vor allem komplexe Zugriffsmechanismen erfordert. Die Resourcenschicht muss angepasst werden, um komplexe Daten speichern zu können und die Präsentationsschicht muss in der Lage sein, komplexe Daten dem Nutzer darzustellen sowie komplexe datenbezogene Nutzereingaben zu ermöglichen. Zuletzt betreffen die Änderungen auch die Operatorenskripte der angebundenen IoT-Umgebung, die Sensordaten in datenmodellkonformer Weise der Plattform bereitstellen müssen.

Abstract

The Multi-purpose Binding and Provisioning Platform (MBP) is an Internet of Things (IoT) platform developed at the University of Stuttgart. It supports the binding of IoT objects, each of which sending individual floating point values to the platform as sensor values. This data is mainly used by the MBP for visualization purposes and the evaluation of IoT rules. In this work, the existing data model of the MBP is extended so that complex sensor data can also be supported by the platform. For this purpose, a new data model concept is introduced that allows nesting of sensor data using object and array structures, as well as allowing different primitive data types as data fields. It will be shown how this concept can be integrated into the MBP and how the necessary adaptations for the various functionalities of the platform that depend on the data model are implemented. The changes affect all architectural layers of the platform. The application logic must be able to process both data models and their concrete data instances, which requires complex access mechanisms. The resource layer must be adapted to store complex data and the presentation layer must be able to present complex data to the user as well as to enable complex data-related user input. Lastly, the changes also affect the operator scripts of the connected IoT environment, which must provide sensor data to the platform in a data model-compliant manner.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

API Application Programming Interface. 22

ASCII American Standard Code for Information Interchange. 55

BSON Binary JSON. 21

CEP Complex Event Processing. 19

CRUD Create Read Update and Delete. 80

DOM Document Object Model. 35

DTDL Digital Twins Definition Language. 43

EPL Event Processing Language. 32

HTML Hypertext Markup Language. 35

HTTP Hypertext Transfer Protocol. 22

IETF Internet Engineering Task Force. 55

IoC Inversion of Control. 22

IoT Internet of Things. 17

IRI Internationalized Resource Identifiers. 46

JEE Jakarta Enterprise Edition. 22

JSON JavaScript Object Notation. 21

JSON-LD JSON for Linking Data. 43

MBP Multi-purpose Binding and Provisioning Platform. 17

MQTT Message Queuing Telemetry Transport. 23

POJO Plain Old Java Object. 22

QoS Quality of Service. 27

RDBMS Relational Database Management System. 21

REST Representational State Transfer. 22

RTSP Real Time Streaming Protocol. 110

SQL Structured Query Language. 21

SSE Server-Sent Events. 35

SSH Secure Shell. 19

UML Unified Modeling Language. 27

WAR Web Application Archive. 22

XML Extensible Markup Language. 66

XPath XML Path Language. 39

1. Einleitung

Das Internet der Dinge (Internet of Things (IoT)) beschreibt ein Konzept, das die Kommunikation heterogener Geräte auf autonome Weise unter dem Einsatz standardisierter Technologien ermöglicht, um gemeinsame Ziele zu erreichen [VF13]. Häufig handelt es sich bei diesen IoT-Geräten um Sensoren, die Daten zur Verarbeitung zur Verfügung stellen, auf deren Grundlage Aktionen in der Umgebung ausgelöst werden sollen. Dadurch kann beispielsweise auf im Gesamtsystem auftretende Ereignisse reagiert werden.

Tätigkeiten, die das Sammeln, Speichern und die Auswertung von Sensordaten zur Auslösung von Aktuatoren umfassen, werden im IoT häufig von zentralen Softwaresystemen übernommen, die als IoT-Plattformen bezeichnet werden [GBF+16; VF13]. Ein Beispiel für eine solche IoT-Plattform ist die Multi-purpose Binding and Provisioning Platform (MBP), die an der Universität Stuttgart entwickelt wird [SHS+20]. Sie verfügt über Funktionalitäten zur automatischen Bindung von IoT-Geräten, Speicherung von Sensordaten, Modellierung von Regeln, Datenanalyse und Auslösen von Aktuatoren. Dabei erlaubt die bisherige Version der MBP jedoch nur die Berücksichtigung einzelner Fließkommazahlen als Sensorwerte, die als Telemetriedaten von der Plattform entgegengenommen werden [SHS+20]. Das ermöglicht beispielsweise die Anbindung eines Temperatursensors, der den Temperaturwert in Grad Celcius angibt. Allerdings sind die auszutauschenden Daten im IoT häufig viel heterogener. Beispielsweise verfügen Beschleunigungssensoren über mehrere Datenachsen oder GPS-Sensoren über zwei Koordinaten. Zusätzlich gibt es auch Sensoren, die nicht nur Fließkommazahlen erzeugen, wie zum Beispiel Kameras und Mikrofone.

Die Notwendigkeit für eine IoT-Plattform in der Lage zu sein, komplexe Daten zu unterstützen, wird auch durch eine Erhebung durch Morais et al. unterstrichen [MSK19]. In ihr wurden 48 experimentelle Paper im Bereich des IoT hinsichtlich der Art der verwendeten Sensoren ausgewertet, um unter anderem zu ermitteln, welche Sensortypen besonders relevant für IoT-Anwendungen sind. Ordnet man diesen Sensortypen wahrscheinlichen benötigten Datentypen zu, wird erkenntlich, dass lediglich circa 60 Prozent der in der Studie erfassten IoT-Anwendungen mit dem derzeitigen Datenmodell einfacher Fließkommazahlen der MBP auskommen würden. Weitere 15 Prozent sind Sensoren mit mehreren Fließkommazahl-Datenachsen und die letzten 25 Prozent erfordern ferner andere Datentypen wie Strings oder Binärdaten.

Das Ziel dieser Bachelorarbeit ist es deshalb, die MBP dahingehend zu erweitern auch solche komplexen Sensordaten hinsichtlich ihrer Speicherung und Verarbeitung berücksichtigen zu können. Hierfür muss das bestehende Datenmodell der MBP erweitert und davon abhängige

Funktionen angepasst werden. Diese Anpassungen haben dabei nicht nur voraussichtliche Auswirkungen auf die Implementierung der MBP, sondern auch auf die Nachrichten, die zwischen den IoT-Geräten und der Plattform ausgetauscht werden.

Zur Umsetzung dieses Ziels ist es zunächst notwendig, die bisherige Architektur der MBP und die konkrete Implementierung einzelner ihrer bereitgestellten Funktionen zu analysieren, die eine direkte Abhängigkeit zum verwendeten Datenmodell besitzen. Deshalb wird im ersten Grundlagenkapitel die MBP vorgestellt. Dabei wird auf die allgemeine Funktionsweise der Plattform eingegangen, um ein besseres grundlegendes Bild von ihr zu vermitteln, sowie auf konkrete, für die Arbeit relevante, Implementierungsdetails. Anschließend folgt eine kurze Einführung in die Abfragesprache JsonPath, der eine wichtige Rolle im Hauptteil dieser Arbeit zukommt.

Kapitel 3 zeigt nachfolgend beispielhaft anhand von drei gängigen IoT-Diensten, wie andere IoT-Plattformen dem Problem der Unterstützung komplexer IoT-Daten entgegentreten. Nach einer kurzen Diskussion, ob sich diese Lösungen auch auf die MBP anwenden lassen, folgt der eigentliche Hauptteil dieser Arbeit, Kapitel 4. Dabei geht es um alle Konzepte und schließlich auch um konkrete Implementierungsmaßnahmen, die zur Lösung des formulierten Problems der Arbeit beitragen. Die getroffenen Änderungen an der MBP werden hierbei in der Reihenfolge vorgestellt, wie sie auch sukzessive in die Plattform integriert wurden. Auch wird gezeigt, wie die Nutzer der Plattform die Änderungen hinsichtlich der Bedienung der Benutzeroberfläche zu berücksichtigen haben. Am Ende der Arbeit steht eine kurze Zusammenfassung der erarbeiteten Lösung sowie ein Ausblick auf zusätzliche Erweiterungsmöglichkeiten der MBP, die sich aus dem neu hinzugefügten Datenmodell-Konzept ergeben.

2. Grundlagen

2.1. Die IoT-Plattform MBP

Die MBP ist eine IoT-Plattform zur Erstellung, Verwaltung und Überwachung von IoT-Umgebungen. Sie verfolgt zunächst das Ziel, eine für den Nutzer möglichst einfache Anbindung von IoT-Objekten wie beispielsweise Geräten mit Sensoren und Aktuatoren zu ermöglichen, indem der Grad an erforderlicher manueller Hardwarekonfiguration zur Geräteanbindung an die Plattform auf ein notwendiges Minimum reduziert wird. Dazu setzt die MBP auf einen automatisierten Anbindungsprozess, bei dem mittels Secure Shell (SSH) notwendige Softwareskripte auf der anzubindenden Hardware installiert werden können. Die Gesamtheit dieser Skripte, die das Ziel der Anbindung eines IoT-Objekts verfolgt, wird als Operator bezeichnet. Jedes Skript stellt dabei eine bestimmte Funktionalität zur Steuerung des Lebenszyklus einer Anbindung bereit. Dazu gehört die Installation, der Start, der Stop und die Deinstalltaion der Skripte. Neben der Anbindung einzelner Geräte ermöglicht die MBP auch das Erstellen und Verwalten ganzer IoT-Umgebungen. Diese können mittels eines grafischen Modellierungstools definiert werden. Anschließend kann eine gemeinsame Anbindung aller modellierten IoT-Objekte durchgeführt werden [SHS+20].

Zur Überwachung von eingehenden Sensordaten bietet die Benutzeroberfläche der MBP Dashboards, die Sensordaten und zugehörige Statistiken visualisieren. Des Weiteren können benutzerdefinierte Regeln auf Basis von Sensorwerten definiert werden, die Aktionen wie beispielsweise eine Kommandoversendung an Aktuatoren auslösen. Regeln können mittels grafischer Werkzeuge erstellt werden und werden von der MBP intern unter Verwendung von Complex Event Processing (CEP)-Technologien verwaltet und ausgewertet. Typische Anwendungsdomänen solcher Regelungssysteme finden sich beispielsweise in der Klimatechnik [SHS+20].

Die MBP wird als Open-Source-Projekt unter einer Apache-Softwarelizenz entwickelt und ihre Software-Artefakte zu diesem Zweck in einem GitHub Repository, zu finden unter der Literaturangabe [IPV], bereitgestellt. Hauptentwickler der Software sind Beschäftigte und Studierende des Instituts für Parallele und Verteilte Systeme (IPVS) der Universität Stuttgart. Alle Ausführungen in dieser Arbeit, die Bezug auf die bestehende Implementierung der MBP nehmen, beziehen sich auf die Inhalte des Master-Zweiges des Repositorys, wie er bis Ende März 2021 vorlag.

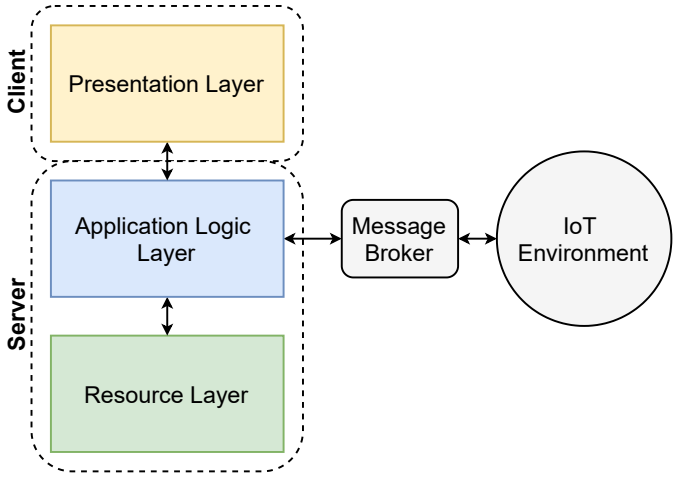

Abbildung 2.1.: Schematische Darstellung der Erweiterung der üblichen Drei-Schichten-Architektur für Web-Applikationen mit einer zusätzlichen Datenschnittstelle zur zu verwaltenden IoT-Umgebung.

Im Rahmen dieser Arbeit wird die derzeitige softwaretechnische Umsetzung der MBP angepasst, um auch komplexere IoT-Daten verarbeiten zu können. Dafür müssen im Vorfeld konzeptionelle Entwurfsentscheidungen und später konkrete Implementierungsmaßnahmen getroffen werden. Um diese besser nachvollziehen zu können, wird im folgenden Abschnitt der architektonische Gesamtaufbau der Plattform, sowie später im Detail, die derzeitigen technischen Lösungen, zur Verarbeitung simpler numerischer Daten, analysiert.

2.1.1. Aufbau und Architektur

Der software-architektonische Gesamtaufbau der MBP liegt einer Client-Server-Architektur zu Grunde. Dabei wird im Wesentlichen einer Drei-Schichten-Architektur gefolgt, mit Präsentationsschicht, Anwendungslogikschicht und Resourcenschicht. Ein fundamentaler Unterschied der MBP, im Vergleich zu anderen gängigen Web-Applikationen mit einer 3-Tier-Architektur, ist, dass die Schicht der Anwendungslogik nicht ausschließlich eine Schnittstelle zur Präsentations- und Resourcenschicht besitzt. Hinzu kommt eine Schnittstelle zur IoT-Umgebung, die von der MBP verwaltet werden soll. Zur Umsetzung dieser zusätzlichen Schnittstelle wird eine Message-Broker-Komponente verwendet. Über sie werden alle Nutzdaten zwischen der Plattform und ihren angebundener Objekte ausgetauscht. Dazu zählen beispielsweise Sensordaten, die an die Plattform gesendet werden, aber auch Kommandos, die ausgehend von der Plattform an Aktuatoren gesendet werden. Abbildung 2.1 stellt diese Erweiterung der Drei-Schichten-Architektur schematisch dar.

Zur unterstützenden Realisierung der einzelnen Schichten und ihrer bereitgestellten Funktionalitäten kommen verschiedene Softwaretechnologien zum Einsatz. Im Folgenden wird jede architektonische Schicht der MBP zusammen mit den dortigen primär verwendeten Technologien in kurzem Umfang vorgestellt.

Resourcenschicht

Die Datenschicht der MBP bildet eine MongoDB-Datenbank. Dies ist eine auf der Speicherung von Dokumenten basierende nicht-relationale Datenbank, die deshalb auch zur Gruppe der sogenannten Non-Structured Query Language (SQL)- oder NoSQL-Datenbanken gezählt wird. Statt vorgegebener Tabellenspalten mit fest definiertem Schema wie bei relationalen Datenbanksystemen (Relational Database Management System (RDBMS)) ermöglicht die MongoDB die Speicherung von Daten in sogenannten *Documents* [Cho13]. Diese werden innerhalb verschiedener Collections, also Sammlungen von Dokumenten, gruppiert und gespeichert. Die Inhalte von *Documents* orientieren sich dabei an der Struktur von JavaScript Object Notation (JSON)-Objekten, mit dem Unterschied, dass zusätzliche Datentypen unterstützt werden. Dieses Speicherformat wird intern binär kodiert verwaltet, weshalb es Binary JSON (BSON) genannt wird [Monb]. Genau wie mit JSON, bei der Objekte ineinander verschachtelt werden können, können auch Dokumente in andere Dokumente eingebettet werden, wodurch beliebig komplex verschachtelte Dokumentenstrukturen entstehen können. Diese Affinität zu JSON macht die MongoDB insbesondere für den Einsatz im Kontext von Webapplikationen geeignet, bei denen über das Netzwerk ausgetauschte Nutzdaten häufig ebenfalls in JSON vorliegen. Jedem neu hinzugefügtem Dokument wird von der MongoDB eine eindeutige Objekt-ID vergeben, die in einem automatisch generiertem BSON-Feld namens `_id` gespeichert wird und somit eine Primärschlüssel-Rolle einnimmt [Cho13].

Zur Optimierung von Datenbankoperationen lassen sich über Collections Indizes definieren. Dafür stehen verschiedene Index-Typen zur Verfügung wie solche über einzelne Felder, zusammengesetzte Felder oder über Array-Datenstrukturen. Dadurch soll eine sublineare Zugriffszeit (in Abhängigkeit der Dokumentenanzahl) auf Dokumente ermöglicht werden, sofern die Suchanfragen auf indizierte Felder abzielen. In ihrer Standardkonfiguration nutzt die MongoDB zur internen Speicherverwaltung die WiredTiger Storage Engine. Diese verwendet ein Dokumentencache-System, welches erstens dafür sorgt, dass Datenbankoperationen schneller verarbeitet werden können und zweitens mit Features wie Checkpointing und Journals die Konsistenz der Datenbank, auch bei gleichzeitigem Zugriff mehrerer Clients, auf Festplatten-Ebene garantiert [Mon20].

Unter anderem aufgrund dieses Dokumentencachings wird bei der Verwendung der MongoDB vom offiziellen MongoDB-Blog [Mat20] empfohlen, Daten, die in der Regel gleichzeitig abgerufen werden, gemeinsam in einem größeren Dokument zu speichern, anstatt etwa Datenbanknormalisierungen, wie mit RDBMS gängig, durchzuführen. Das kann mittels Dokumenteneinbettung statt Referenzierung erreicht werden. Zwar vermeiden Normalisierungen unnötige Redundanzen und sparen Speicherplatz, doch verursachen sie zusätzliche Collections für Dokumente, die in Indizes und dem Cache zusätzlich, bei der Auswertung von Datenbankoperationen, berücksichtigt werden müssen. Bei steigender Collection- und Dokumentenanzahl schränkt das die Performanz ein [Mat20]. Zu berücksichtigen ist bei der Erstellung großer eingebetteter Dokumente jedoch, dass die Größe von Dokumenten 16 Megabytes nicht überschreiten darf [Cho13].

Anwendungslogikschicht

Die Anwendungslogik der MBP wird innerhalb eines Apache-Tomcat-Webservers ausgeführt. Dabei handelt es sich um eine Open-Source-Implementierung von Java-Server-Technologien zur dynamischen Bereitstellung von Webinhalten. Dies wird unter anderem mittels Jakarta-Servlets ermöglicht [VG11]. Auf Grundlage des Tomcat-Servers kommt zusätzlich das Spring-Framework von VMWare zum Einsatz. Dies ist ein Web-Container zur Verwaltung von Jakarta- Servlets, der unter anderem als Top-Level-Framework für Jakarta Enterprise Edition (JEE) agieren kann [VMw21]. Zur einfacheren Verwendung und Konfiguration dieses Frameworks wird die vorgefertigte Spring-Boot-Lösung für die MBP eingesetzt. Das ist eine eine vorkonfigurierte Spring-Plattform mit, von den Entwicklern empfohlenen Third-Party-Bibliotheken, um möglichst schnell und mit wenig erforderlichem Vorwissen Spring-Applikationen entwickeln zu können. Unter anderem ist es mit Spring-Boot möglich eine direkte Anbindung zu Tomcat-Servern zu realisieren, ohne manuell Web Application Archive (WAR)-Dateien austauschen zu müssen [Wal16].

Da es sich bei Spring um ein Framework handelt, macht es Gebrauch von dem Prinzip des Inversion of Control (IoC). Das bedeutet, dass der resultierende Kontrollfluss der Anwendung vom Framework gesteuert wird und der Entwickler Programmcode durch die Verwendung vordefinierter Schnittstellen in die Kontrollflusslogik einkapseln kann. Im Fall von Spring erfolgt dies mittels Java-Annotationen. Ein Anwendungsfall für häufig verwendete Annotationen in Spring-Boot, sind solche zur Definition von Representational State Transfer (REST)-Schnittstellen. Über sie kann in einfacher Art und Weise definiert werden, welche Hypertext Transfer Protocol (HTTP)-Operationen für welche Netzwerkadresse, mit welchem Inhalt zulässig sind und wie das System auf Anfragen reagieren soll. In Verbindung mit der Drittbibliothek Jackson kann in diesem Kontext auch ein automatisches Mapping von JSON-Nutzdaten zu Plain Old Java Object (POJO)s definiert werden, was dem Entwickler manuelles Datenparsing abnimmt [Wal16].

Präsentationsschicht

Die Präsentationsschicht der MBP wird mittels des REST-Application Programming Interface (API), die von der Anwendungslogik bereitgestellt wird, an das Gesamtsystem angebunden. Dadurch entsteht eine lose Kopplung von Frontend und Backend. So ist es möglich, dass für die MBP unterschiedliche Visualisierungsanwendungen zum Einsatz kommen, wovon auch Gebrauch gemacht wird. Neben einer browserbasierten Single-Page-Anwendung wird eine Android-Applikation für Mobilgeräte angeboten. Diese funktioniert allerdings nur unter einer veralteten Version der MBP, weshalb die Android-Applikation nicht für die im Rahmen dieser Arbeit vorgestellten Weiterentwicklung der Plattform berücksichtigt wird. Der weitere Fokus, im Bezug auf die Benutzeroberfläche der MBP, liegt deshalb auf der browserbasierten Single-Page-Anwendung.

Zur Entwicklung der Browser-Anwendung kommt das JavaScript-basierte AngularJS-Framework zum Einsatz. Dieses unterstützt unter anderem eine bessere Modualisierung von JavaScript-Komponenten, wodurch auch Single-Page-Applikationen leichter umsetzbar werden. Unterstützend wird JQuery als JavaScript-Bibliothek eingesetzt.

Message-Broker zur Anbindung von IoT-Objekten

Message Queuing Telemetry Transport (MQTT) ist ein auf TCP/IP aufgebautes Netzwerkprotkoll, spezifiziert von Organisation OASIS [OAS14], um Nachrichten zwischen Geräten auszutauschen. Es unterstützt einen Publish-Subscribe-Mechanismus, weshalb es sich als Protokoll für Message-Queueing-Technologien eignet. Zusätzlich wirbt es damit, leichtgewichtig zu sein und somit auch für Netzwerke mit niedrigen Bandbreiten einsetzbar zu sein. Dies macht es insbesondere im Kontext des IoT zu einem häufig verwendeten Protokoll [OAS14].

Die Implementierung des Protokolls wird in der MBP von dem Message Broker Mosquitto übernommen, einer Open-Source-Software der Eclipse Foundation [Lig17]. Diese kümmert sich um alle Belange des Nachrichtenroutings unter Berücksichtigung des Publish-Subscribe-Mechanismus, der durch die MQTT-Spezifikation vorgegeben ist. In der Standardkonfiguraton der MBP läuft Mosquitto auf dem gleichen Rechnerknoten, der auch als MBP-Server fungiert. Es ist jedoch prinzipiell möglich eine arbiträre IP-Adresse als Mosquitto-Standort anzugeben. Jedes IoT-Objekt, das an die Plattform angebunden ist, hat eine eindeutige Identifikationsnummer, die gemeinsam mit einem Prefix für die Objekt-Art (Sensor, Aktuator, Gerät, Montoring Component) als Kodierung für ein MQTT-Topic dient. An dieses Topic veröffentlichen die jeweiligen Geräte dann die Nachrichten, die sie an die MBP senden wollen. Zu diesem Zweck abonniert die MBP jedes dieser Topics, um so Nachrichten von den jeweiligen angebundenen Geräte zu erhalten.

2.1.2. Grundlegende Funktionsweise

Nachdem im vorherigen Unterkapitel die grobe Gesamtarchitektur der MBP, mit Fokus auf den eingesetzten Technologien, vorgestellt wurde, soll nun näher auf die konkrete Funktionsweise der Plattform eingegangen werden. Dazu wird im Folgenden sowohl die Bedienung aus Nutzersicht, als auch die konkrete Softwareimplementierung vorgestellt. Der Fokus liegt dabei auf allen Funktionalitäten, die indirekt oder unmittelbar mit dem Umgang von komplexen IoT-Daten zusammenhängen. Gleichzeitig soll ein grundlegender Einblick in die Bedienung der verschiedenen Funktionalitäten der MBP gegeben werden, um eine bessere Vorstellung von der Plattform zu vermitteln.

Erstellen von IoT-Objekt-Entitäten zur Geräteanbindung

Zur Anbindung von Geräten an die MBP müssen zunächst drei verschiedene Instanzen von Entitäten in der MBP angelegt werden. Dabei handelt es sich um eine Instanz eines Operators, eines Gerätes und eines Sensors. Für jeder dieser drei Entitäten existiert eine eigene Collection in der MongoDB-Datenbank, mit jeweils einem eigenen Dokument für die konkrete Instanz. Abbildung 2.2 zeigt diese Entitäten und ihre Beziehungen zueinander in einem Enitity-Relationship-Diagram unter Verwendung der Min-Max-Notation. Dabei werden nur Attribute dargestellt, die für den wesentlichen Einsatz dieser Entitäten am wichtigsten sind. In gleicher Weise, wie im Folgenden beispielhaft die Sensoranbindung an die Plattform erklärt wird, kann analog mit Aktuatoren verfahren werden.

Bevor ein Sensor angelegt werden kann, müssen zunächst Instanzen der Entitäten Operator und Device existieren. Ein Device ist die virtuelle Repräsentation eines physischen Gerätes, das mittels SSH an die MBP angebunden werden kann. Dazu werden die IP-Adresse und die SSH-Zugangsdaten in der Device-Entität hinterlegt. Zugangsdaten können dabei sowohl ein Nutzername und ein Passwort sein, als auch ein SSH-Schlüsselpaar. Für letzteres steht in der MBP eine separate Entität zur Verfügung, die bei Bedarf erstellt und einem Device zugeordnet werden kann. Der SSH-Anschluss ist eine zwingende Voraussetzung, weshalb nur solche Geräte angeschlossen werden können, deren Betriebssystem entsprechende Clients unterstützen. Typischerweise läuft deshalb auf den angebundenen Geräten ein unix-basiertes Betriebssystem. Einem Gerät können mehrere Sensoren zugeordnet sein, wobei die konkrete Sensoranbindung am physischen Gerät in Gänze dem Nutzer der Plattform überlassen ist.

Zur Steuerung des Lebenszyklus von angebundenen Sensoren und Aktuatoren wird eine Operator-Instanz definiert, die jeweils alle Softwareskripte umfasst, die für die Plattformanbindung benötigt werden. Die MBP erwartet dabei, dass zwingend folgende Shell-Skripte mit angegebener Benennung existieren:

1. **install.sh:** Skript zur Installation etwaiger benötigter Drittprogramme, die beispielsweise mittels Paket-Manager automatisiert auf dem Betriebssystem des Devices installiert werden können.
2. **start.sh:** Skript zum Starten von Software, die die eigentliche Logik der Sensor- oder Aktuatoranbindung an das Gerät enthält. Da diese sich im Falle von Sensoren primär mit der Extraktion von Sensordaten beschäftigt, wird sie im Folgenden als Extraktions-Software bezeichnet. Sie muss außerdem die MQTT-Kommunikation mit der MBP implementieren, wobei Verbindungsinformationen, wie die IP-Adresse des Message Brokers, in einer, von der MBP automatisch generierten, Konfigurationsdatei zur Verfügung gestellt werden.
3. **running.sh:** Skript, das einen `True`-Wert zurückgibt, solange die Extraktions-Software aktiv ist.
4. **stop.sh:** Skript zum Stoppen der Extraktions-Software.

Zusätzlich können beliebige weitere Dateien einem Operator hinzugefügt werden, die auf dem angebundenen Gerät mittels SSH in gleicher Weise platziert werden. So kann beispielsweise auch die Extraktions-Software direkt von der MBP verwaltet und bereitgestellt werden. Neben Softwareskripten zur Anbindung, ist es auch möglich Parameter zu spezifizieren, die an das Gerät gesendet werden müssen, wenn die Extraktions-Software gestartet wird. Diese Parameter sollen dem Nutzer die Möglichkeit bieten nachträgliche Konfigurationseinstellungen für seine angebundenen Geräte zu treffen, ohne neue Operator- und Sensorentitäten anlegen zu müssen. Zuletzt kann für einen Operator auch noch eine Einheit spezifiziert werden, die als zusätzliche semantische Information, über die von der Extraktions-Software gesendeten Zahlenwerte, dient.

Nachdem jeweils eine Instanz von Operator und Device angelegt wurden, kann mit der Erstellung der eigentlichen Sensor-Entität fortgefahren werden. Für sie wird jeweils ein Operator und ein Gerät spezifiziert sowie zusätzlich ein Sensortyp angegeben. Der Sensortyp hat derzeit in der MBP ausschließlich visuelle Auswirkungen, etwa bei der symbolischen Anzeige im Modellierungstool für IoT-Umgebungen. Wie aus dem Entity-Relationship-Diagramm 2.2 entnommen werden kann, ist jedem Sensor oder Aktuator genau ein Operator und Gerät (Device) zugeordnet. Ein Device kann so wie ein Operator jedoch mehreren Sensoren, beziehungsweise Aktuatoren, zugeordnet sein. Dies ermöglicht die Wiederverwendbarkeit von Operatoren für anzubindende Komponenten, die die gleichen Operatorskripte verwenden sollen.

Ein Spezialfall eines Operators, der nicht in Abbildung 2.2 aufgeführt wird, ist die Entität des Monitoring Operators. Das sind Operatoren, deren Skripte direkt, ohne den Zwischenschritt einer Sensorerstellung, auf Geräten platziert werden. Dadurch ist es möglich Gerätedaten in der MBP ohne Erstellung einer Sensorentität zu berücksichtigen. Diese Funktion soll vor allem der Überwachung von Geräteeigenschaften dienen. Ein Beispiel hierfür ist das Auslesen der CPU-Temperatur eines Gerätes. Zusätzlich zu den bereits beschriebenen Operatorangaben wird bei der Erstellung von Monitoring Operatoren eine Liste von Gerätetypen angegeben. Sie gibt an, für welche Gerätetypen (zum Beispiel virtuelle Maschine und PC) der Monitoring Operator verwendbar ist.

Installation und Start der Operatorenskripte

Zur Installation und Steuerung der Operatorskripte, die im vorherigen Abschnitt mit der jeweiligen Sensor- oder Aktuatorentität verknüpft wurden, bietet die Benutzeroberfläche der MBP ein Panel in der Detailansicht für Sensoren, beziehungsweise Aktuatoren, an. Alle ausgelösten Aktionen in diesem Panel werden mittels SSH auf dem jeweiligen verbundenen Gerät durchgeführt. So gibt es für jedes Shell-Skrit, mit Ausnhame von `running.sh` einen Button, der die jeweilige Aktion für den Sensor ausführt.

Wurden für den Operator zuvor Parameter spezifiziert, so müssen diese in die vorgesehenen Eingabefelder des Panels vor dem Start der Operatorskripte eingegeben werden.

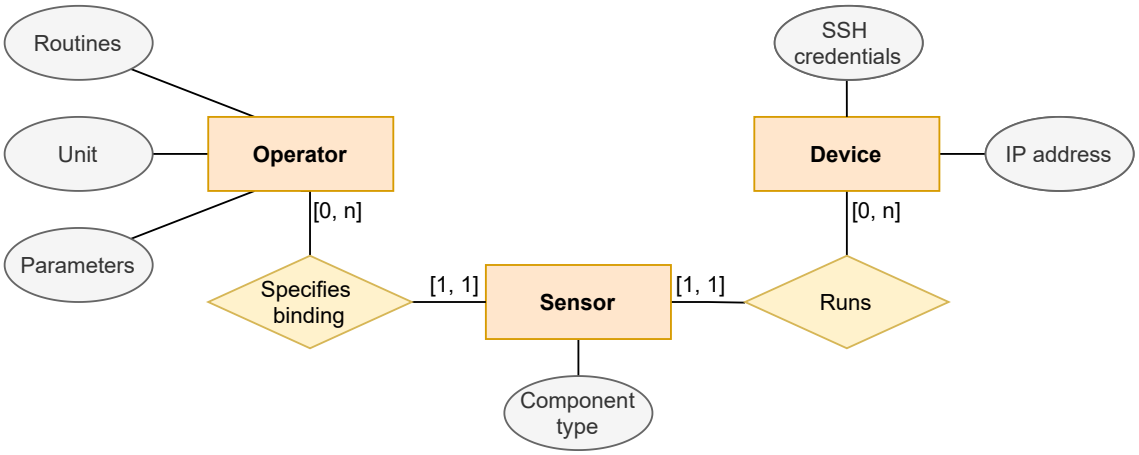

Abbildung 2.2.: Entity-Relationship-Diagramm mit den MBP-Entitäten die zur Geräteanbindung angelegt werden müssen. Es werden nur Attribute abgebildet, die für die grundlegende Funktion dieser Entitäten entscheidend sind. Die Kardinalitäten sind in Min-Max-Notation angegeben.

Umgang mit empfangenen MQTT-Nachrichten

Sobald Operatorskripte von IoT-Objekten gestartet wurden, die Telemetriedaten an die MBP senden sollen, werden auf den jeweiligen Geräten mithilfe der Extraktions-Software Sensordaten extrahiert und an den Message Broker geschickt. Der Aufbau der Nutzdaten, die so unter Verwendung des MQTT-Protokolls versendet werden, muss dabei einem festen Schema folgen. Nur so können sie von der MBP korrekt gelesen und interpretiert werden. Listing 2.1 zeigt ein Beispiel für eine korrekt aufgebaute und formatierte Nachricht, die von einem Sensor an die MBP gesendet wird. Es muss sich um ein JSON-Objekt mit folgenden Schlüssel-Werte-Paaren handeln:

1. **component:** String, der den Komponententyp in Großbuchstaben angibt. Die MBP kennt die Typen *Device*, *Sensor*, *Actuator* und *Monitoring*.
2. **id:** String, der die eindeutige Identifikationszeichenkette der sendenden Komponente angibt. Dabei handelt es sich um die Objekt-ID der virtuellen Entität in der Datenbank, die die physische Komponente in der MBP abbildet.
3. **value:** Sensorwert vom JSON-Typ *Number*, der eine Ganzzahl oder Dezimalzahl sein kann.

Da die JSON-Spezifikation Felder eines Objektes als reihenfolgenlos festlegt [Bra+14], wird ihre Anordnung von der MBP ebenfalls nicht vorgeschrieben.

Die Anwendungslogik der MBP enthält einen Service, der alle Belange der MQTT-Kommunikation mit dem Message Broker verwaltet. Beim Serverstart abonniert dieser Service alle MQTT-Topics, die als Präfix einen der vordefinierten Komponententypen (siehe erster Punkt in obiger Liste), gefolgt von einem Schrägstrich, haben. Dadurch erhält

Listing 2.1 Beispiel einer validen MQTT-Nachricht von einem Sensor an die MBP

```
{
  "component": "SENSOR",
  "id": "600db8c76dc71450f89a3927",
  "value": 14.25
}
```

er alle Nachrichten, die unter Verwendung eines solches Topic von den IoT-Objekten an den Message Broker gesendet werden. Zur internen Weiterverarbeitung der Nachricht in der MBP steht ein Service namens *ValueLogReceiverArrivalHandler* zur Verfügung. Dieser registriert einen Callback beim MQTTService und wird in dieser Weise immer dann aufgerufen, sobald neue Nachrichten an die vorgegebenen Topics gesendet werden. Der Service wandelt die ankommenden MQTT-Nutzdaten, die zu diesem Zeitpunkt noch als JSON-Objekt vorliegen, in ein internes Speicherobjekt der Java-Klasse *ValueLog* um. Dieses Objekt ermöglicht anderen Services der Anwendungslogik einen einheitlichen und einfachen Zugriff auf Sensordaten, ohne dass jedes mal erneut die Rohdaten in Form des JSON-Objekt geparst werden müssen. Sollte bei dem Parsing-Vorgang ein Fehler auftreten, etwa in einem Fall falsch formatierter Nachrichten, so wird eine Java-Exception geworfen, die zur Folge hat, dass die Nachricht verworfen wird.

Abbildung 2.3 zeigt ein Unified Modeling Language (UML)-Diagramm der ValueLog-Klasse. Die Felder `value`, `component` und `idref` sind die analogen Repräsentationen der JSON-Felder aus den rohen Nutzdaten (siehe Listing 2.1). Das Feld `message` speichert den kompletten orginalen JSON-String. Darüber hinaus gibt es Felder, die aus den sonstigen MQTT-Protkoll-Daten gewonnen werden. Dazu gehört die Angabe des Topics, an welches die Nachricht gerichtet war und eine Ganzzahl, die die verwendete Quality of Service (QoS)-Stufe kodiert. Null steht gemäß des MQTT-Standards für eine At-Most-Once-, Eins für At-Least-Once und Zwei für eine Exactly-Once-Semantik [OAS14]. Die MBP verwendet für die derzeit vorhandenen Operatorskripte das QoS-Level Null. Zuletzt enthält das ValueLog-Objekt noch ein Feld für einen Zeitpunkt in Form der Java-Klasse *Instant*, die in der Java Standard Edition 8 enthalten ist.Dieser Zeitpunkt wird vom *ValueLogReceiverArrivalHandler* immer auf die aktuelle Serverzeit gesetzt, wodurch ein ValueLog immer den Zeitstempel trägt, mit dem Zeitpunkt an dem die Umwandlung von einem JSON-Objekt zu dem ValueLog-Objekt erfolgt ist.

Innerhalb der MBP gibt es eine Reihe von Services, die über neu ankommende Telemetriedaten von IoT-Objekten informiert werden müssen. Zu diesem Zweck wird das objektorientierte Design-Pattern des Observers implementiert. Observer agieren dabei als Beobachterklassen, die sich bei einer Subject-Klasse registrieren und immer dann vom Subject benachrichtigt werden, wenn der zu beobachtende Systemstatus erreicht ist, der eine Zustandssynchronisation erfordert [GHJV94]. Das Subject ist, angewendet auf die ValueLog-Verteilung in der MBP der Service *ValueLogReceiverArrivalHandler*, bei dem sich andere Services registrieren können. Immer dann, wenn die Nutzdaten einer neuen

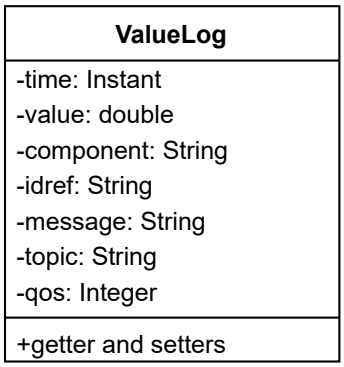

Abbildung 2.3.: UML-Diagramm der ValueLog-Klasse, die zur internen Repräsentation von Telemetriedaten der angebundenen IoT-Objekte dient. Ein ValueLog-Objekt enthält die Daten, die zu einem Zeitpunkt innerhalb einer MQTT-Nachricht an die MBP gesendet wurde.

MQTT, vom Subject in ein ValueLog-Objekt umgewandelt wird, benachrichtigt es alle Observer, die sich zuvor bei ihm registriert haben. Abbildung 2.4 veranschaulicht dieses Vorgehen. Nach der Umwandlung der Telemetriedaten zu ValueLogs werden diese an vier unterschiedliche Services weitergeleitet, die in Folge unterschiedliche funktionsspezifische Aktionen ausführen. Eine genauere Beschreibung dieser Observer-Services folgt in den nächsten Abschnitten.

Persistieren von Zeitreihendaten in der MongoDB

Der in Abbildung 2.4 von links als drittes abgebildete Observer-Service, der *ValueLogWriter*, ist dafür zuständig empfangene Telemetriedaten der angebundenen IoT-Objekte in der MongoDB-Datenbank persistent abzuspeichern. Für die Speicherung von ValueLogs aller angebundenen IoT-Objekte existiert eine gemeinsame Dokumenten-Collection in der Datenbank. Das bringt die Notwendigkeit mit sich, dass ValueLogs eindeutig zu einem IoT-Objekt zuordenbar sind, was durch das `idref`-Feld der ValueLog-Klasse umgesetzt wird. Der offizielle Java-Treiber für die MongoDB implementiert ein Mapping von POJOs zu einem Java-Zwischenspeicherobjekt, Objekte von sogenannten *Document*-Klassen, die dann direkt über den Treiber in die angebundene MongoDB geschrieben werden können. Dieses Feature wird auch zur Speicherung von ValueLogs in der MBP verwendet, was bedeutet, dass ValueLogs ganz analog zu ihrer Klassendefinition (siehe Abbildung 2.3) in der Datenbank als Dokument gespeichert werden können.

Offen bleibt jedoch die Frage, in welchen Gesamtstrukturen ValueLogs am sinnvollsten in der Datenbank-Collection gespeichert werden sollen. Eine naive Herangehensweise wäre, für alle ValueLogs ein separates Dokument in der Collection anzulegen, was einem ähnlichen Vorgehen entspräche, wie bei relationalen Datenbanken neue Zeilen für neue Dateneinträge anzulegen. Das bringt für die MongoDB allerdings vor allem drei wesentliche Nachteile mit sich, die Walters in einem MongoDB-Blogbeitrag vorstellt [Wal19]. Einzelne Dokumente

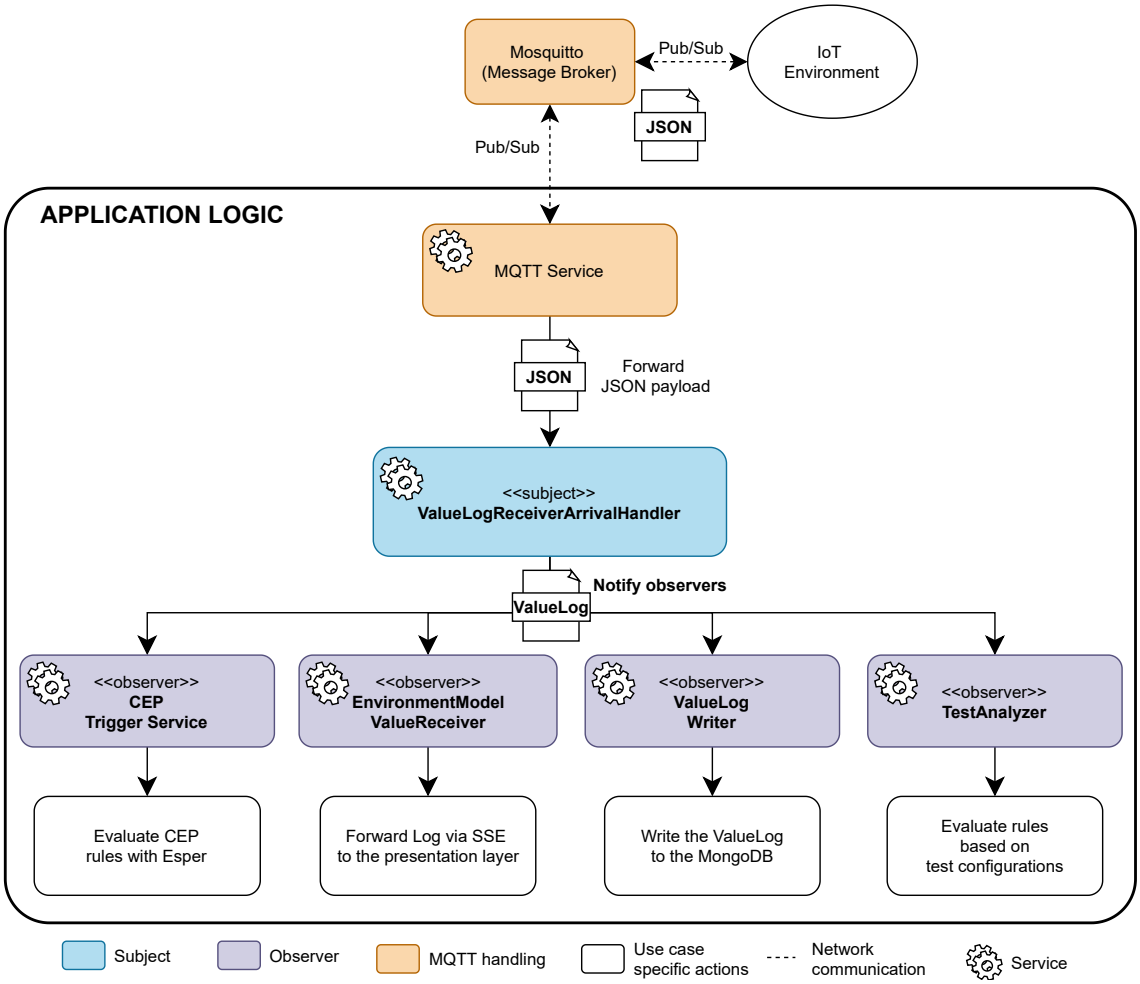

Abbildung 2.4.: Weg der Telemetriedaten von den IoT-Objekten innerhalb der verwalteten IoT-Umgebung bis hin zu spezialisierten anwendungsbezogenen Services der Anwendungslogik. Die zunächst als JSON-Objekt vorliegenden Nutzdaten werden in ein internes ValueLog-Speicherobjekt umgewandelt und in dieser Form nach dem Prinzip des Observer-Patterns an verschiedene Services weitergeleitet.

verursachen in der MongoDB einen höheren atomaren Speicherbedarf als ein Dokument, das mehrere eingebettete Dokumente enthält. Für wenige Datenbank-Dokumente fällt diese Diskrepanz weniger ins Gewicht, wird aber bei Zeitreihendaten, die potenziell in hoher Frequenz abgespeichert werden müssen und somit eine große Dokumentenanzahl verursachen, relevant. Des Weiteren erhöht sich auch der Arbeitsspeicherverbrauch, der durch das Index- und Dokumentencaching der Speicherengine, verursacht wird. Auch hier ist es speichereffizienter wenige eingebettete Dokumente zu speichern, als viele einzelne. Einfach nachvollziehbar ist dieses Phänomen anhand des Indexes, der über alle einzeln abgespeicherten Dokumente berechnet wird. Je mehr Dokumente, desto größer der Index, weshalb auch hier eingebettete Dokumente einen Speichervorteil mit sich bringen. Aus der

erhöhten Index- und Dokumentencachegröße resultiert direkt der dritte Nachteil, nämlich das Datenbankabfragen für sehr viele einzelne Dokumente inperformanter werden, eben da ein größerer Index und mehr einzelne Dokumente bei der Verarbeitung der Anfrage berücksichtigt werden müssen [Wal19].

Um diesen Nachteilen zu begegnen, schlagen die MongoDB-Entwickler als Best Practice für potenziell große Zeitreihendatenmengen das Entwurfsmuster des Bucket Patterns vor [Wal19]. Anstatt für jeden Datenpunkt einzelne Dokumente anzulegen werden, dem Pattern folgend, mehrere Dokumente gruppiert und in einem Dokument gemeinsam, als eingebettete Dokumente, gespeichert. Dies entspricht einem sinnbildlichen Eimer (Bucket), mit einer festen Kapazitätsgrenze, der dazu dient mehrere Dokumente zu sammeln. Tests von Walter zeigen, dass eine Dokumenteneinbettung mit jeweils 60 vernesteten Dokumenten pro tatsächlich gespeichertem Dokument erhebliche Fest- und Arbeitsspeicherentlastungen bewirken. Dies wurde für sekündlich neu hinzugefügte Daten über einen Zeitraum von 28 Tagen gezeigt. Für die Gruppierung von Dokumenten zu Buckets werden zwei Vorgehensweisen vorgeschlagen. Die erste Variante besteht darin die Bucket-Inhalte zeitbasiert festzulegen, indem etwa festgeschrieben wird, dass ein Eimer immer den Zeitraum eines bestimmten Zeitintervalls abdeckt. Die zweite Variante ist eine feste Kapazitätsobergrenze für jeden Bucket festzulegen, der dann zeitunabhängig solange aufgefüllt wird, bis er vollständig gefüllt ist und ein neuer Bucket angelegt werden muss [Wal19]. In der MBP wird die zuletzt genannte Variante für die Speicherung von ValueLogs umgesetzt.

Abbildung 2.5 zeigt schemenhaft die praktische Anwendung des Bucket Patterns auf die Collection zur Speicherung von ValueLogs in der MBP. Neben den ValueLogs, gespeichert als Array von Dokumenten, werden pro Bucket vier weitere Datenfelder gespeichert. `Idref` speichert die Identifikationszeichenkette einer IoT-Objekt-Entität, zu denen die einzelnen ValueLogs zuzuordnen sind. Das bedeutet, dass in gemeinsamen Dokumenten sinnvollerweise nur ValueLogs gespeichert werden, die von einem einzelnen IoT-Objekt an die MBP gesendet wurden. `First` und `Last` sind jeweils 64-Bit-Integer, die Zeitangaben in UNIX-Zeit, also in Sekunden seit dem 1. Januar 1970, speichern. `First` ist der Zeitstempel des ValueLogs mit dem zeitlich frühesten Zeitstempel und Last der mit dem spätesten Zeitstempel. Über diese Angaben ist es möglich schnell zeitliche Aussagen über die Gesamtheit der gespeicherten ValueLogs eines Buckets treffen zu können, ohne über alle ValueLogs iterieren zu müssen. Die genutzte Kapazität eines Buckets, also die Anzahl der eingebetteten ValueLog-Dokumente, wird in dem Feld `nvalues` gespeichert. Die maximale Kapazität wird von der MBP dabei mit 80 festgelegt. Sobald diese Grenze überschritten wird, wird ein neues Dokument zur Fortführung der Zeitreihe erstellt. `Idref`, `first` und `last` sind gute Kandidaten für Felder zur Indizierung der Collection, um die häufigsten Datenbankabfragen schneller beantworten zu können. Die MBP nutzt derzeit jedoch keinen benutzerdefinierten Index, sondern lediglich den Standardindex basierend auf der Objekt-ID der Dokumente.

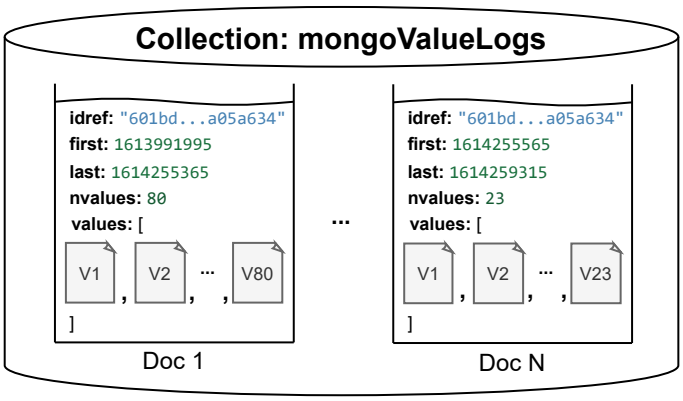

Abbildung 2.5.: Schemenhafte Darstellung der Anwendung des Bucket Patterns auf die MongoDB-Collection zur Speicherung von ValueLogs. Gespeicherte Dokumente, sinnbildlich Buckets, beinhalten bis zu 80 eingebettete Dokumente, die wiederum jeweils einen ValueLog repräsentieren.

Ein Nachteil der Anwendung des Bucket Patterns ist, dass der Zugriff auf ValueLogs komplexere Datenbankanfragen erfordert, um die Einbettung der ValueLogs richtig herzustellen und bei Abfragen diese wieder aufzulösen. In der MBP wird deshalb ein separater Service bereitgestellt, der sich speziell mit dieser Aufgabe beschäftigt. Dieser Service wird auch vom *ValueLogWriter-Observer* verwendet, um das korrekte Schreiben von ValueLogs, konform mit dem Bucket-Pattern, sicherzustellen. Der genaue Aufbau von hierfür notwendigen Datenbankabfragen und Sortieranweisungen ist letztlich für die Verarbeitung von komplexen IoT-Daten von keiner weiteren Relevanz, wie Kapitel 4.4 zeigen wird. Deshalb wird auf die konkrete Implementierung im Weiteren nicht eingegangen.

IoT-Regeln in der MBP

Der in Abbildung 2.4 als erstes von links abgebildete Observer-Service ist Teil der Anwendungslogik, die die Detektion und Verarbeitung von Events zur Realisierung von IoT-Regelungssystemen ermöglicht. Auf der Grundlage von Telemetriedaten angebundener Sensoren können in der MBP Bedingungen definiert werden, bei deren Eintritt benutzerdefinierte Aktionen ausgeführt werden. Solche Aktionen sind vor allem das Senden von Nachrichten an Aktuatoren, können aber auch automatische Anbindungen von weiteren IoT-Objekten sein. Die Implementierung dieses Regelungssystems stützt sich auf vorhandene Technologien des CEPs [SHS+20]. Dabei handelt es sich um Konzepte und Werkzeuge, die eine Detektion von Ereignissen basierend auf Daten aus mehreren Quellen ermöglichen. Insbesondere sind dabei Ereignisse inbegriffen, die sich ausschließlich durch die Kombinierung mehrerer Einzelereignisse erkennen lassen [Luc98] , was den Einsatz von CEP für verteilte Anwendungen, wie im IoT anzutreffen, qualifiziert.

Die MBP verwendet die Software Esper als Java-Laufzeitumgebung für das CEP basierend auf IoT-Daten. Die im Folgenden vorgestellten Esper-Features sind der Esper-Dokumentation der Version 5.4 [Esp16] zu entnehmen. Insbesondere wirbt Esper mit hoher Skalierbarkeit bezüglich der Anzahl an zu verarbeitender Nachrichten und kurzen Latenzzeiten, weshalb die Technologie zur Echtzeitdatenverarbeitung geeignet ist. Um zu definieren, welche Eventtypen verfügbar sind und welche Events unter welchen Bedingungen ausgelöst werden sollen, unterstützt Esper eine eigene Abfragesprache, die Event Processing Language (EPL). Diese ist angelehnt an die SQL wird jedoch um weitere Sprachelemente erweitert wie beispielsweise Befehle zur Berücksichtigung von bestimmten Zeitintervallen beim Auswerten der Eventdetektion. Eine detaillierte Sprachdefinition findet sich in [Esp16], im Folgenden werden jedoch nur solche Sprachelemente näher beleuchtet, die für die MBP im Zusammenhang mit der Verarbeitung von IoT-Daten relevant sind.

Die MBP registriert für jedes IoT-Objekt, das als digitale Repräsentation in der Plattform existiert (siehe auch Kapitel 2.1.2) einen Eventtyp bei der CEP-Engine. In Abbildung 2.7 sind das die jeweiligen Component Create Services, die über den *CEP Trigger Service* die Eventtypen bei Esper anmelden. Ein Eventtyp besteht im Wesentlichen aus einem Namen und Datenfeldern, die dazu dienen Eventdaten zu speichern. Im Fall der MBP sind das alle relevanten Telemetriedaten, die von einem IoT-Objekt an die Plattform gesendet werden, sowie der Zeitstempel, wann der Datenerhalt in der Anwendungslogik erfolgte. Die Erstellung diesen Eventtyps erfolgt mit Esper über ein EPL-Create-Schema-Statement. Listing 2.2 ist ein Beispiel für ein solches Statement, bei dem ein Event für ein Sensor der MBP angelegt wird. Da die MBP nicht in der Lage ist komplexe IoT-Daten zu berücksichtigen, wird hier nur ein Datenfeld für die zu erwartete Gleitkommazahl als einziges Telemetriedatum, neben dem Zeitstempel, angelegt. Der Name des Eventtyps besteht aus einer Komponentenbezeichnung als Präfix, hier im Beispiel `sensor`, gefolgt von der eindeutigen Objekt-ID der Komponente innerhalb der MBP.

Um basierend auf den registrierten Eventtypen in der Lage zu sein, tatsächliche Events zu detektieren, werden Esper EPL-Select-Statements übergeben. Diese geben jeweils an, welche Bedingungen erfüllt sein müssen, damit Events als solche berücksichtigt werden und zugehörige Callbacks zur Regelausführung in Folge ausgeführt werden. Ein Beispiel für ein solches Statement ist Listing 2.3. Das Statement wählt in diesem Beispiel alle Events aus, die die Bedingung des angegebenen Patterns sowie die des `WHERE`-Zweiges, erfüllen. Die Bedingung, formuliert im Pattern, ist, dass zunächst ein Event von einem ersten Sensor mit einem Sensorwert größer als 20 eingegangen sein muss und anschließend ein Event von einem zweiten Sensor. Dies muss jedoch in einem Zeitintervall von zwei Sekunden geschehen. Trifft das Pattern zu, so wird die `WHERE`-Bedingung ausgewertet, die zusätzlich fordert, dass die Sensorwerte des ersten Sensors kleiner-gleich 50 und die des zweiten Sensors genau 232 sind. Sollten alle diese Bedingungen erfüllt sein, führt die Esper-Engine einen Callback aus, der ihr zuvor gemeinsam mit dem EPL-Select-Statement übergeben wurde. Diese Prozedur kann auch aus Abbildung 2.7 entnommen werden. Über den Service Rule Engine wird dem *CEP Trigger Service* mitgeteilt, dass eine bestimmte Regel bei der Esper-Engine registriert werden soll. Dazu wird ein EPL-Select-Statement, so wie anhand des Beispiels vorgestellt, und ein Callback übergeben. Dieser Callback verweist

auf eine Methode des Rule Executors und wird ausgeführt, sobald die Esper Engine unter Anleitung des Select-Statements ein Event detektiert. Der Callback enthält insbesondere die vorspezifizierte Aktion, die ausgeführt werden muss, um die Regel, wie vom Nutzer definiert, korrekt zu berücksichtigen. Eine solche Aktion ist beispielsweise das Senden einer Nachricht an einen Aktuator.

Nach abgeschlossener Registrierung von Eventtypen und EPL-Abfragen, benötigt Esper zur Eventdetektion noch die atomaren IoT-Objekt-Events, für die zuvor jeweils ein Eventtyp registriert wurde. Ein solches Event muss immer dann berücksichtigt werden, wenn Telemetriedaten angebundener IoT-Objekte die Plattform erreichen. Deshalb agiert der *CEP Trigger Service* als Observer für ValueLogs, die vom *ValueLogReceiverArrivalHandler* bereitgestellt werden (siehe Abbildung 2.4). Die ValueLogs können von Esper in ihrer Java-Objektform nicht direkt verarbeitet werden, weshalb der *CEP Trigger Service* zunächst eine Umwandlung des ValueLogs in ein weiteres Zwischenformat durchführt, einem *CEPValueLogEvent*. Dieses Objekt fungiert als Wrapper für eine Java-Map mit Strings als Schlüssel und Java-Objekten als zugeordneter Wert. Die Namen der Schlüssel sowie die Typen der Wert-Objekte müssen dabei konform mit der Eventtyp-Definition sein, die mittels des EPL-Create-Statements bei Esper registriert wurden. Konkret ist das, in der derzeitigen Version der MBP, eine Map mit dem Eintrag für den Sensorwert und die Empfangszeit (siehe CEP Event in Abbildung 2.7). Die Map, gemeinsam mit dem Eventname, werden der Esper-Engine übergeben, die dann in Echtzeit überprüft, ob durch das eingetretene atomare IoT-Objekt-Event Regeln ausgeführt werden müssen, die zuvor über die *Rule Engine* registriert wurden.

Zur Anlegung von IoT-Regeln über die Benutzeroberfläche müssen jeweils getrennt Bedingungen und Aktionen angelegt werden. Diese können anschließend jeweils bei der Anlegung einer komletten Regel-Definition miteinander verbunden werden. Besonders relevant für die spätere Berücksichtigung komplexer IoT-Daten ist das Tool zur Erstellung von Bedingungen, zu sehen in Abbildung 2.6. Dabei handelt es sich um eine grafische Eingabemöglichkeit zur Erstellung von EPL-Select-Statements. Dies soll den Nutzern die Notwendigkeit ersparen, sich umfangreiches Wissen über EPL aneignen zu müssen, obgleich manuelle Anpassungen an den grafisch erstellten Statements in eine zweiten Schritt ebenfalls unterstützt werden. Zum Erstellen der Select-Statements müssen zunächst mittels Drag-And-Drop IoT-Objekte, deren Eventtyp berücksichtigt werden soll, in die obere Box hineingeschoben werden, die anschließend mit je einer der Operatoren `before`, `or` und `and` verknüpft werden. Anschließend kann für jedes IoT-Objekt eine Filter-Bedingung definiert werden, die angibt, welche Events des zugrundeliegenden Objekt-Eventtyps für eine weitere Evaluierung berücksichtigt werden sollen. Da die MBP ausschließlich ein Value-Feld für die Eventtypen unterstützt, nimmt auch das Bedingungs-Tool an, dass `.value` stets der abzufragende Schlüssel für Events darstellt. Die eigentlichen Bedingungen können dann im unteren Panel definiert werden. Dabei stehen sowohl einfache Bedingung über einzelne Sensorevents zur Verfügung als auch komplexere Bedingungen wie Aggregationen über mehrere Sensorevents. Beispielsweise kann eine Bedingung über ein gleitendes arithmetisches Mittel definiert werden, indem zusätzlich ein Zeitfenster definiert wird, innerhalb dessen das Mittel jeweils berechnet werden soll. Ist der Nutzer mit der erstellten

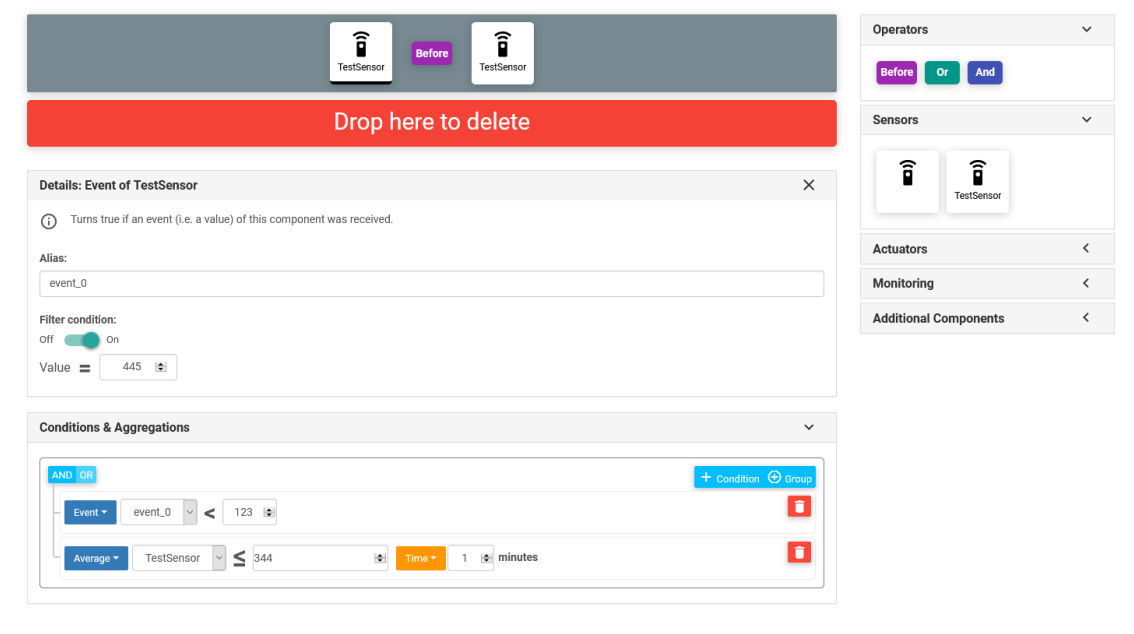

Abbildung 2.6.: Benutzeroberfläche zur Erstellung von CEP-Bedingungen. Zu sehen ist eine modellierte Regel bestehend aus einer Filterbedingung und zwei EPL-WHERE-Bedingungen unter Verwendung einer einzelnen Sensorevent-Bedingung und einer Aggregations-Bedingung.

Listing 2.2 Beispiel eines EPL-Statements zur Erstellung eines Eventtyps in der MBP.

```
CREATE SCHEMA sensor_6038e55b178a3c5f509e8127(time long, value double)
```

Konfiguration seiner Bedingung zufrieden, kann er mittels Navigation über einen Next-Button noch manuelle Anpassungen am generierten EPL-Statement vornehmen und die Erstellung anschließend bestätigen.

Berücksichtigung von IoT-Daten bei der Ausführung von Environment Models

Environment Models sind eine Möglichkeit in der MBP, wie IoT-Objekte an die Plattform angebunden werden können. Der Vorteil dieser Anbindungsmethode gegenüber dem normalen Vorgehen, bei dem Objekte nacheinander einzeln angebunden werden müssen,

Listing 2.3 Beispiel eines EPL-Statements zur Detektion von Events in der MBP.

```
SELECT * FROM pattern
  [every((event_0=sensor_6038e55b178a3c5f509e8127(value > 20) ->
  event_1=sensor_6038e15a178a3c5d509e2345 WHERE timer:within(2000)))]
WHERE ((event_0.value <= 50) AND (event_1.value = 232))
```

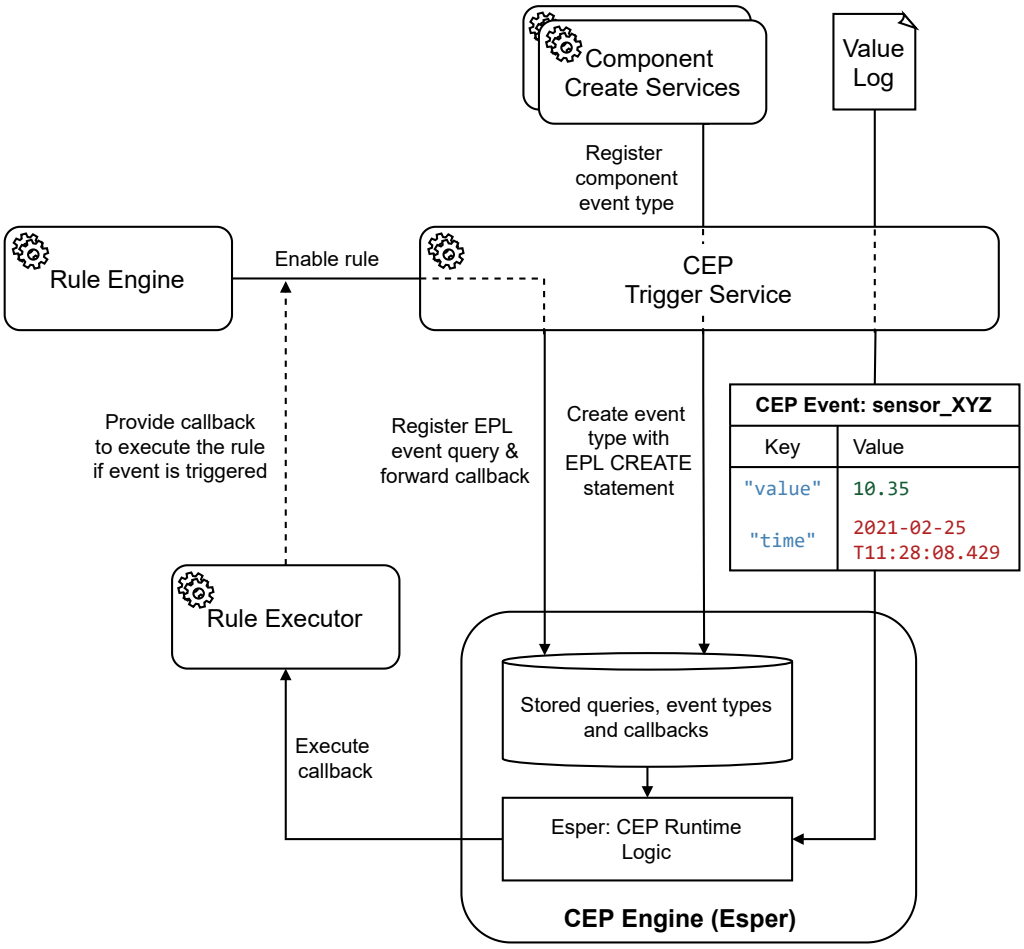

Abbildung 2.7.: Vereinfachte Darstellung der Abläufe innerhalb der MBP-Anwendungslogik zur Registrierung von Regeln bei der Esper-Engine und der anschließenden kontinuierlichen Eventdetektion.

ist, dass notwendige Operatorskripte gleichzeitig installiert und gestartet werden können. Sobald Telemetriedaten von Objekten empfangen werden, werden diese außerdem grafisch neben den modellierten Komponenten visualisiert.

Wie aus Abbildung 2.4 ersichtlich ist, ist der EnvironmentmodelValueReceiver ein Observer für ankommende ValueLogs und ist damit für die Verarbeitung von IoT-Daten relevant. Die Aufgabe dieses Service besteht darin, die Präsentationsschicht der Plattform mittels des Weiterleitens von erhaltenen Telemetriedaten zu benachrichtigen, damit diese die Daten geeignet für die jeweiligen Komponenten des Environment Models darstellen kann. Dazu wird der ValueLog zunächst in eine JSON-Repräsentation überführt und anschließend mittles Server-Sent Events (SSE) an das Frontend als Hypertext Markup Language (HTML) Document Object Model (DOM) Event übermittelt. Die SSE-API ist ein integrierter Bestandteil von HTML5 und ermöglicht es, abweichend vom sonst üblichen HTTP-Request-Answer-Paradigma, Daten auf Initiative des Servers über HTTP als Event-Stream zu einem Client zu senden [Hic09]. Abbildung 2.8 zeigt einen Ausschnitt der Benutzeroberfläche des

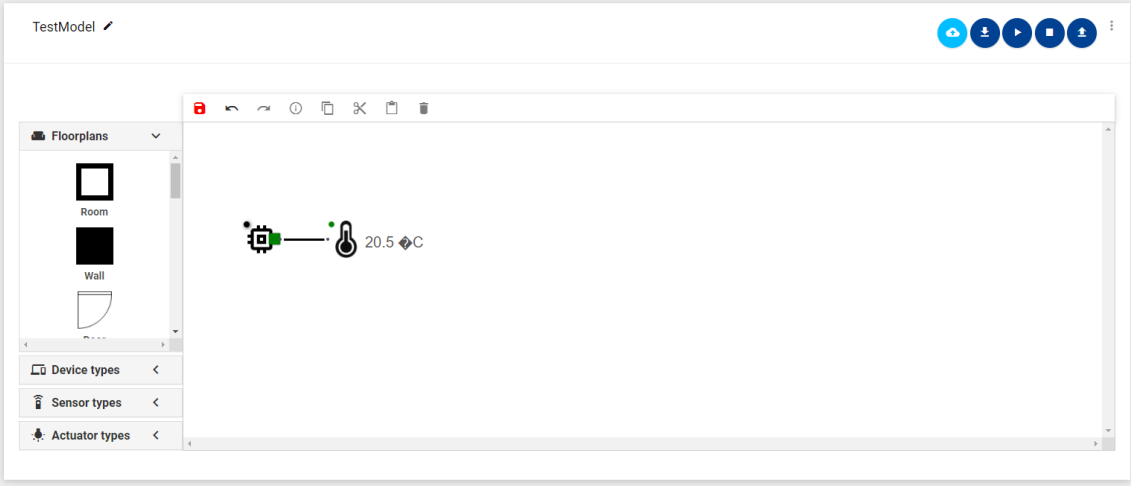

Abbildung 2.8.: Ausschnitt der Benutzeroberfläche des Environment-Model-Tools mit einer beispielhaften modelllierten IoT-Umgebung. Sensordaten werden neben den Sensor-Icons als kurzzeitig erscheinender Text angezeigt.

Environment Modelling Tools mit einer beispielhaften Modellierung einer IoT-Umgebung. Sensorwerte werden, so wie in der Abbildung, immer dann für wenige Sekunden neben den Komponenten angezeigt, wenn diese soeben von der Plattform empfangen wurden.

Verarbeitung von IoT-Daten im Testing Tool

Das Testing Tool ist eine Komponente der MBP, die ebenfalls ankommende Telemetriedaten als Observer abonniert. Mithilfe des Tools können vom Nutzer definierte Regeln gezielt getestet werden, um zu überprüfen, ob Regelsysteme der modellierten IoT-Umgebung sich wie gewünscht verhalten. Dabei kann sowohl auf bereits angebundene reale IoT-Objekte zurückgegriffen werden oder alternativ mit Sensor-Simulatoren gearbeitet werden. Die Abhängigkeiten des Testing Tools von der derzeitigen Verarbeitung einfacher IoT-Daten beinhalten unter anderem die Ausgabe von Sensordaten in Form von Diagrammen und Tabellen, sowohl in der Benutzeroberfläche als auch innerhalb von PDF-Testbericht-Dateien. Eine Anpassung für die Verarbeitung von komplexen Daten ist deshalb mit einem verhältnismäßig hohem Implementierungsaufwand verbunden. Aus diesem Grund wurde das Testing Tool für die Anpassungen der MBP im Rahmen dieser Arbeit nicht berücksichtigt, um den Entwicklungsfokus auf die primären MBP-Funktionalitäten richten zu können. Die im Hauptteil dieser Arbeit vorgestellten Konzepte lassen sich jedoch auch auf die notwendigen Anpassungen des Testing Tools anwenden.

Visualisierung von IoT-Daten

Zur Visualisierung von IoT-Daten angebundener IoT-Objekte bietet die MBP zwei verschiedene Frontend-Ansichten an. Eine erste wurde bereits im Abschnitt zu den Environment Models vorgestellt (siehe 2.1.2). Diese Art der Visualisierung ist jedoch eher als zweitrangiges Visualisierungswerkzeug, ergänzend zur Modellierung und Anbindung von IoT-Umgebungen zu sehen. Das primäre Visualisierungstool für IoT-Daten findet sich hingegen in den Detailansichten der einzelnen angebundenen IoT-Objekten.

Ein Beispiel für eine solche Detailansicht zeigt Abbildung 2.9 am Beispiel eines angebundenen Sensors. Oben rechts sind zunächst in einem Panel allgemeine Statistiken über alle jemals von diesem Sensor erhaltenen Daten aufgelistet. Dazu gehören allgemeine Daten wie die Anzahl an gemessenen Sensorwerten, aber auch verschiedene Lage- und Streuungsmaße wie das arithmetische Mittel und die Standardabweichung. Über dieses Panel lassen sich unter anderem auch kompatible Einheiten für die Statistiken umrechnen, sowie alle bestehenden Sensordaten löschen. Die eigentliche Darstellung der vorhandenen Datenpunkte geschieht mittels eines Liniendiagramms, mit dem jeweiligen Sensorwert auf der Ordinate und der Nummer dieses Sensorwerts auf der Abszisse. Die Empfangszeit eines Sensorwerts lässt sich anzeigen, indem mit der Maus auf einzelne Datenpunkte geklickt wird. Das Diagramm wird in zweierlei Ausführungen angeboten, jeweils in separaten Panels. Historische Sensordaten werden in einem Historical Chart angezeigt und muss manuell vom Benutzer via Button-Klick aktualisiert werden, sollten neue Sensorwerte hinzukommen. Es bietet des Weiteren die Möglichkeit die Anzahl der angezeigten Sensorwerte einzustellen. Das Live-Chart hingegen wird automatisch alle 30 Sekunden von der Plattform aktualisiert und zeigt jeweils nur die 20 jüngsten Sensorwerte. Beide Charts werden im Hintergrund mittels HTTP-GET-Requests mit Sensordaten versorgt. Diese werden vom Server als JSON-Objekt zur Verfügung gestellt, wobei sowohl die Daten für den Live- als auch für den Historical Chart stets aus der ValueLog-Collection der Datenbank entnommen werden. Dazu werden intern zunächst die relevaten Buckets der ValueLog-Collection herausgefiltert und diese anschließend mittels vom MongoDB-Java-Treiber angebotetenen POJO-Mappern in ValueLog-Objekte überführt. Unter Verwendung von Spring Boot lassen sich für diese Objekte dann eine REST-Schnittstelle definieren, wobei der Konvertierungsschritt von POJO zu JSON von den mit Spring Boot mitgelieferten Bibliotheken übernommen wird.

2.2. JsonPath

In diesem Kapitel soll eine kurze Einführung in die Abfragesprache JsonPath gegeben werden, da diese eine wichtige Rolle für die in Kapitel 4 vorgestellte Lösung zur Verarbeitung komplexer IoT-Daten spielt.

Die Idee für JsonPath und deren Definition stammt von Stefan Gössner, aus der Notwendigkeit heraus, einen einheitlichen Weg zu finden, wie auf Daten gespeichert in JSON-Objekten zugegriffen werden kann. Dabei ist es das Ziel einen möglichst ähnlichen Datenzugriff zu

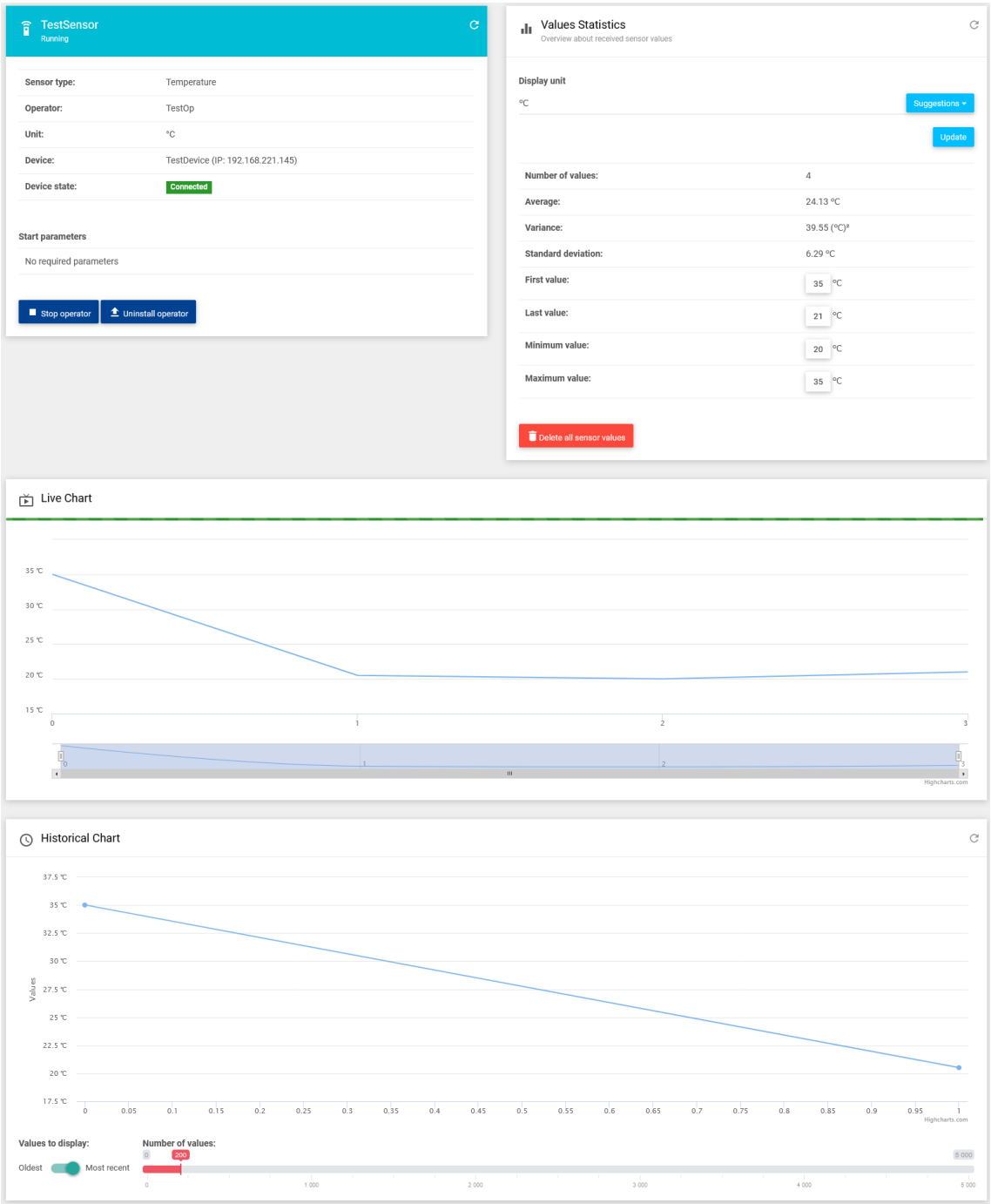

Abbildung 2.9.: Sensordetailansicht für einen an die MBP angebundenen Sensor. Die Operatorenskripte sind im Beispiel gerade aktiv, weshalb auch die Sensor-Live-Chart-Ansicht angezeigt wird.

ermöglichen wie es die XML Path Language (XPath) für XML-Dokumente ermöglicht. Die ursprüngliche Implementierung von JsonPath von Gössner ist für JavaScript und PHP geschrieben [Fri19]. Eine analoge Java-Implementierung, mit zusätzlicher Sprachdokumentation, auf die sich im Folgenden berufen wird, wird von der Firma Jayway als Open-Source-Projekt in GitHub bereitgestellt [Jay]. Die grundsätzliche Funktionsweise und die wichtigsten Sprachelemente, die innerhalb dieser Arbeit verwendet werden, werden anhand eines Beispiel-JSON-Objekts in Listing 2.4 vorgestellt.

Ein JsonPath-Statement beginnt immer mit einem `$`-Symbol zur Kennzeichnung des Wurzelelements eines JSON-Objekt-Pfads. Dies entspricht einem Zugriff auf die äußeren geschweiften Klammern eines JSON-Objekts. Alle Pfadelemente werden mit einem Punkt voneinander abgetrennt, wobei die Pfadelemente selbst jeweils in den meisten Fällen JSON-Schlüssel repräsentieren, die unmittelbar durch die Angabe ihres Namens berücksichtigt werden. Um im Beispiel den Namen des Thermometers abzufragen, lautet ein möglicher JsonPath: `$.thermometer.name`. Eine analoge Syntax ist die Verwendung von eckigen Klammern und Anführungszeichen, statt der Verwendung des Punkt-Operators für die Abfrage von Kindelementen: `$['thermometer']['name']`. Um auf Arrays zuzugreifen, wird nach dem JSON-Schlüssel-Namen in eckigen Klammern der jeweilige Index, auf den zugegriffen werden soll, angegeben. Sollen alle Indizes berücksichtigt werden, so kann dies entweder mittels Array-Slices, der Angabe von mehreren Indizes oder unter Verwendung des Wildcard-Operators `*` realisiert werden. So fragt `$['thermometer']['temperatures'][0]` nach dem ersten Objekt im Temperatures-Array, während `$['thermometer']['temperatures'][*]` alle im Array verschachtelten Objekte als Array ausgibt.

Darüber hinaus bietet JsonPath noch erweiterte Abfragefunktionalitäten, denen jedoch für die entwickelten Lösungen in dieser Arbeit keine Bedeutung zukommt. Dazu gehören beispielsweise Filteroperationen, die Abfragen basierend auf den konkreten Werten der zugeordneten JSON-Schlüssel realisieren. Auch Deep-Scan-Operatoren, oder bereitgestellte Funktionen im Rahmen von Filterprädikaten (zum Beispiel Summe, Minimum, Maximum, arithmetisches Mittel) werden an dieser Stelle nicht weiter behandelt.

Listing 2.4 Beispiel für ein JSON-Objekt auf dessen Inhalte im Folgenden mittels JsonPath zugegriffen werden soll.

```
  {
    "thermometer": {
      "name": "Messgeraet1",
      "temperatures":[
      {
        "val": 8.21,
        "unit": "°C"
      },
      {
        "val": 141.21,
        "unit": "°F"
      }
      ]
    }
}
```

3. Verwandte Arbeiten

Bevor sich mit der Findung einer geeigneten Lösung für die Plattform MBP beschäftigt wird, soll zunächst ein Überblick geschaffen werden, wie mit komplexen IoT-Daten in anderen IoT-Diensten umgegangen wird. Die im Folgenden präsentierten Rechercheergebnisse beschränken sich dabei auf eine Auswahl von Systemen, deren öffentlich zugänglichen Dokumentation ausreichend war, um Rückschlüsse auf die interne Funktionsweise, bezüglich der Verwaltung und Verarbeitung komplexer IoT-Daten, ziehen zu können.

3.1. IoT-Dienste gängiger Cloud-Computing-Plattformen

Cloud-Computing bezeichnet ein Modell, mit dem bei Bedarf jederzeit und von überall bequem über ein Netzwerk auf einen geteilten Pool von konfigurierbaren Rechenressourcen zugegriffen werden kann [MG11]. Unter Rechenressourcen werden dabei beispielsweise Netze, Server, Speicher, Rechenanwendungen und -dienste verstanden. Das US-amerikanische National Institute of Standards and Technology, das diese Definition in erweiterter Form herausgegeben hat, unterscheidet zwischen drei verschiedenen Dienstleistungsmodellen des Cloud Computings. Diese ordnen angebotene Dienste von Cloud-Computing-Systemen den Kategorien Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) und Infrastrucutre-as-a-Service (IaaS) zu.

Einige Cloud-Computing-Plattformen bieten auch für das IoT Dienstleistungen an. Im Rahmen dieser Arbeit wurden dazu die Cloud-Computing-Services von Google (Google Cloud Platform) [Goo21] und Microsoft (Microsoft Azure) [Mic21] betrachtet. Die Unterstützung von IoT-Geräten erfolgt bei beiden Systemen in erster Linie durch SaaS-Cloud-Services, die die Anbindung von Geräten über gängige Protokolle wie MQTT ermöglichen und Schnittstellen für den Anschluss anderer Cloud-Services bereitstellen. In Verbindung mit den zusätzlichen angebotenen Cloud-Services der Plattformen lassen sich aber auch die anderen Dienstleistungsmodelle des Cloud-Computings für Anwendungen im Kontext des IoT nutzen. Am Beispiel von Google wird ein Fallbeispiel deutlich, bei dem sämtliche Konfigurationsarbeit, bezogen auf den Umgang mit komplexen IoT-Daten, im Wesentlichen den Anwendern überlassen wird. Microsoft Azure hingegen bietet zusätzlich die Möglichkeit der Definition von *Digital Twins*, die dem System unter anderem die etwaigen zu erwarteten komplexen Datenstrukturen eingehender Telemetriedaten bekannt machen [Mic21].

Generell ist für beide Cloud-Plattformen anzumerken, dass ihre Zielgruppe vordergründig Unternehmen sind, die beispielsweise mittels IoT-Technologien ihre Produktionsanlagen im größerem Stil überwachen wollen. Dementsprechend bieten diese Plattformen keine vorgefertigten Ad-hoc-Lösungen, sondern nur die Mittel, um individuelle Lösungen selbst entwickeln zu können. Ohne informationstechnische Vorkentnisse ist es einem Nutzer kaum möglich, die notwendigen Konfigurationsarbeiten, die auch das Schreiben von individuellem Softwarecode beinhalten, zu erbringen. Deshalb ist anzunehmen, dass Lösungen für den Umgang mit komplexen IoT-Daten aus dem Bereich der Cloud-Computing-Services mit erhöhter Wahrscheinlichkeit nicht eins-zu-eins auf die MBP anwendbar sein werden.

3.1.1. Google Cloud Platform

Zur Unterstützung von IoT-Diensten enthält die Google Cloud Platform einen Service namens *Cloud IoT Core*, der eine bidirektionale Anbindung von IoT-Geräten an die Gesamtplattform ermöglicht [Goo21]. Der Nachrichtenaustausch erfolgt wahlweise mittels der Protokolle HTTP oder MQTT. Im Sinne der modularen Verknüpfbarkeit der Plattform-Dienste, ist dies im Wesentlichen auch die einzige Funktionalität, die vom *IoT Core* geboten wird. So gibt dieser Dienst beispielsweise nicht vor, wie gesendete oder zu sendende Nutzdaten von und an Geräte auszusehen haben oder was ihre erwarteten Inhalte sind. Eine solche syntaktische Auswertung und gegebenenfalls Interpretation der Daten erfolgt erst, wenn der Nutzer der Plattform den *IoT Core* mit zusätzlichen Google-Cloud-Diensten verbindet, um seine gewünschten Nutzungsziele zu erreichen. Eine Möglichkeit, um einen Anwendungsfall der Datenspeicherung mit anschließender Datenvisualisierung zu realisieren, wird in [Vie17] beschrieben. Dort wird der Dienst *Cloud Pub/Sub* dazu verwendet, Daten aus dem *IoT Core* eventbasiert an weitere Google-Cloud-Dienste weiterzuleiten. Empfangen von einer Firebase-Datenbank via Topic-Subscription, lassen sich die rohen Nutzdaten mittels *Firebase Cloud Functions* in Googles Big Query-Datenkbank schreiben. Dazu muss vorher ein vordefiniertes Datenbankschema in Big Query angelegt werden sowie Programmcode für die *Firebase Cloud Functions* geschrieben werden. Google Data Studio kann dann wiederum die Daten aus der Big Query-Datenbank lesen und verwenden, um sie geeignet zu visualisieren [Vie17].

Im Hinblick auf die Verwaltung von komplexen IoT-Daten wird deutlich, dass je nach Anwendungsfall unter Umständen für jedes neue Gerät erneut Entwicklungs- und Konfigurationsarbeit geleistet werden muss. So müsste man, wenn man weiterhin dem beschriebenem Beispielprozess der Sensoranbindung folgt, für jeden weiteren Sensor erneut manuell ein neues Datenbankschema anlegen. Auch müsste man für jeden Sensor zusätzlichen Programmcode schreiben, um die Daten mittels *Firebase-Functions* aus ihrer Rohform zu extrahieren und in die Big Query-Datenbank zu schreiben. Ein Entwickler von Google Cloud-Lösungen könnte diesen Vorgang vereinfachen, indem er beispielsweise generischen Code für die dynamische Anlegung von Datenbankschemata schreibt und integriert. All das bleibt jedoch dem Nutzer (hier der Entwickler) selbst überlassen.

3.1.2. Microsoft IoT Plug and Play

IoT Plug and Play ist ein IoT-Service, der Cloud-Computing-Plattform Microsoft Azure [Mic21]. Im Unterschied zu parallel existenten Azure-Diensten wie Azure IoT Central, die ebenfalls eine Anbindung von IoT-Geräten ermöglichen, soll IoT Plug and Play eine Lösung bieten, die mit einer weniger aufwendigen manuellen Konfigurationen durch den Nutzer auskommt. Um das zu realisieren, setzt der Dienst auf ein Konzept von digitalen Zwillingen (Digital Twins) [Mic21]. Ein solches Gerätemodell muss von einem Nutzer für jeden Gerätetyp in der Plattform erstellt werden. Anschließend kann das Gerät mit der Plattform mittels MQTT verbunden werden. Die Definition des *Digital Twins* erfolgt in der von Microsoft zu diesem Zweck entwickelten Definitionssprache Digital Twins Definition Language (DTDL). Sie ist unter einer Creative-Commons-Lizenz auf GitHub veröffentlicht und dokumentiert [Mic]. Alle Ausführungen dieses Kapitels berufen sich auf diese Dokumentation.

DTDL ist in JSON for Linking Data (JSON-LD)-konformer Syntax verfasst und besteht aus einer vorgegebenen Menge an Metamodell-Klassen. Die wichtigsten Unterklassen bilden dabei *Telemetry*, *Property*, *Command*, *Component* und *Relationship*. Da eine Erläuterung des vollen Umfangs von DTDL an dieser Stelle nicht sinnvoll wäre, wird im Folgenden nur flüchtig auf die wichtigste Klasse eingegangen, die letztlich für die Unterstützung komplexer IoT-Daten am bedeutsamsten ist. Dabei handelt es sich um die Klasse *Telemetry*.

Telemetry beschreibt, welche Daten von einem *Digital Twin* an die Plattform gesendet werden können. Die Struktur der Daten wird dabei durch die Definition eines Schemas, unter Verwendung einer Schema-Metaklasse angegeben. Ein valides Schema kann entweder primitv oder komplex sein. Primitive Schemas sind die Definition eines primitiven Datentyps, von denen eine vordefinierte Menge verfügbar ist. Dies sind *Boolean*, *Double*, *Float*, *Integer*, *Long*, *String* und vier weitere Typen, die unterschiedliche Datum- und Zeitformate repräsentieren. Komplexe Schemata modellieren mittels verschiedener Datenstrukturklassen zusammengesetzte, komplexe Datentypen. Dafür stehen die Typen *Array*, *Enum*, *Map* und *Object* zur Verfügung, die ihrerseits wiederum Elementklassen unterstützen, für die rekursiv erneut ein Schema festgelegt werden muss. Unter Verwendung dieser Metaklassen können so beliebig verschachtelte komplexe Datentypen definiert werden.

Die Metaklassen *Property* und *Telemetry* können ergänzend mit sogenannten *Semantic Types* annotiert werden. Dabei handelt es sich um eine Menge vordefinierter semantischer Datenbeschreibungen wie beispielsweise Beschleunigung, Energie, Kraft und Breitengrad. Sie enthalten eine Menge von Einheitentypen, die ihrerseits wiederum einer Menge vorgegebener Einheiten zugeordnet sind. Zum Beispiel ist der *Semantic Type* für Strom einem Einheitentyp namens CurrentUnit zugeordnet, der die Einheiten Ampere, Mikroampere und Milliampere enthält. Semantische Typen bieten dem IoT-Dienst oder anderen angebundenen Cloud-Services die Möglichkeit, automatisiert semantische Entscheidungen für eingehende Telemetriedaten zu treffen. Ob IoT Plug and Play davon tatsächlich Gebrauch macht, konnte aus der aktuellen Dokumentation derzeit nicht eindeutig entnommen werden. Semantische

Typen sind derzeit nur für Klassen verfügbar, deren angegebenen Einheiten mit den semantischen Typen kompatibel sind und deren Schema-Typ ein numerischer primitiver Datentyp ist.

3.2. Die Smart-Home-Plattform OpenHab

Ein wichtiger Anwendungsbereich des IoT sind Smart Homes. Während IoT-Lösungen bekannter Cloud-Computing-Anbieter hauptsächlich Unternehmen zu ihrem Nutzerkreis zählen, zielen Softwarelösungen für Smart Homes eher auf Privatanwender als ihre Zielgruppe ab. Dies erfordert einen erhöhten Fokus auf Nutzerfreundlichkeit und vorgefertigte Lösungen, etwa im Bereich von simplen Regelungssystemen. Denn von privaten Anwendern kann kein umfangreiches informationstechnisches Expertenwissen vorausgesetzt werden oder die Bereitschaft, sich dieses im Detail anzueignen.

Ein Beispiel für eine solche Plattform ist OpenHab. Sie bezeichnet sich selbst als Open-Source-Automatisierungssoftware für Smart Homes. Diese wird, ähnlich wie auch die MBP, auf einem Gerät im lokalen Netzwerk installiert und bietet eine browsergestützte Benutzeroberfläche. Alle folgenden Beschreibungen berufen sich auf die Dokumentation der Plattform in [Fou].

OpenHab unterscheidet zur internen Verwaltung angebundener Geräte vorrangig zwischen den zwei Konzepten *Things* und *Items*. *Things* repräsentieren physikalische Geräte, die direkt an die Plattform angebunden werden. Ein *Thing* kann dabei mehrere *Items* haben, die diesem *Thing* physikalisch zugeordnet sind. Ein Beispiel hierfür wäre ein Multisensor oder ein Computer, der Daten mehrerer Sensoren oder Geräte sammelt und mit OpenHab verbunden ist. Jedes *Thing* stellt sogenannte *Channels* zur Verfügung, die jeweils angeben, welche Funktionalitäten das *Thing* anbietet. Die *Channels* haben eine zentrale Bedeutung bei der Verwaltung und Verarbeitung komplexer IoT-Daten. Ein *Channel* definiert sich vor allem über die Art der Daten, die über ihn an das Thing, also das physikalische Gerät, gesendet werden kann, beziehungsweise von diesem Thing an OpenHab gesendet wird. Dabei wird jedoch nur ein primitiver Datentyp pro *Channel* unterstützt wie beispielsweise *String*, *Number*, *Color* und *Image*. Zu betonen ist hierbei jedoch, dass die Definition mehrerer *Channels* für ein *Thing* nicht zur Folge hat, dass pro *Channel* getrennte MQTT-Nachrichten ausgetauscht werden müssen. Ein *Channel* wird nur für die interne Verwaltung der Daten von OpenHab benötigt, um der Plattform bekannt zu machen, wie mit einem *Thing* kommuniziert werden kann. Mittels sogenannter Transformations-Pattern wird angegeben, wie die Werte aus den Rohdaten der angeschlossenen Geräte zu extrahieren ist. Unterstützt wird dazu insbesondere die Abfragen-Sprache JsonPath, die bereits in Kapitel 2.2 vorgestellt wurde. Durch das Konzept des *Channels* ist OpenHab immer bekannt, welche Daten für welche physischen Objekte (Things) zu verwalten sind.

Anders als *Things* repräsentieren *Items* nicht zwingend physikalische Objekte, sondern eher eine Funktionalität, die auf einer virtuellen Abstraktionsebene der Applikation zur Verfügung gestellt wird. Um für die Außenwelt letztlich doch eine Bedeutung zu haben, können Items mittels *Bindings* an *Channels* gebunden werden. Das hat zur Folge, dass wenn Werte in einem *Item* geändert werden, das mit einem Aktuator verbunden ist, dieser den neuen Wert über die *Channel*-Abstraktionsebene übermittelt bekommt. Andersherum aktualisieren sich die Werte von Items, die an einen *Channel* verbunden sind, der mit einem Sensor-Thing verbunden ist. Für ein Item stehen konzeptionell ähnliche primitive Datentypen wie für *Channels* zur Verfügung. Zusätzlich gibt es noch einen Typ *Group*, der es erlaubt andere Items zu speichern. Dies erlaubt eine rekursive Verschachtelung und Gruppierung von Items. Die Plattform empfiehlt solche Gruppierungen dann einzusetzen, wenn sinnvolle hierarchische Beziehungen wie Ortshierarchien (Stockwerk eines Hauses, Raum, ...) modelliert werden sollen. Es ist also eher nicht vorgesehen mittels *Groups* für ein *Item* komplexe Datenstrukturen festzulegen, auch wenn dies möglich erscheint.

Zusammenfassend unterstützt OpenHab komplexe IoT-Daten mittels unterschiedlicher vordefinierter Datentypen die an die Plattform gesendet werden können. Das Konzept von *Channels* macht der Plattform bekannt, welche dies für welche angebundenen Geräte sind. Jedoch werden zusammengesetzte Datentypen nativ nicht unterstützt. Es ist beispielsweise nicht möglich *Channels* zu definieren, die zusammengesetzte Datenstrukturen oder gar Array-Strukturen repräsentieren. Für Anwendungsfälle, bei denen solche zusammengesetzten Datentypen sinnvoll wären, muss man als OpenHab-Nutzer beispielsweise mehrere separate *Channels* und *Items* für jeden benötigten Sensorwert erstellen, der in der Plattform weiterverarbeitet werden soll. Gleichzeitig kann diese eingeschränkte Nutzbarkeit von zusammengesetzten Datentypen ein Vorteil für Nutzer sein, die diese zusätzliche Komplexität nicht benötigen und sich somit einfacher im Gesamtsystem zurechtfinden.

3.3. Abgrenzung

Die vorgestellten drei Beispiele zu vorhandenen Lösungen im Umgang mit komplexen IoT-Daten eignen sich nicht zur direkten Anwendung auf die MBP. Eine generische Lösung durch modular verknüpfbare Services, wie anhand der Google Cloud Platform vorgestellt, hätte zur Folge, dass alle notwendigen Konfigurationen zur Verwaltung komplexer Daten dem Nutzer überlassen sind. Das würde bedeuten, dass sich der MBP-Nutzer selbst um Datenbankanbindundungen und Datenzugriffe kümmern muss, also manuelle Entwicklungsarbeit leisten muss. Für eine Plattform wie die MBP, die keine losen und modular einsetzbaren Services unterstützt und gleichzeitig das benötigte Expertenwissen zur Verwendung der Plattform auf ein notwendiges Minimum reduzieren möchte, ist dies nicht praktikabel und erstrebenswert.

Ähnlich verhält es sich mit einer möglichen Anwendbarkeit der Lösung von Microsoft IoT Plug and Play. Mit DTDL wird von ihr eine mächtige Definitionssprache zur Modellierung von *Digital Twins* unterstützt. Unter anderem werden zahlreiche primitive und komplexe

Datenstrukturen zur Modellierung komplexer IoT-Daten angeboten. Nutzer müssen jedoch bereits mit dem Aufbau und der Funktionsweise dieser Sprache vertraut sein, um *Digital Twins* valide definieren zu können. Dazu gehört auch die zugrundeliegende Struktur von JSON-LD sowie weiterer verwendeter standardisierter Sprachelemente wie Internationalized Resource Identifiers (IRI)s [Mic]. Von den möglichen Schwierigkeiten im Kontext der Nutzerfreundlichkeit abgesehen, müssten bei einer Verwendung von DTDL auch ihre komplexeren Sprachelemente von der MBP unterstützt werden, was einen erheblichen Entwicklungsaufwand nach sich ziehen würde. So sind für einige Konzepte wie zum Beispiel Relationships zwischen mehreren *Digital Twins* derzeit keine Funktionalität in der MBP vorhanden.

Die Lösung zur Unterstützung komplexer IoT-Daten von OpenHab hingegen ist vergleichsweise minimalistisch gehalten und ließe sich deshalb eher auf die MBP übertragen. Das Konzept der *Channels*, die einzelne Datenströme zwischen Geräten und der Plattform modellieren, ist jedoch für die MBP eher ungeeignet, da das vorhandene MBP-Konzept der Operatoren bereits die Anbindung von Geräten an die Plattform abbildet. Dessen ungeachtet werden von *Channels* auch keine zusammengesetzten Datentypen unterstützt, was ebenfalls für eine andere Lösung für die MBP sprechen kann.

Zusammenfassend muss also im Rahmen dieser Arbeit ein neues Konzept zur Verwaltung komplexer Daten ausgearbeitet werden, das speziell auf die vorhandenen Gegebenheiten der MBP, wie in Kapitel 2 vorgestellt, angepasst ist. Auch soll so gewährleistet werden, dass ebenso die nicht-funktionalen Anforderungen der Plattform angemessen berücksichtigt werden können, wie eine möglichst hohe Nutzerfreundlichkeit. Eine detaillierte Beschreibung der Anforderungen an die zu findende Lösung folgt im Hauptteil dieser Arbeit.

4. Verarbeitung komplexer IoT-Daten in der MBP

Dieses Kapitel befasst sich mit den Änderungen, die an der MBP vorgenommen werden mussten, um die Verarbeitung komplexer IoT-Daten zu ermöglichen. Dabei werden die Änderungen in der Reihenfolge vorgestellt, wie sie auch sukzessive in die vorhandene Software integriert werden können und wurden. Zunächst werden jeweils die hinzugefügten allgemeinen Konzepte erläutert und gegebenenfalls alternative Lösungsmöglichkeiten diskutiert und gegeneinander abgewägt. Anschließend folgen konkreter Ausführungen zur praktischen Umsetzung dieser Konzepte. Dies beinhaltet relevante softwarearchitektonische Maßnahmen, verwendete Algorithmen und eingesetzte Technologien. Zuletzt werden die Nutzungsmöglichkeiten der neuen Funktionalitäten durch die MBP-Anwender dargestellt.

4.1. Entwurf und Integration eines Datenmodells für komplexe Sensordaten

Wie in Kapitel 2 bereits ausgeführt, verlassen sich die derzeitig vorhandenen Softwarelösungen innerhalb der MBP auf die Grundannahme, dass alle Nutzdaten angebundener IoT-Geräte, die mittels des MQTT-Protokolls an die Anwendungslogik der MBP gesendet werden, jeweils genau einen Zahlenwert enthalten. Dieser Zahlenwert wird fortführend als Gleitkommazahl des Java-Datentyps Double verwaltet und verarbeitet. Eine Verletzung dieser Annahme, die im Zuge der Zulassung komplexer IoT-Daten notwendig ist, führt dazu, dass die einzelnen Softwarekomponenten kein verlässliches Wissen mehr darüber besitzen, wie die gesendeten Daten der Geräte syntaktisch zu verarbeiten sowie gegebenenfalls zu interpretieren sind.

Um diesem Problem, speziell auf die MBP angepasst, entgegenzutreten, soll ein Konzept verfolgt werden, das in der Mächtigkeit, in der Definition von komplexen IoT-Daten, zwischen den Lösungen von IoT Plug and Play und OpenHab einsortierbar ist. Für an die MBP angebundene IoT-Geräte soll ein Datenmodell definiert werden können, das Informationen über die zu erwarteten zulässigen Telemetriedaten zur Verfügung stellt und dabei Antworten auf die folgenden zwei Fragen liefern soll:

1. Wie sind die zu erwarteten IoT-Objekt-Daten strukturell aufgebaut und syntaktisch formatiert?

2. Wie sind die Nutzdaten zu speichern, zu verarbeiten und gegebenenfalls semantisch zu interpretieren?

Diese zu lösenden Problemstellungen sind eine Teilmenge der Anforderungen, die im IoT durch das Konzept eines Digital Twins gelöst werden können. In dieser Terminologie gesprochen, kann man das einzuführende Datenmodell für IoT-Geräte als Erweiterung des bereits in der MBP in Grundzügen vorhandenen Digital-Twin-Konzeptes sehen. Wie die konkrete Integration des Datenmodells in das Digital-Twin-Konzept der MBP realisiert wird, ist Thema eines folgenden Unterkapitels. In den nächsten Unterkapiteln wird das Konzept der Datenmodelle für IoT-Geräte zunächst im Detail eingeführt. Allem voran werden Anforderungen an ein solches Datenmodell aufgestellt und anschließend Datenstrukturen in Erwägung gezogen, die diese erfüllen.

4.1.1. Anforderungen an das Datenmodell

Die Anforderungen an das Datenmodell, für die von der MBP verwalteten Digital Twins, ergeben sich primär aus den verschiedenen funktionalen Eigenschaften der Plattform, die Abhängigkeiten zu der Verwendung von Sensordaten besitzen und weiterhin unterstützt werden sollen. Zusätzlich handelt es sich sekundär um wünschenswerte Usability-Eigenschaften, die zwar zur rein funktionalen Unterstützung komplexer IoT-Daten nicht zwingend notwendig wären, jedoch vor allem im Hinblick auf die Nutzerfreundlichkeit sinnvoll erscheinen. Die Hauptanforderungen, die für den Rahmen dieser Arbeit festgelegt wurden, sind:

Vorgabe der syntaktischen Struktur von Telemetriedaten
: Sensordaten, die mittels MQTT an die Plattform gesendet werden, sollen unter Heranziehung des Datenmodells auf ihre syntaktische Validität überprüft werden können.

Unterstützung verschiedener primitiver Datentypen
: Das Datenmodell soll definieren, welche primitiven Datentypen welchen Datenfelder der zu erwarteten komplexen Telemetriedaten zugeordnet sind. Es muss möglich sein diese Datentypen sowohl analog in der der Java-Umgebung der Anwendungslogik als auch in der Datenbank repräsentieren zu können. Im Zuge der Sensorwert-Validierung sollen diese Datentypen ebenso berücksichtigt werden können.

Vorgabe der Struktur interner Speicherobjekte
: Telemetriedaten der Geräte müssen in allen Schichten der erweiterten Drei-Schichten-Architektur der MBP repräsentiert werden können. Aus dem Datenmodell soll hervorgehen, wie die Verschachtelung der jeweiligen notwendigen transienten und persistenten Speicherobjekte aussieht und wie auf einzelne Datenfelder oder zusammengesetzte Datentypen wieder zugegriffen werden kann.

Unterstützung zusammengesetzter Objekte
: Es soll möglich sein, einzelne Datenfelder eines primitiven Datentyps mittels Objektstrukturen zu gruppieren. Auf diese Weise ist es Nutzern möglich, semantisch

zusammengehörige Daten übersichtlich als zusammengesetzte Datentypen zu modellieren. Für die MBP bieten Objektstrukturen zusätzlich das Potenzial semantische Rückschlüsse auf die Daten zu ziehen. Denn so ist bekannt, welche Sensordaten eines Sensors wahrscheinlich einem gemeinsamen semantischen Kontext unterliegen. Ein Anwendungsfall hierfür wären beispielsweise GPS-Sensoren, die neben GPS-Koordinaten oft auch Daten wie die Höhenlage über der Normalnull erfassen. Der Längen- und Breitengrad ließe sich als separates Koordinatenobjekt neben der Höhenlage modellieren.

Unterstützung von Array-Datenstrukturen

Der Hauptvorteil von Array-Datenstrukturen besteht darin, dass bei einer großen Anzahl an benötigter Datenfeldern nicht jedes Datenfeld einzeln mehrfach für das Datenmodell modelliert werden muss, wenn der Datentyp der Array-Elemente im Voraus bekannt ist. Auch beim Erstellen der Operatorskripte oder dem Zugriff auf Arrays in der Anwendungslogik kann dies eine Erleichterung darstellen. Gleichförmige Datenobjektsammlungen sind unter Verwendung von Arrays als solche zu erkennen und einfach handzuhaben.

Berücksichtigung von Metadaten

Mittels des Datenmodells sollen potenziell beliebige Metadaten über die modellierten Datenstrukturen definiert und verwaltet werden können. Für die MBP wird zunächst und in erster Linie eine Angabe von Einheiten für die einzelnen Datenfelder eines Sensordatums benötigt. Die Metadaten müssen den betreffenden Datenfeldern eindeutig zuordenbar sein.

Einfache Verständlichkeit und Erstellbarkeit

Das Konzept des Datenmodells soll möglichst verständlich und intuitiv verwendbar sein. Benutzer der MBP sollen ohne umfangreiches Vorwissen in der Lage sein, Datenmodelle für ihre zu verwaltenden IoT-Geräte zu erstellen und richtig einzusetzen.

Repräsentierbarkeit komplexer IoT-Daten mittels JSON

Wie in Kapitel 2.1.2 beschrieben, tauschen die MBP und ihre angebundenen IoT-Geräte Daten in einem JSON-Format aus. Dieses Datenformat hat den Vorteil, dass es zum einen leicht für Menschen zu verstehen und zu lesen ist, aber auch von zahlreichen bereits vorhandenen Programmbibliotheken und Frameworks unterstützt wird. Dadurch wird auch die interne Handhabung, beispielsweise innerhalb der Anwendungslogik oder bei der Sensordatenextrahierung, durch die Operatorskripte, vereinfacht. Auch zwischen Anwendungslogik und Präsentationsschicht werden JSON-Objekte ausgetauscht. Das Datenmodell soll deshalb in der Lage sein JSON-formatierte Telemetriedaten abzubilden.

4.1.2. Definition von MBP-Datenmodellen

Zur Erfüllung der definierten Anforderungen bieten sich als zugrundeliegende Datenstruktur für Datenmodelle n-äre Bäume an, auch als Mehrwegbäume bezeichnet. Dabei handelt es sich um einen Baum, dessen Knoten eine beliebige Anzahl an Kindknoten besitzen darf. Der Vorteil dieser Datenstruktur zur Modellierung von Datenmodellen ist, dass sie in natürlicher Art und Weise zusammengesetzte Datentypen abbilden kann. Datenobjekte, die untergeordnete Daten enthalten, können so als Elternknoten dieser enthaltenen anderen Datenobjekte abgebildet werden. Hinzukommend eignen sich die unterschiedlichen Traversierungsverfahren für Bäume, um unter anderem Datenmodelle effizient und gezielt zu durchsuchen und sind je nach Anwendungsfall flexibel anzuwenden. Wie zum Beispiel in Kapitel 4.1.4 dargelegt wird, eignet sich eine Pre-Order-Traversierung durch den Baum für einen Datenvalidierungsalgorithmus.

Abbildung 4.1 zeigt ein Metamodell für Datenmodelle, wie es im konkreten Fall für den Einsatz in der MBP definiert wurde. Zunächst trägt das Datenmodell einen Namen und eine Beschreibung, um für den Nutzer leichter wieder identifizierbar zu sein. Es besteht aus einer Menge von Baumknoten, die entweder einen primitiven Typen, ein Objekt oder ein Array repräsentieren. Für jeden Knoten des Baumes wird ein eindeutiger Name definiert. Optional kann eine Beschreibung und ein String für Metadaten angegeben werden. Dieses Feld dient aktuell als konzeptioneller Platzhalter für etwaige Weiterentwicklungen der MBP, die zusätzlich für jeden Datenmodellknoten semantische Informationen benötigen sollten. Da die derzeitig einzig relevante semantische Information die Einheitsangabe von primitiven Typen ist, kommt diesem Feld im weiteren Verlauf dieser Arbeit zunächst keine nähere Bedeutung zu.

Handelt es sich bei dem Datenmodellbaumknoten um einen primitiven Typ, muss für ihn zwingend zusätzlich zu den anderen Knotenfeldern ein Datentyp definiert werden. Welche Datentypen dem Datenmodell zur Verfügung stehen, ist Thema des nächsten Unterkapitels. `Unit` ist ein optionales Einheitenfeld, das die Maßeinheit eines Sensorwertes spezifiziert.

Neben primitiven Typen stehen für das Datenmodell zwei strukturgebende, komplexe Datentypen zur Verfügung. Beide hierfür vorgesehene Baumknoten dürfen dabei keine Blätter des Baumes sein und dürfen ausschließlich als Eltern-Knoten weiterer Knoten fungieren. Das hat zur Folge, dass alle Blattknoten des Datenmodellbaums letztlich primitive Typen als Blätter haben. Im Unterschied zu Objekten müssen Arrays dabei genau ein Kindknoten haben und müssen zwingend eine positive ganzzahlige Größe spezifizieren.

Als zusätzliche Konvention, die nicht im Metamodell von Abbildung 4.1 abgebildet ist, soll gelten, dass jedes Datenmodell als Wurzel einen Knoten vom Typ *Object* hat. Das ermöglicht einen einheitlichen Zugriff auf das oberste Datenelement eines Datenmodells und erleichtert damit den später erforderlichen Zugriff auf konkrete Instanzen des Datenmodells. Wenn man von den unterschiedlichen primitiven Typen absieht, besteht damit das Datenmodell genau aus den Grundbestandteilen, aus denen auch die JSON aufgebaut ist. Damit ist auch die zuvor formulierte Anforderung der JSON-Abbildbarkeit erfüllt. Das gilt jedoch nur mit einer Einschränkung. Während Arrays in JSON beliebig aneinandergereihte Daten

unterschiedlicher Datentypen repräsentieren dürfen [Bra+14], sorgt die Einschränkung der Anzahl von Array-Kinder auf eins dafür, dass in JSON-Arrays, die von dem Datenmodell abgebildet werden, nur jeweils Werte des gleichen Typs gespeichert werden dürfen. Diese Entscheidung wurde getroffen, da es sonst deutlich komplexer werden würde, Array-Strukturen mittels des Datenmodells zu modellieren. Denn wenn im Voraus nicht bekannt ist, welche Datentypen die verschiedenen Elemente eines Arrays haben, müsste man jedes erlaubte Array-Element als einzelnen Knoten modellieren, was für große Arrays einen erheblichen Overhead erzeugt. Alternativ könnte man über zusätzliche Konventionen nachdenken wie beispielsweise, dass Kinder von Array-Knoten spezifizieren, welche Menge an Datentypen für das Array erlaubt sind. Das wiederum führt jedoch zu Uneindeutigkeiten in der Interpretation von interferierbaren Datentypen wie etwa Integer und Long. Nicht zuletzt entstünde so auch eine gewisse Redundanz zu dem Konzept der Objects, die ja genau für den Anwendungsfall zusammengesetzter, potenziell unterschiedlicher, Daten eingeführt wurden.

Eine weitere Designentscheidung ist, dass Arrays eine feste vordefinierte Größe haben müssen. Diese Einschränkung erleichtert den Zugriff auf komplexe IoT-Daten erheblich, wie in einem späteren Teil der Arbeit gezeigt wird. Das gilt sowohl für den Nutzer als auch für Implementierungen innerhalb der Anwendungslogik der MBP. Dessen ungeachtet können Anwender, die sich nicht ganz sicher sind, wie viele Sensordaten sie im Vorhinein zu erwarten haben, vorsichtshalber ausreichend große Zahlen als obere Grenze für die Arraygröße angeben. Sollten dann weniger Daten zu schicken sein, könnten die unbelegten Arrayplätze beispielsweise mit konventionell definierten Platzhalterwerten belegt werden, die einem Anwender leere Plätze signalisieren.

Zur Veranschaulichung zeigt Abbildung 4.2 ein Beispiel-Datenmodellbaum. Als Beispiel für einen primitiven Datentypen wurde hier zunächst der Datentyp Double als Repräsentation einer Gleitkommazahl gewählt. Auf die verschiedenen anderen verfügbaren Datentypen wird in einem Folgekapitel eingegangen.

Zuletzt soll für die Datenmodellbäume die Eigenschaft gelten, dass die Geschwisterrelation von Knoten eines gemeinsamen Elternknotens kommutativ ist. Das bedeutet, dass es keinen semantischen Unterschied macht, ob ein Kindknoten beispielsweise das linke oder rechte Kind eines Elternknotens ist. Diese zusätzliche Baumeigenschaft wird hauptsächlich aus zwei Gründen eingeführt. Erstens muss sich der Nutzer bei der Definition des Datenmodells so keine Gedanken darüber machen, in welcher Reihenfolge die Kindknoten angeordnet werden müssen und welchen etwaigen Einfluss dies auf die Semantik des Baumes und die spätere Verarbeitung der Daten hat. Zweitens sollen JSON-Objekte, bis auf die obig genannte Einschränkung, komplett durch das Datenmodell abbildbar sein. JSON ist jedoch ein reihenfolgenloses Format, wenn es um die Position von Geschwisterknoten innerhalb einer Vernestungsebene geht. Deshalb gibt es auch keinen wirklich sinnvollen Anwendungsfall für das Datenmodell, feste Reihenfolgen für Geschwisterknoten festzulegen. Für den in Abbildung 4.2 gezeigten Datenmodellbaum gibt es folglich zwölf semantisch äquivalente

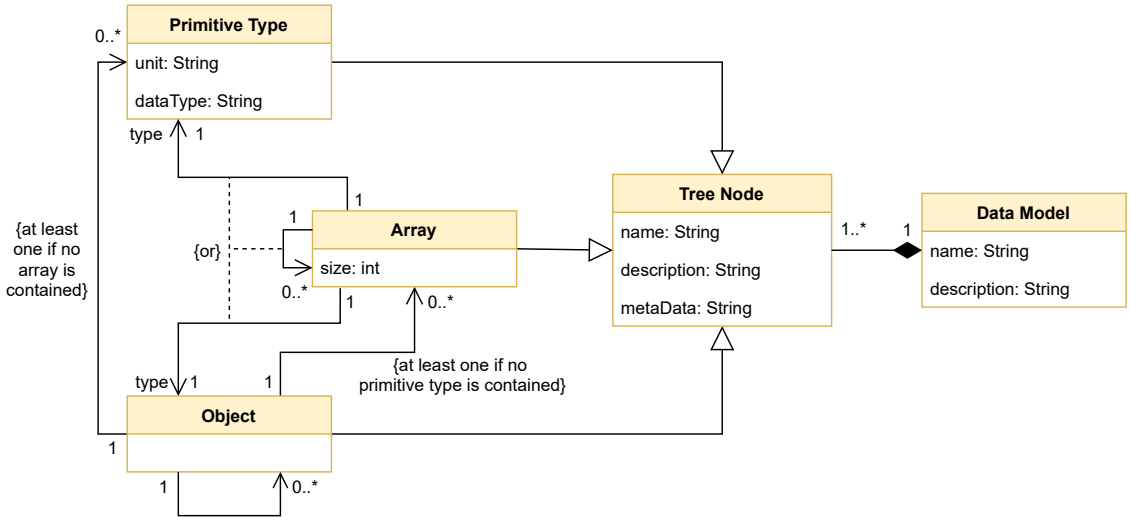

Abbildung 4.1.: Metamodell eines Datenmodells als UML-Klassendiagramm. Die Navigationsrichtung der Assoziationen spiegelt die Eltern-Kind-Beziehung der Objekte wider. Eltern sind die Klassen, von denen aus zu Kindern navigiert werden kann, was der natürlichen Navigationsrichtung der Baumstruktur entspricht.

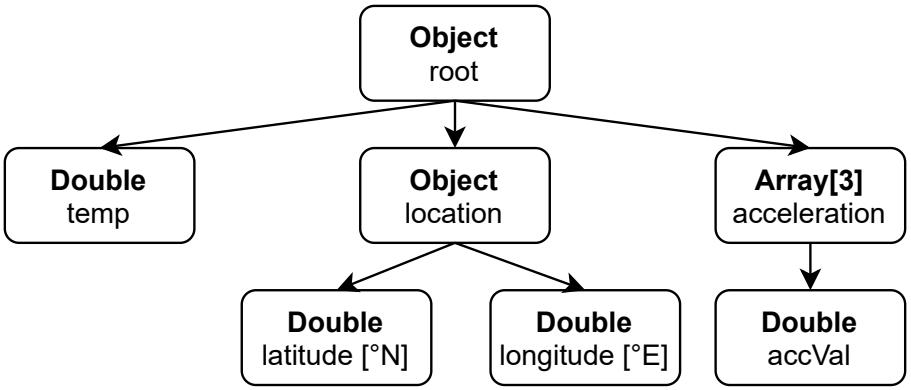

Abbildung 4.2.: Darstellung eines Beispiels für eine korrekte Instanz eines Datenmodell-Metamodells. Knotentypen sind in fett geschrieben, wobei alle Knoten, die kein Array oder Object sind automatisch primitiv sind. Array-Größen sind neben dem Array-Typ in eckigen Klammern dargestellt. Einheiten für primitive Typen sind ebenfalls in eckigen Klammern neben den Knotennamen angegeben.

Definitionsmöglichkeiten. Das ergibt sich aus den sechs möglichen Permutationen der Geschwisterknoten auf Baumlevel eins, multipliziert mit den zwei möglichen Permutationen der Kindknoten des Objekts `location`.

4.1.3. Festlegung verfügbarer primitiver Datentypen

Das Hauptkriterium zur Auswahl von primitiven Datentypen ist die jeweilige vorhandene Unterstützung dieser, in den verschiedenen genutzten Speicher- und Nachrichtenformaten der Telemetriedaten. Bei diesen Formaten handelt es sich zunächst um JSON-Objekte, die als Nutzdaten mittels MQTT die Anwendungslogik der MBP erreichen. Die Anwendungslogik wiederum muss in der Lage sein, die verschachtelten primitiven Datentypen, gemäß des Datenmodells, in primitive Java-Typen oder geeignete Java-Objekte umzuwandeln, um die interne Weiterverarbeitung von IoT-Daten zu ermöglichen. Zu dieser Weiterverarbeitung gehört insbesondere das Schreiben von Value-Logs (siehe Kapitel 2.1.2) in die MongoDB-Datenbank. Dieser Schritt erfordert eine explizite (durch den MBP-Entwickler) oder zumindest implizite (durch bereitgestellte Frameworks der MongoDB) Umwandlung der Datenstruktur in ein BSON-Objekt. Zur Auswahl der Menge unterstützter primitiver Datentypen muss deshalb eine geeignete Schnittmenge der in JSON, in Java und in BSON verfügbaren Datentypen gewählt werden oder gegebenenfalls Speicherkonventionen für nicht nativ unterstützte Datentypen festgelegt werden.

JSON unterstützt vorrangig die Datentypen *Number* für Zahlenwerte, *String* für Zeichenketten und *Boolean* für die boolschen Ausdrücke `true` und `false` [Bra+14]. Neben diesen, gibt es noch einen Datentyp für einen Nullwert [Bra+14], der jedoch für Telemetriedaten nicht sinnvoll einsetzbar ist, da er stets den Wert `null` annimmt.

Laut der Dokumentation der MongoDB unterstützt BSON 17 verschiedene Datentypen, die nicht als veraltet annotiert sind [Mon21] (Stand März 2021). Vier davon werden hauptsächlich von der MongoDB für interne Verwaltungsaufgaben verwendet. Dazu gehört beispielsweise die Object-ID zur eindeutigen Identifizierung von Dokumenten in der Datenbank und TimeStamp, als Zeitstempel, wann das Dokument zuletzt bearbeitet oder hinzugefügt wurde [Mon21]. Array und Object gelten im Kontext der Datenmodelle als nicht primitiv und Null wurde als Datentyp schon bei der Betrachtung der JSON-Datentypen ausgeschlossen. Somit bleiben zehn Datentypen in der engeren Auswahl: *Double*, *String*, *Binary data*, *Boolean*, *Date*, *Regular Expression*,*JavaScript*, *32-bit integer*, *64-bit integer* und *Decimal128*. Die Möglichkeit JavaScript-Code sowie reguläre Ausdrücke in der MongoDB zu speichern, hat für die Verwaltung und Verarbeitung von Telemetriedaten keinen offensichtlichen Anwendungsfall. Es erscheint unwahrscheinlich, dass etwa Sensoren Programmcode oder reguläre Ausdrücke als Daten ausgeben. Deshalb wurden diese beiden Datentypen ebenfalls aus der Auswahl entnommen. Somit bleiben insgesamt acht Typen übrig, die als primitive Typen für die Definition von Datenmodellen eingesetzt werden können. Offen bleibt jedoch die Frage, in welcher konkreten Form diese Datentypen in

der Anwendungslogik oder im JSON-Format repräsentiert werden können und wie die Transformation dieser Repräsentationen zwischen den verschieden Datenformaten im Detail durchgeführt wird.

Umgang mit numerischen, boolschen Typen und Strings

Die Handhabung der Überführung von numerischen, boolschen Typen und Strings in die verschiedenen Datenformate ist für alle verfügbaren Datentypen der einfachere Fall, da für diese Typen bereits vorhergesehene Repräsentationen in JSON existieren. Boolsche Werte werden in JSON und in BSON mittels der Literale `true` und `false` dargestellt [Bra+14; Mon21]. Diese können somit in eine Java-Boolean-Variable (ein Bit) überführt werden.

Strings werden in BSON im UTF-8 repräsentiert [Mon21]. Der JSON-Standard betont, dass der Text in JSON-Strings in UTF-8 kodiert werden muss, um über Netzwerkprotokolle korrekt versendet werden zu können [Bra+14]. Java hingegen verwendet für String-Objekte die UTF-16-Kodierung [Orab]. Die richtige Überführung dieser Formate muss folglich bei der Umwandlung von Strings gewährleistet sein. In der Regel berücksichtigen gängige JSON-Parser solche Kodierungsfragen für ihre jeweilige Plattform bereits, weshalb dies keiner manuellen Fallbehandlung durch MBP-Entwickler bedarf.

Die numerischen Typen sind unter allen drei Datenformaten kompatibel. *32-bit integer* entspricht dem gewöhnlichen Java-Integer und *64-bit integer* entspricht der Größe eines Longs. Auch der BSON-Double-Typ ist unmittelbar mittels Javas Double repräsentierbar. Eine Ausnahme bildet jedoch *Decimal128*, denn Java unterstützt keine primitiven Gleitkommazahlen größer als acht Bytes. Deshalb wird zur Repräsentation dieses Types jeweils ein Objekt der Java-Klasse *BigDecimal* verwendet.

Umgang mit Datums- und Zeitangaben

JSON unterstützt nativ keinen Datentypen für Datumsangaben, weshalb sich zwei sinnvolle Möglichkeiten eröffnen, wie Datumsangaben mittels JSON repräsentiert werden können. Die erste Möglichkeit besteht darin, die Datumsangabe als Zahl, also als JSON-Number-Typ, zu kodieren. Dafür bietet sich beispielsweise die POSIX-Zeit an, die die verstrichene Zeit in Sekunden seit dem ersten Januar 1970 angibt. Die zweite Möglichkeit ist, die Datum- und Zeitangaben als Zeichenkette in JSON-Strings zu speichern. Da die Verwendung von eigens definierten Datenmodellen zur Verarbeitung komplexer IoT-Daten es ermöglicht, selbst die Überführung von JSON in ein für die Anwendungslogik lesbares Format durchzuführen, gibt es keine Einschränkungen, die eine dieser beiden Möglichkeiten bevorteilen würde. Deshalb werden beide Repräsentationsmöglichkeiten für die MBP unterstützt.

Per Konvention müssen Ganzzahlen für Datums- und Zeitangaben der MBP in der POSIX-Zeit in Millisekunden angegeben werden, um korrekt interpretiert werden zu können. Angaben als Zeichenketten müssen einem der vordefinierten zulässigen Formaten folgen.

In Tabelle A.1 des Anhangs werden alle zulässigen Zeichenkettenformate sowie Beispiele für Datums- und Zeitangaben aufgeführt, die für die MBP festgelegt werden. Bei Bedarf sind diese Muster beliebig um zusätzliche erweiterbar.

Umgang mit Binärdaten

Zur Repräsentation von Binärdateien bietet BSON den Datentyp Binary data [Monb]. Für Java eignet sich die Verwendung von Byte-Arrays, da der Byte-Datentyp eine Größe von einem Byte hat und so beliebige, auch ungerade Byte-Anzahlen berücksichtigt werden können. Außerdem ermöglicht die Java-Bibliothek zur Anbindung an die MongoDB ein Mapping zwischen dem Java-Typ *byte[]* und dem Typ *Binary data* der Datenbank, beziehungsweise des BSON-Dokuments [Mona].

Offen bleibt die Frage, wie Binärdaten in JSON dargestellt werden sollen. Dabei hat eine Kodierung mittels Text in Strings den Vorteil, dass unter Verwendung eines größeren Zahlenalphabets Binärdaten kompakter repräsentiert werden können, als es beispielsweise bei reinen Zahlenwerten der Fall ist. Deshalb wird für die MBP festgelegt, dass Binärdaten als String in Base64-Kodierung vorliegen müssen, um korrekt von der Anwendungslogik und somit letztlich auch von der Datenbank interpretiert und gespeichert zu werden.

Ein Standard zur Spezifikation der Base64-Datenkodierung wird von der Internet Engineering Task Force (IETF) [Jos06] herausgegeben. Zur Kodierung der Binärdaten als Zeichenketten werden 65 Zeichen des American Standard Code for Information Interchange (ASCII)-Zeichensatzes verwendet. Das 65-igste Zeichen, ein Gleichheitszeichen, dient dabei als Auffüllzeichen, um stets die Teilbarkeit der Zeichenanzahl im kodierten Text durch vier zu gewährleisten. Diese Bedingung ist notwendig, da die Kodierung eine Gruppierung von immer 24-Bits vorschreibt, die wiederum als Gruppe von vier 6-Bit-Gruppen angesehen werden. Für jeder dieser 6-Bit-Gruppen steht dann genau ein Zeichen des Kodierungsalphabets zur Verfügung, denn bei der Verwendung von sechs Bits sind genau 64 unterschiedliche Bitkombinationen möglich.

Tabelle 4.1 zeigt zusammenfassend eine Übersicht über die, für MBP-Datenmodelle zur Verfügung stehenden, primitive Datentypen und deren unterschiedlichen Repräsentationen in den jeweiligen Datenformaten.

4.1.4. Definitionsformat und Validierung von Datenmodellbäumen

JSON-Spezifikation für Datenmodelle

Nachdem der Aufbau von Datenmodellen grundlegend definiert wurde, soll es nun näher um die konkrete Syntax gehen, die gewählt wurde, um Datenmodelle zu spezifizieren. Diese Repräsentation soll in erster Linie dazu verwendet werden, um als HTTP-Nutzlast an die REST-API gesendet werden zu können, sodass die Anwendungslogik sich, auf Grundlage

JSON		BSON		Java	
Name	**Größe [bit]**	**Name**	**Größe [bit]**	**Name**	**Größe [bit]**
Number	-	32-bit integer	32	int	32
		64-bit integer	64	long	64
		Double	64	double	64
		Decimal128	128	BigDecimal	-
Boolean	-	Boolean	-	boolean	1
String	-	String	-	String	-
		Date	-	Date	-
		Binary data	-	byte[]	8 * size

Tabelle 4.1.: Übersicht über die unterstützten Datentypen, die in der Terminologie der Datenmodelldefinition als primitive Typen gelten. Gegenübergestellt sind jeweils korrespondierende Typen sowie zeilenversetzt Typen, die mehreren anderen Typen eines anderen Speicherformat entsprechen können. Eine Ausnahme bildet dabei der Typ Date, der in JSON sowohl mittels String als auch mittels eines Wertes vom Typ Number repräsentiert werden kann. Dynamische Speichergrößen sind mit einem Bindestrich gekennzeichnet.

dieser Spezifikation, den Datenmodellbaum als Datenstruktur intern aufbauen kann. Da im Grunde alle Nutzdaten der MBP, die zwischen Präsentationsschicht und Anwendungslogik ausgetauscht werden, JSON-Objekte sind, wird hierfür ebenfalls ein JSON-Objekt definiert. Listing 4.1 zeigt die JSON-Spezifikation des Datenmodells-Beispiels aus Abbildung 4.2. Für jeden Datenmodellbaum wird ein JSON-Objekt innerhalb eines `treeNode`-Arrays angelegt, wobei die Reihenfolge der Objekte innerhalb des Arrays arbiträr gewählt sein darf. Die Felder der einzelnen Objekte sind im Wesentlichen analog zu denen aus der Definition des Metamodells in Abbildung 4.1. Das `type`-Feld gibt jeweils an, um was für einen Baumknotentyp es sich handelt, wobei der angegebenen String ein Element aus einer vordefinierten Menge von verfügbaren Datentyp-Strings sein muss, passend zu den Datentypen, wie sie im letzten Unterkapitel definiert wurden. Darüber hinaus sind `parent` und `children` Felder, die die strukturellen Beziehungen des einzelnen Baumknotens zu den anderen Knoten im Baum spezifizieren. Dabei dienen die Namen der Knoten als Objektreferenzen auf die anderen definierten Knoten, was die Konvention erfordert, dass Namen für Datenmodellknoten immer einzigartig vergeben sein müssen, um die Eindeutigkeit des Datenmodells zu bewahren. Diese zusätzliche erforderliche Datenmodelleigenschaft stellt dabei keine zu große Einschränkung für den Nutzer dar, da Duplikat-Schlüssel in JSON auf der gleichen Verschachtelungsebene ohnehin nicht zulässig sind [Bra+14].

Die Entscheidung Datenmodellbäume auf diese Weise zu definieren hat primär die Motivation, dass so, wenn die einzelnen Knoten einmal in Java-Objekte überführt wurden, sich der Baum durch eine einfache Iteration über alle Knoten zusammensetzen lässt. Gleichzeitig ist das Konzept des Datenmodellbaumes transparent repräsentiert und kann von einem Nutzer somit leicht nachvollzogen werden. Dennoch kann sich gefragt werden,

weshalb nicht etwa vorhandene Schema-Formate, wie JSON-Schema zum Einsatz kommen, das unter anderem gerade zu dem Zweck der Validierung von JSON-Objekten konzipiert wurde [PRS+16]. Der Vorteil des Einsatzes von JSON-Schema wäre zumindest, dass die Validierung von JSON-Objekten von externen Bibliotheken durchgeführt werden könnte und gleichzeitig eine dem Nutzer möglicherweise schon bekannte Syntax verwendet wird. Nachteile dieser Variante ist jedoch, dass die Möglichkeit bestehen müsste, JSON-Schema in einen Datenmodellbaum zu überführen. Dies ist jedoch nicht trivial, da JSON-Schema umfangreiche Sprachfeatures wie unter anderem rekursive Schema-Definitionen unterstützt, die in diesem Fall von einem Datenmodellbaum-Parser berücksichtigt werden müssten. Zwar könnte man den Funktionsumfang auf grundlegende Sprachelemente einschränken, dies müsste aber einem Nutzer gegenüber kommuniziert werden. Zweitens unterstützen gängige Bibliotheken JSON-Schema zwar, um JSON-Objekte zu validieren und gegebenenfalls anschließend in eine POJO-Repräsentation zu überführen, darüber hinaus gibt es aber keinen Vorteil für Anwendungen, die über diese syntaktische Auswertung hinaus gehen. Deshalb ist der Mehrwert einer Verwendung von JSON-Schema, als fertige Lösung zur Definition von Datenmodellen, eher ungeeignet für die Anwendung in der MBP.

Um in der Anwendungslogik die in JSON spezifizierten Datenmodelle in eine reale Baumdatenstruktur zu überführen, sind mehrere Schritte notwendig, die unter anderem in Abbildung 4.3 dargestellt sind. Zunächst wird das JSON-Objekt mittels automatischen POJO-Mapping der Jackson-Bibliothek, unterstützt durch Spring Boot, in Java-Objekte mit analogen Feldern überführt. Anschließend erfolgt eine Validierung und ein gleichzeitiger Aufbau des Datenmodellbaumes. Dabei werden die Java-Objekte des Zwischenformats, die jeweils einzelne Baumknoten repräsentieren, in finale Knotenobjekte umgesetzt. Im Unterschied zu dem Zwischenformat verwenden diese spezifischere Java-Objekte für ihre Felder statt generischer String-Typen, bedingt durch das JSON-Format. Unter Verwendung der Eltern- und Kindreferenzen des Zwischenformates können diese Namensreferenzen dann in Objektreferenzen der Knotenobjekte des finalen Formates umgesetzt werden, um so eine tatsächlich vorliegende Baumstruktur herzustellen. Die in Abbildung 4.3 gezeigten Felder der jeweiligen Klassen stellen nur eine Teilmenge der tatsächlich enthaltenen Felder und zusätzlichen Datenstrukturen dar. Für unterschiedliche Aufgaben, die das Datenmodell später lösen soll, werden im Laufe dieses Kapitels weitere Inhalte des Datenmodells eingeführt, die unter anderem schon beim Aufbau des Datenmodells erstellt werden. Zur besseren Verständlichkeit werden diese an dieser Stelle noch nicht behandelt.

Validierung von Datenmodellbäumen

Die Validierung von Datenmodellbäumen dient dazu zu verhindern, dass Nutzer invalide Datenbäume definieren, die zu späterem Fehlerverhalten der Software führen könnten. Ebenso gilt dies für das Speicherformat des Modells in der Datenbank, wo administrative oder technische Fehler zur Korrumpierung des Datenmodellspeicherformats führen könnten. Dieser Schritt ist besonders wichtig, da das festgelegte JSON-Spezifikationsformat für Datenmodelle anfällig gegenüber Fehldefinitionen ist. Beispielsweise ist es prinzipiell

möglich, durch falsche Angaben in den Eltern- und Kindarrays, beliebige Graphen zu definieren, die zum Beispiel keiner Baumstruktur entsprechen oder gar zusammenhanglos, also mit mehreren Wurzeln, sind. Neben solcher Definitionsfehlern soll zusätzlich auch die Größe des Datenmodellbaumes eingeschränkt werden können, um einerseits große Algorithmenlaufzeiten zu verhindern, die so potenzielle Angriffsflächen für Denial of Service-Attacken bieten könnten. Andererseits produzieren sehr große Bäume auch Schwierigkeiten bei der Darstellungen im Frontend, sodass eine Größeneinschränkung zusätzliche Randfallbetrachtungen bei der Entwicklung der Benutzeroberfläche erspart.

Im Folgenden sind die Validierungsschritte aufgeführt, die im Rahmen des Erstellens von Datenmodellbäumen durchgeführt werden. Dabei wird nach dem Prinzip der Short-Path-Evaluation vorgegangen, die Validierung also möglichst früh nach Erkennen eines Fehlers terminiert, um unnötige weitere Rechenzeiten einzusparen:

1. **Syntaktische Validierung der Speicherformate.** Sollte es nicht möglich sein die JSON- oder BSON-Repräsentationen mittels der jeweiligen POJO-Mapper zu parsen, liegen offenkundig syntaktische Fehldefinitionen vor, die eine Verwerfung des Datenmodells zu Folge haben.
2. **Überprüfung einzelner Baumknoten.** Anschließend folgt eine Validierung aller Eigenschaften, die Baumknoten zwingend erfüllen müssen, um überhaupt Teil eines validen Baumes sein zu können. Dazu gehören einfache Sanity Checks wie die Überprüfung auf leere Strings oder negative Ganzzahlen bei Array-Größenangaben. Auch die Überprüfung, ob mittels Namen referenzierte Knoten überhaupt definierte Knoteninstanzen sind, erfolgt in diesem Schritt. Daneben werden Eigenschaften überprüft, wie dass Knoten primitiver Typen keine Kinder haben dürfen, Array-Knoten eine definierte Größe und genau einen Knoten haben oder Knoten nicht zugleich Kinder und Eltern sein können oder gar sich selbst referenzieren. Auch wird darauf geachtet, dass Objekte stets Kinder haben müssen, das heißt selbst keine Blätter des Baumes sein dürfen. Wurde ein Knoten validiert, wird er von seinem Zwischenformat als `DataModelNode`-Instanz in eine `DataModelTreeNode`-Instanz (siehe Abbildung 4.3) überführt.
3. **Überprüfung globaler Baumeigenschaften.** Zuletzt erfolgen alle Validierungsschritte, die nur unter Berücksichtigung aller Baumknoten gleichzeitig durchgeführt werden können. Dazu gehört die Ermittlung des Wurzelknotens des Datenmodellbaumes, der sich dadurch auszeichnet, dass er keine Elternknoten hat. Das muss für einen validen Baum gemäß der festgelegten Definition auf genau einen Knoten zutreffen. Anschließend wird überprüft, ob alle Knoten ausgehend von dieser Wurzel erreichbar sind sowie, ob es sich bei dem Baum überhaupt um einen Baum handelt und nicht etwa um einen zyklischen Graphen. Für diesen Schritt wird ein Pre-Order-Traversierungsalgorithmus genutzt. Sobald ein Knoten des Baumes mehrfach traversiert wird, bricht der Algorithmus ab und zeigt einen Fehler an, da dies ein Indikator für einen Zyklus im Graphen ist. Dem hinzufügend wird zu Beginn auch die Knotenanzahl und die Höhe des Baumes ermittelt. Die Grenzen wurden dabei auf

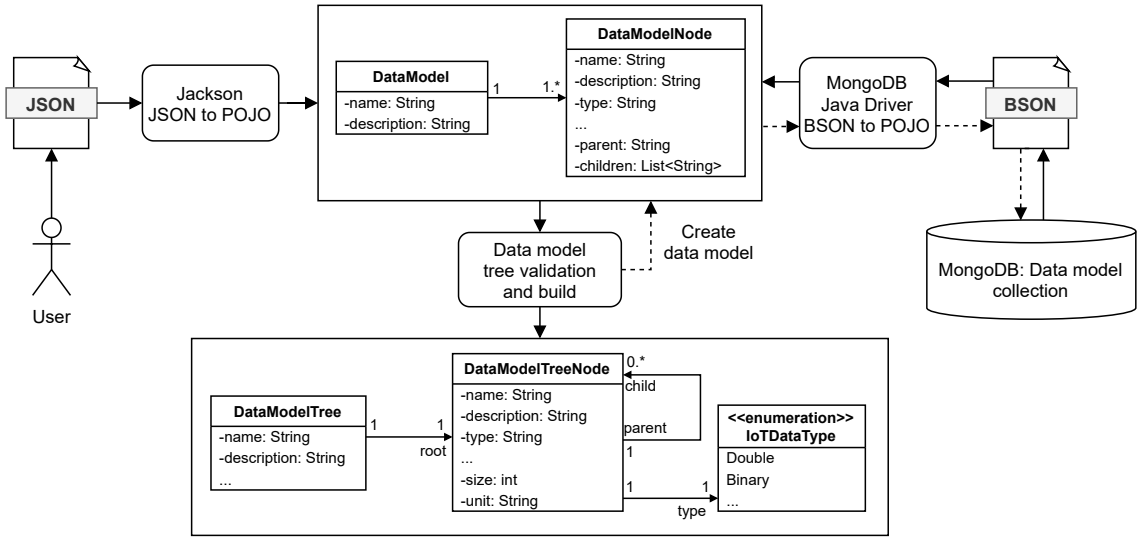

Abbildung 4.3.: Aufbau und Validierung eines Datenmodellbaumes innerhalb der Anwendungslogik auf Grundlage unterschiedlicher Speicherformate. Für den Aufbau von Datenmodellen wird ein internes Zwischenobjekt-Format angelegt, das anschließend in eine finale Objektrepräsentation überführt wird. Die n-äre Datenmodellbaum-Datenstruktur wird mittels entsprechender gegenseitiger Objektreferenzen realisiert.

25 Knoten mit einer Baumgesamthöhe von insgesamt fünf Niveaus festgesetzt. Zur Ermittlung der Baumhöhe werden, ausgehend von den Blätterknoten des Baumes, alle Pfade des Baumes rückwärts traversiert und die Anzahl begegneter Knoten zwischengespeichert. Das Maximum aller Pfadhöhen ist die Höhe des Baumes.

Um den Nutzern die Erstellung von Datenmodellen zu erleichtern, werden für viele unterschiedliche Fehlerarten genaue Fehlerbeschreibungen als Programmfeedback in den HTTP-Antworten mitgegeben, um diese gezielt beheben zu können.

4.1.5. Persistieren von Datenmodellen

In ähnlicher Weise, wie im vorigen Kapitel JSON als Spezifikationsformat für Datenmodelle genutzt wurde, erfolgt die Speicherung von Datenmodellen in der MongoDB in Form von analogen BSON-Dokumenten. Das soll heißen, dass die Speicherstruktur und die Inhalte sich im Wesentlichen nicht voneinander unterscheiden, bis auf die unterschiedliche Handhabung der Typen in beiden Datenformaten. Diese analoge Vorgehensweise hat den Vorteil, dass zum Abruf und zum Bau der Datenmodellbäume in der Anwendungslogik der gleiche Bau- und Validierungsalgorithmus verwendet werden kann wie bei der Deserialisierung der JSON-Spezifikationsformate. Insbesondere kann also das gleiche Objektzwischenformat, wie in Abbildung 4.3 dargestellt, verwendet werden. Das Mapping von BSON-Dokumenten

Listing 4.1 Beispiel für ein JSON-Objekt, das das Datenmodell-Beispiel aus Abbildung 4.2 definiert.

```
{
  "name": "Example data model",
  "description": "Description of the data model",
  "treeNodes": [{
    "name": "root",
    "type": "object",
    "parent": null,
    "children": ["temp", "location", "acceleration"]
  },
  {
    "name": "temp",
    "type": "double",
    "parent": "root",
    "children": []
  },
  {
    "name": "location",
    "description": "Object storing gps coordinates.",
    "type": "object",
    "parent": "root",
    "children": ["latitude", "longitude"]
  },
  {
    "name": "acceleration",
    "type": "array",
    "size": 2,
    "parent": "root",
    "children": ["accVal"]
  },
  {
    "name": "latitude",
    "type": "double",
    "parent": "location",
    "unit": "°N",
    "children": []
  },
  {
    "name": "longitude",
    "type": "double",
    "parent": "location",
    "unit": "°E",
    "children": []
  },
  {
    "name": "accVal",
    "type": "double",
    "parent": "acceleration",
    "children": []
  }]}
```

zu Java-Objekten wird von dem Java-Treiber der MongoDB übernommen. Dieser ermöglicht sowohl das Persistieren von Datenmodellen auf Grundlage des Zwischenformats als auch das Deserialisieren gespeicherter BSON-Datenmodelle zurück zu dem Zwischenformat, um es auf Grundlage dessen wieder den Datenmodellbaum innerhalb der Anwendungslogik aufbauen zu können.

Was bisher noch nicht geklärt wurde ist, wie das Datenmodell an die bereits existenten Entitäten für die Verwaltung und Anbindung von IoT-Objekten angebunden werden soll. Dazu sind mehrere Optionen denkbar. Eine Möglichkeit besteht darin, Datenmodelle immer zusammen mit zugehörigen ValueLogs zu speichern, was den Vorteil hat, dass zu den konkreten Ausprägungen der Telemetriedaten immer direkt eine Lese- und Interpretationsanleitung vorliegt. Solch ein Vorgehen findet beispielsweise auch bei Serialisierungstechnologien wie Apache Avro Verwendung, wo mit zu serialisierende JSON-Daten immer auch direkt ein Datenschema, konkret als JSON-Schema, mitgesendet wird [The21]. Der große Nachteil dieser Vorgehensweise ist, dass übertragen auf die MBP fortlaufend der komplette Datenmodellbaum valdiert und aufgebaut werden müsste, was einen vermeidbaren zusätzlichen Rechenaufwand darstellt. Stattdessen kann in der MBP die Tatsache ausgenutzt werden, dass die Art der angebundenen IoT-Objekte der Plattform stets bekannt sind und mittels Objekt-IDs auch in den jeweiligen ValueLogs referenziert sind. Letztlich sind es die Operator-Entitäten mit ihren Operatorskripten, die dafür zuständig sind die Sensorwert-Extraktor-Software bereitzustellen, und damit auch festlegen, welche Daten über MQTT an die MBP gesendet werden. Somit ergibt es Sinn für diese Entitäten auch jeweils das Datenmodell zu definieren, das den Aufbau und gegebenenfalls auch die Semantik dieser Daten beschreibt. Da jedes sendende IoT-Objekt (Sensor, Aktuator, Gerät) genau einen Operator hat, ist das Datenmodell für diese Objekte somit eindeutig definiert.

Abbildung 4.4 zeigt das überarbeitete Entity-Relationship-Diagramm von Abbildung 2.2. Ein Operator hat nun zusätzlich eine Referenz auf ein Datenmodell, das die Daten beschreibt, die die ihn verwendenden IoT-Objekte an die MBP senden werden. Für die Datenmodelle wird dabei eine separate MongoDB-Collection angelegt und die jeweiligen Objekt-IDs der Datenmodell-Dokumente in den Operator-Dokumenten als Dokumentenreferenz hinterlegt. Zwar wäre eine direkte Dokumenteneinbettung des Datenmodells in den Operator, wie in Kapitel 2.1.1 dargestellt, im Hinblick auf die Zugriffszeiten sinnvoll, ist jedoch an dieser Stelle eher zu vernachlässigen, da der Zugriff auf die Operator-Dokumente in einem nur geringen Ausmaße erfolgen wird (siehe auch im späteren Kapitel 4.3.1 über den Datenmodell-Cache).

Benutzeroberfläche zur Datenmodellerstellung

Um mittels der Benutzeroberfläche Datenmodelle zu erstellen, wurde unter dem Menüreiter Operators ein neuer Link zu einer Seite zur Verwaltung solcher angelegt. Abbildung 4.5 zeigt einen Screenshot dieser Seite, wobei für ein Datenmodell jeweils Name, Beschreibung und ein Datenbeispiel angegeben wird, das nach Mausklick angezeigt wird, wie in Abbildung 4.6

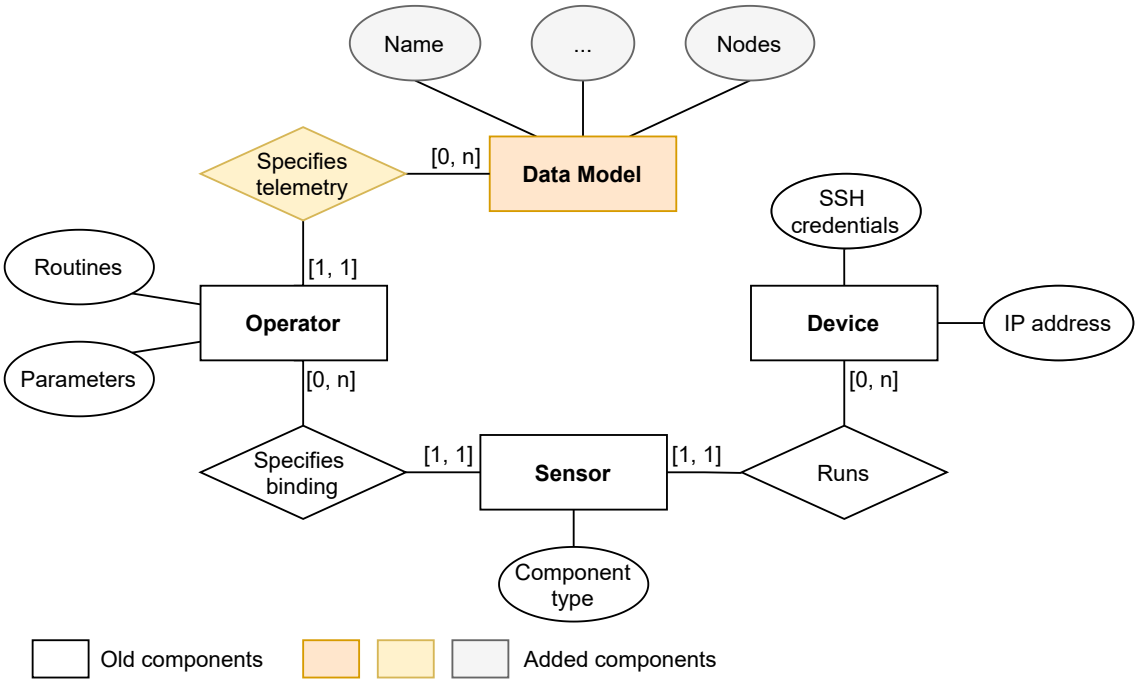

Abbildung 4.4.: Erweitertes Entity-Relationship-Diagramm von Abbildung 2.2. Die Anbindung der Datenmodell-Entität erfolgt durch eine Datenbankreferenz über den Operator. Es werden nur Attribute angezeigt, die für die grundlegende Funktion dieser Entitäten, an dieser Stelle der Arbeit, entscheidend sind. Die Kardinalitäten sind in Min-Max-Notation angegeben.

gezeigt. Das Datenbeispiel gibt an, wie korrekt formatierte IoT-Daten als Teil der JSON-MQTT-Nutzlast, unter Verwendung des jeweiligen Datenmodells, beispielhaft aussehen können. Das soll dem Nutzer helfen, erstens bereits erstellte Datenmodell anhand ihrer Struktur wiedererkennen zu können und zweitens gezielt nach diesem vorgegebenen Schema die Operatorskripte anpassen zu können.

Die Erstellung dieser Beispiel-Telemetriedaten findet für jedes Datenmodell einmalig bei dessen Erstellung statt und wird ebenfalls in dem Datenbankdokument des Datenmodells persistiert. Dazu wird das Datenmodell in Pre-Order-Reihenfolge rekursiv traversiert und für jeden Knoten entsprechende JSON-Elemente dem JSON-Beispielobjekt hinzugefügt. Algorithmus 4.1 zeigt die Funktion, die für diese rekursive Vorgehensweise genutzt wird. Initial wird der Wurzelknoten des Datenmodells übergeben sowie das Wurzelelement eines JSON-Objekts. Diesem Objekt werden dann sukzessive mit jedem rekursiven Aufruf, in Pre-Order-Reihenfolge, weitere JSON-Elemente, in Abhängigkeit des soeben behandelten Datenmodellknotens, hinzugefügt. Am Ende des Durchlaufes ist das komplette JSON-Objekt in dem initial übergebenem Objekt enthalten. Kennzeichnend für den Algorithmus ist die notwendige Fallunterscheidung zwischen JSON-Arrays und -Objekten. Denn für JSON-Objekt-Felder ist es notwendig, dass allen Feldern ein Schlüssel hinzugefügt wird, wofür der jeweilige Name des Datenmodellknotens genutzt wird. Arrays hingegen speichern

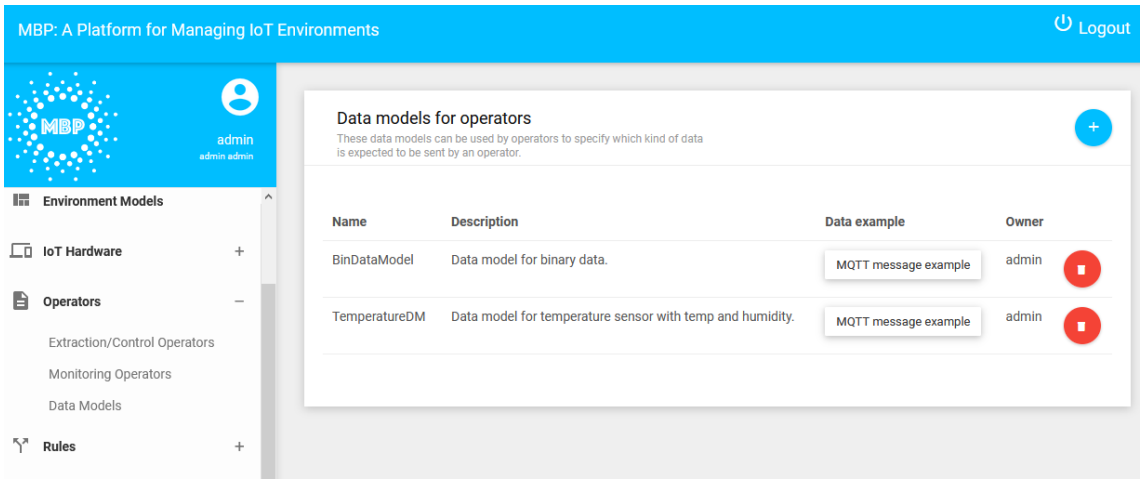

Abbildung 4.5.: Benutzeroberfläche zur Erstellung von neuen Datenmodellen oder zur Verwaltung bereits existenter.

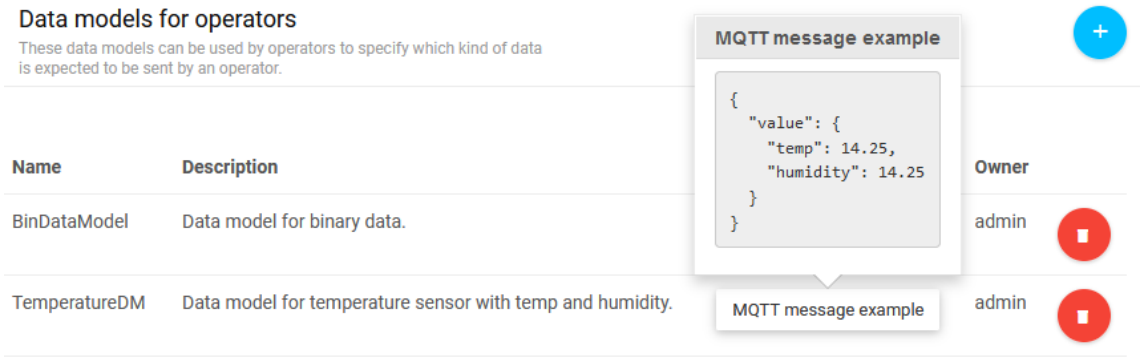

Abbildung 4.6.: Menüansicht von Abbildung 4.5 mit geöffnetem Telemetrie-Datenbeispiel für ein Datenmodell.

die Daten schlüssellos, jedoch identifizierbar durch einen Array-Index, ab. Der jeweilige rekursive Aufruf erfolgt für Knoten, die vom Typ Array oder Object sind und deshalb weitere Kindknoten haben können. Im Falle von Objects werden rekursive Aufrufe für alle Kinder veranlasst und bei Arrays so viele Aufrufe wie die spezifizierte Größe des Arrays es vorgibt. Hier zeigt sich, dass die in Abschnitt 4.1.2 eingeführte Konvention der festgelegten Array-Größen bei der Datenmodellbaumdefinition den Umgang mit Arrays für solche Anwendungszwecke erleichtert.

Neben der Anzeige der Beispiel-Telemetriedaten gibt es ein Button für die Option, das Datenmodell zu entfernen. Sollte das Datenmodell bereits mit einem Operator verbunden sein, so wird in diesem Fall eine Warnung unter Auflistung aller Operatoren ausgegeben, die dieses Datenmodell verwenden und durch ein Löschen nicht mehr von der MBP unterstützt werden können. Ein Editieren von Datenmodell wird, wie auch bei allen anderen MBP-Entitäten zur IoT-Objektanbindung, nicht unterstützt. Das hat den Grund, dass nach

Algorithmus 4.1 Rekursive Erzeugung von Beispieltelemetriedaten eines Datenmodells

```
procedure getJsonOfNode(currNode, lastArray, lastObject)
    if last json element is an object then
        if currNode is object then
            newJsonObject ← {}
            lastObject.add("currNode.key": newJsonObject)
            for all child in currNode.children do
                getJsonOfNode(child, null, newJsonObject)
            end for
        else if currNode is array then
            newJsonArray ← []
            lastObject.add("currNode.key": newJsonArray)
            for all i ∈ {0, 1, 2, ..., currNode.size} do
                getJsonOfNode(currNode.child, newJsonArray, null)
            end for
        else if currNode is primitive then
            lastObject.add("currNode.key": primitive example data)
        end if
    else if last json element is an array then
        ...                 // Analog to the first if branch, but with indices instead of keys
    end if
end procedure
```

einem Editieren der Entitätsinformationen bereits auf angebundenen Geräten installierte Operatorskripte ebenfalls automatisiert aktualisiert werden müssten. Dies kann jedoch von der MBP nicht automatisiert geleistet werden und muss deshalb manuell vom Benutzer durchgeführt werden.

Die eigentliche Ansicht zur Erstellung der Datenmodelle erscheint bei Klick auf den Plus-Button. Zunächst können in dem erscheinendem Fenster ein Name und eine optionale Beschreibung des Datenmodells angegeben werden. Darunter ist die Eingabemethode des eigentlichen Datenmodellbaumes zu sehen. Dabei handelt es sich um eine Baumstruktur, die analog zu der schon eingeführten Datenmodelldefinition aufgebaut ist. Jeder Knoten wird als aufklappbare Box dargestellt, wobei Eltern-Kind-Relationen mittels eingerückter Knoten visualisiert werden. Für jeden Knoten gibt es bis zu vier mögliche Nutzeroperationen, abhängig von seinem Datentyp. Für alle Knoten, mit Ausnahme des Wurzelknotens, steht eine Editier-Option zur Verfügung, die bei Mausklick alle Eingabefelder öffnet, die für den jeweiligen Knotentyp relevant sind. In Abbildung 4.7 ist beispielsweise zu sehen, dass für primitive Typen der Typ, der Name, die Beschreibung und die Einheit angepasst werden können. Für Arrays wird statt einer Einheit eine Arraygröße angegeben und für Objekte weder eine Einheit noch eine Größe. Für Arrays und Objekte steht eine Kind-Knoten-Hinzufüge-Option bereit (in der Abbildung zu sehen als +-Button). Arrays dürfen jedoch maximal einen Knoten haben, wie in Kapitel 4.1.2 erklärt, weshalb auch in der

Beispielansicht von Abbildung 4.7 keine Hinzufüge-Option für das Array `acceleration` angezeigt wird. Hinzugefügte Kindknoten lassen sich löschen, indem der rote Lösch-Button neben den Elternknoten betätigt wird. Die letzte Operation dient lediglich der komfortableren Darstellung der Baumstruktur. Dies ist das Einklappen und Ausklappen von Teilbäumen und ist in der Abbildung als hellblauer Button mit einem nach unten zeigenden Pfeil nach zu sehen.

Wird der Typ eines Knotens gewechselt, der zu Beginn standardmäßig immer als Double-Wert festgelegt ist, so passt sich die Eingabemethode entsprechend an, um die Korrektheit der Datenmodellbaumdefinition zu bewahren. Entscheidet sich der Nutzer beispielsweise aus einem Objekt mit mehreren Kindknoten ein Array zu machen, so werden automatisch alle Kinderknoten, bis auf eines, gelöscht. Für primitive Typen werden in diesem Fall alle Kinder gelöscht. Auf diese Weise ist es für den Nutzer strukturell eher schwer möglich, falsche Datenmodellbäume zu definieren, was der Nutzerfreundlichkeit zuträglich ist, da so keine detaillierten Vorkenntnisse zur Datenmodellbaumdefinition notwendig sind. Sollte der Nutzer dennoch einen Fehler begehen, schlägt die Validierung in der Anwendungslogik fehl und dem Nutzer wird konkret angezeigt, bei welchem Knoten aus welcher Ursache ein Fehler auftrat.

Da das Erstellen von Datenmodellbäumen, zumindest für sehr große Bäumen, durchaus aufwendig sein kann, besteht zusätzlich die Option, direkt eine JSON-Definition des Datenmodellbaumes in ein Textfeld anzugeben, indem auf den Button `Show json definition` geklickt wird. So können Nutzer, die beispielsweise zu schon vorhandenen Operatorskripten bereits Datenmodelle in Form eines JSON-Formats abgespeichert haben, ihre Definition direkt in das erscheinende Eingabefeld kopieren, ohne erneut manuell sich durch die grafische Eingabemethode klicken zu müssen. Das steigert die potenzielle Bedieneffizienz für erfahrene Nutzer.

Wurde das Datenmodell erstellt, kann dieses im Anschluss mit Operatoren verknüpft werden. Dazu bietet nun die Ansicht für das Erstellen von Operatoren ein Dropdown-Menü, aus dem alle verfügbaren Datenmodelle auswählbar sind. Im Gegenzug wurde das Eingabefeld für Einheiten entfernt, da Einheiten für Sensorwerte mit den neuen Änderung nun ausschließlich über das Datenmodell definiert werden.

4.1.6. Umgang mit eingehenden komplexen IoT-Daten

Überarbeitetes Nachrichtenformat der MQTT-Telemetrie-Nachrichten

Auch wenn aus den vorherigen Abschnitten bereits indirekt angedeutet wurde, wie Telemetriedaten datenmodellkonform an die MBP gesendet werden sollen, geht dieser Abschnitt noch einmal explizit darauf ein, wie die Nutzdaten in Folge der Konzeptabänderungen konventionell auszusehen haben. Als Datenformat soll JSON zur Repräsentation komplexer IoT-Daten eingesetzt werden. Das hat den Vorteil, dass das bestehende JSON-Format der

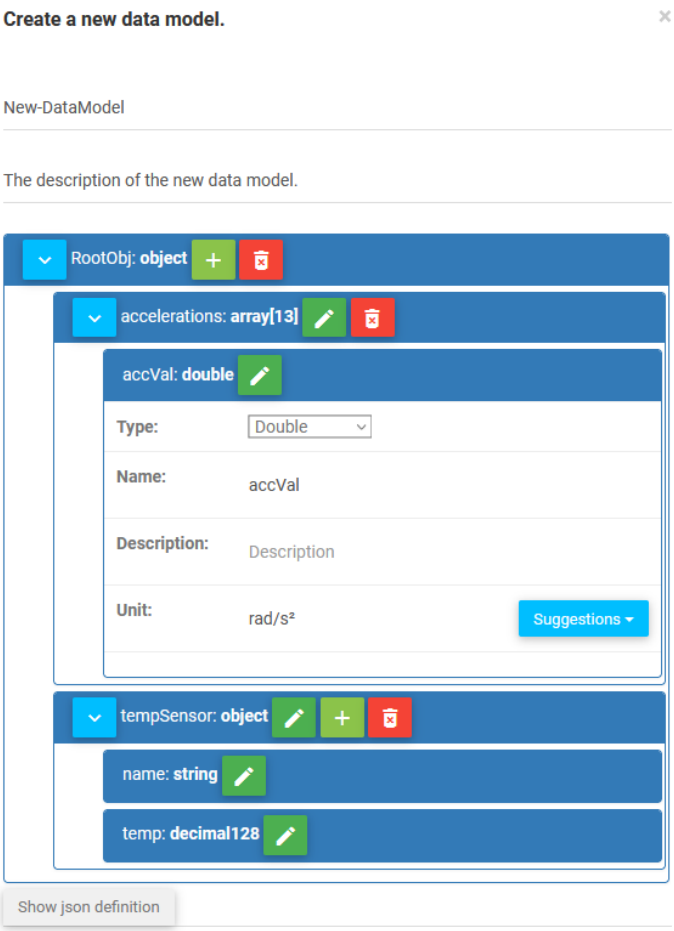

Abbildung 4.7.: Grafisches Benutzeroberflächen-Werkzeug zur Erstellung von Datenmodellen. Datenmodellknoten können dynamisch einem Baum hinzugefügt werden, wobei die definierten Datenmodellbaum-Einschränkungen dem Nutzer vorgegeben werden.

gesendeten MQTT-Daten weiterhin verwendet werden kann und nur um eingebettete JSON-Objekte erweitert werden muss. Zwar könnte man auch alternative Datenrepräsentation ohne JSON in Erwägung ziehen, zum Beispiel speziell kodierte Zeichenketten, möglicherweise unter Verwendung anderer Formate wie Extensible Markup Language (XML), jedoch bietet JSON als Speicherformat darüber hinaus für die MBP weitere Vorteile. Zunächst ist JSON ein weit verbreitetes und geläufiges Datenformat, das große technologische Unterstützung im Rahmen vorhandener Software-Bibliotheken und -Frameworks erfährt. Dieser Umstand erleichtert so beispielsweise Nutzern der MBP-Operatorskripte zu schreiben, die komplexe Sensordaten in das erforderliche MQTT-Nutzlast-Datenformat überführen können müssen. Konkret sind die derzeit vorhandenen Sensortwert-Extraktions-Skripte der MBP in Python geschrieben, das unter Verwendung von Standardbibliotheken ein direktes Mapping von Standard-Python-Datenstrukturen zu JSON-Strings ermöglicht [Lut13]. Die Verwendung von JSON erspart somit etwaige zusätzliche, vom Nutzer selbst zu entwickeln-

de, Sensorwert-Parsing-Schritte. Gleichzeitig ist JSON allgemein ein für den Menschen einfach zu lesendes und verständliches Format. Zuletzt ist auch zu bemerken, dass der Einsatz von JSON sich nahtlos in die bereits verwendeten Datenformate der MBP einfügt. So werden von der MBP nicht nur MQTT-Nachrichten in einem JSON-Format erwartet, sondern auch JSON-Objekte als Serverantworten für die REST-API verwendet. Auch die verwendete Datenbank, MongoDB, besitzt eine grundlegende Affinität zu JSON, da ihre Speicherformate, wie in Kapitel 2.1.1 dargelegt, sich stark an der JSON-Syntax orientieren. Unter anderem sind deshalb andere standardisierte Datenformate, wie die XML, für die komplexen Daten der MBP eher als zweite Wahl anzusehen.

Zur Erweiterung des bestehenden MQTT-Nachrichtenformats wird die bereits vorhandene JSON-Objektstruktur beibehalten. Im Unterschied zu Listing 4.2 werden nun allerdings, statt eines einfachen Zahlenwerts, ein JSON-Objekt für das Feld mit dem Schlüssel `value` erwartet. Die Struktur dieses Objektes muss dabei der im Operator hinterlegten Datenmodelldefinition folgen, um korrekt von der MBP gelesen werden zu können (mehr zur Datenvalidierung findet sich in Kapitel 4.3). Um hierfür ein Beispiel zu geben, zeigt Listing 4.2 wie eine valide MQTT-Nachricht, von einem Sensor ausgehend, der einen Operator mit dem Datenmodell aus Abbildung 4.2 verwendet. Erkenntlich daraus ist auch, dass bis auf die JSON-Schlüssel und die Struktur der Daten keine weiteren semantischen Informationen wie etwa Einheiten übermittelt werden. Diese zusätzlichen Informationen sind ausschließlich über das zuvor definierte Datenmodell ermittelbar. Diese Vorgehensweise hat den Vorteil, dass versendete MQTT-Nachrichten relativ leichtgewichtig bleiben und so auch der Gesamtperformanz für die Verarbeitung der eingehenden IoT-Daten zuträglich ist. Auch wird von dem Programmierer der Operatorskripte auf diese Weise nicht erwartet, in bestimmter Art und Weise semantische Informationen in das Nachrichtenformat unterbringen zu müssen, was zusätzliches Expertenwissen über die getroffenen Annahmen der MBP erfordern würde. Als Nachteil kann jedoch gesehen werden, dass das Nachrichtenformat so proprietär auf die MBP angepasst ist und womöglich schwerer für parallele Anwendungszwecke zu verwenden ist, für die ein möglichst kompletter Datensatz benötigt wird.

Die Einschränkung für das JSON-Objekt zur Versendung der komplexen IoT-Daten ist somit, dass es der Struktur des zuvor definierten Datenmodells für den jeweiligen Operator folgt. Unter anderem müssen also JSON-Schlüssel den Knotennamen des Modells entsprechen und Array-Strukturen der Größen- und Elementspezifikation des Modells folgen.

4.2. Repräsentation komplexer IoT-Daten in der Java-Anwendungslogik

Wie in Kapitel 2.1.2 beschrieben, werden empfangene MQTT-Nachrichten von IoT-Objekten an die MBP dazu verwendet, sie in ValueLog-Objekte zu überführen und an verschiedene interne Services zur weiteren Datenverarbeitung weiterzuleiten. An dieser

Listing 4.2 Beispiel einer validen MQTT-Nachricht eines Sensor an die MBP. Die Sensordaten der Nachricht ist konform mit dem Datenmodell aus Beispiel 4.2 formatiert.

```
{
  "component": "SENSOR",
  "id": "600db8c76dc71450f89a3927",
  "value": {
    "temp": 14.2,
    "location": {
      "latitude": 15.23,
      "longitude": -50.12
    },
    "acceleration": [
    0.25,
    20.21,
    41.0
]}}
```

Stelle stellt sich die Frage, wie komplexe IoT-Daten innerhalb dieser ValueLog-Objekte repräsentiert werden sollen. Eine geeignete Datenrepräsentation sollte die folgenden Eigenschaften erfüllen:

1. Das Format soll möglichst ohne aufwendige Zwischenschritte in JSON und BSON umwandelbar sein. JSON zum Nachrichtenaustausch über MQTT und mit der Präsentationsschicht, BSON für die Se- und Deserialisierung von BSON-Dokumenten aus der Datenbank. Die Formatüberführung soll dabei jeweils in beide Richtungen möglich sein (zum Beispiel ValueLog zu JSON und JSON zu ValueLog).
2. Die unterstützten primitiven Datentypen, so wie sie für das Datenmodell in Tabelle 4.1 definiert wurden, sollen für die jeweiligen gespeicherten Datenfelder verwendet werden können.
3. Die komplexen Datentypen Array und Object sollen in geeigneter Weise abgebildet werden können, um die Struktur der IoT-Daten über alle Datenformate hinweg konsistent zu erhalten.
4. Unter Heranziehung des Datenmodells soll es möglich sein, gezielte Lese- und Schreiboperationen auf die gespeicherten Daten anzuwenden.

Besonders Punkt vier ist wichtig, um die tatsächliche Verarbeitung von komplexen IoT-Daten auf Ebene der Anwendungslogik zu ermöglichen. Unter gezielten Lese- und Schreiboperationen sind hier solche Operationen zu verstehen, die ausgehend von den komplexen Gesamtdaten genau auf die Teildaten angewendet werden, die bestimmte anwendungsfallabhängige Eigenschaften erfüllen. Ein Beispiel für eine solche Operation ist das Runden von Dezimalzahlen auf eine gemeinsame Genauigkeit. Dies erfordert zunächst

einen Lese- und anschließend einen Schreibzugriff auf alle Datenfelder des Gesamtdatums, die vom Typ *Double* oder *Decimal128* sind. Bevor sich allerdings mit der Realisierung dieser Zugriffe beschäftigt wird, soll zunächst das Datenformat an sich vorgestellt werden.

4.2.1. Document-Klasse zur Repräsentation komplexer Daten in Java

Die Grundidee des gewählten Datenformats besteht darin, Datenobjekte, im Sinne des Object-Typs für Datenmodelle, als Java-HashMap zu repräsentieren und Arrays als verkettete Listen. Maps bieten in Java analog zu JSON-Objekten die Möglichkeit, Schüssel-Werte-Paare zu speichern und eignen sich deshalb als übergeordnete Datenstruktur hierfür. Ähnlich eignen sich Listen für Arrays, da sie, genau wie ihre JSON-Pendants eine indexbasierte Speichermöglichkeit anbieten. Für die Repräsentation primitiver Typen werden die Java-Klassen und -Typen verwendet, wie sie in Tabelle 4.1 definiert wurden. Ihre Instanzen werden entweder unter Angabe eines Schlüssels in der Map oder als Element in der Liste gespeichert. Analog zu verschachtelten JSON-Objekten und Arrays können auf diese Weise auch die jeweiligen Maps und Listen verschachtelt angelegt werden, sodass eine durch JSON oder BSON vorgegebene Datenstruktur erhalten werden kann.

Die oben formulierte Idee wird bereits von einer Klasse des MongoDB-Java-Treibers, namens *Document*, umgesetzt. Sie implementiert unter anderem das Java-Interface Map mit Strings als Schlüssel und Objekten vom allgemeinem Java-Object-Typ als Werte [Monc]. Die Verwendung dieser Klasse hat gegenüber einer neuen eigenen Implementierung den Vorteil, dass in ihr bereits Methoden implementiert werden, die eine gegenseitige Konvertierung der Datenstrukturen JSON und BSON unterstützen. Sie kann so im Allgemeinen als Java-Repräsentation von Dokumenten der MongoDB angesehen werden. Beispielsweise werden eingebettete Dokumente in BSON (oder analog verschachtelte Objekte in JSON) als rekursiv enthaltene *Document*-Objekte repräsentiert.

Abbildung 4.8 zeigt ein Beispiel der Datenrepräsentation komplexer IoT-Daten mithilfe der Klasse *Document*. Die Daten entsprechen denen des Value-Feldes aus der JSON-Repräsentation von Listing 2.1. Anhand des Beispiels ist sichtbar, wie vernestete Speicherstrukturen dynamisch an die zu repräsentierenden komplexen Daten angepasst abgebildet werden können. Das Location-Objekt zur Speicherung von Geo-Koordinaten wird als eigenes *Document*realisiert, dessen Instanz in der implementierten Map des *Documents* unter dem Schüssel `location` gespeichert wird. In gleicher Weise verweist `acceleration` auf eine Referenz einer Liste, die Double-Objekte speichert. Statt der Double-Objekte lassen sich analog für andere Datenmodelle alle primitiven Datentypen in den Datenstrukturen abspeichern, wie sie für die Java-Spalte in Tabelle A.1 aufgelistet werden.

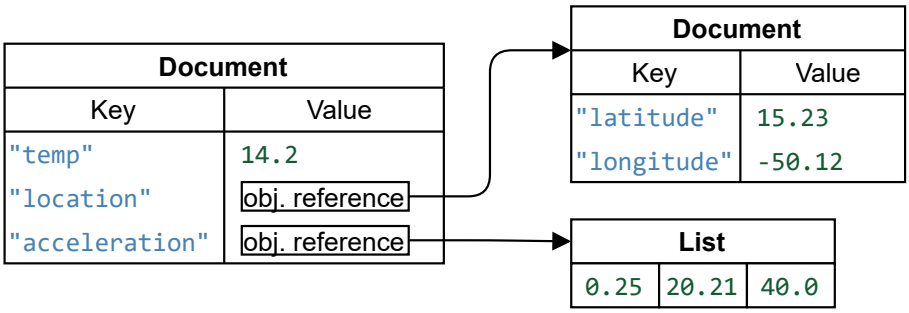

Abbildung 4.8.: Schemenhafte Darstellung der komplexen IoT-Daten-Repräsentation in der Java-Anwendungslogik anhand des Datenbeispiels von Listing 4.2. Die *Document*- und *List*-Boxen stellen hier jeweils Instanzen der jeweiligen Java-Klassen *Document* und *List* dar. Die dargestellten Zahlen sind für das Beispiel alle Instanzen der Klasse Double und die Zeichenketten Instanzen der Klasse String.

4.2.2. Gezielter Datenzugriff auf Document-Objekte

Der gezielte Zugriff auf komplexe Daten, gespeichert in *Document*-Objekten, ist eine der Hauptanwendungsfälle des Datenmodells. Ohne dieses besteht in der MBP kein Wissen darüber, wie Daten einzelner IoT-Objekte tatsächlich verwaltet werden, insbesondere also auch nicht, wie sie automatisiert verarbeitet werden können, was einen gezielten Zugriff auf die einzelnen Datenfelder der *Document*-Objekte erfordert. Um das Datenmodell hierfür in geeigneter Weise zu verwenden, wird eine Zugriffsklasse implementiert.

Die Grundidee des Datenzugriffs besteht darin, dem Zugriffsobjekt einen Knoten des Datenmodellbaumes zu übergeben. Dieses Objekt bietet mittels angebotener Methoden anschließend die Möglichkeit, die Daten zu lesen und zu bearbeiten, deren Speicherort durch diesen Datenmodellknoten abgebildet wird. Das funktioniert allerdings nur, wenn das zu lesende *Document* von einem IoT-Objekt stammt, dessen Datenmodell genau diesen Knoten als Teil des Datenmodells hat. Arrays stellen für den Zugriff eine besondere Herausforderung dar, da für einzelne Array-Elemente im Datenmodell keine einzelnen Knoten im Datenmodell definiert werden. Deshalb müssen zusätzlich, sollten im Baumpfad des zu lesenden Knoten Array-Knoten existieren, Indizes angegeben werden. Dazu wird eine Queue-Datenstruktur mit Array-Indizes dem Zugriffsobjekt übergeben, wobei die Reihenfolge der Indizes in der Queue startend von der Wurzel bis hin zu den Blättern des Baumes absteigend angegeben werden muss. Neben konkreten Indizes in Form von Ganzzahlen wird auch ein Wildcard-Index unterstützt, der zur Folge hat, dass alle Array-Elemente eines Knotens in Folge berücksichtigt werden.

Das Zugriffsobjekt bietet somit die gleichen Grundfunktionen wie JsonPath, dessen wichtigster Funktionsumfang in Kapitel 2.2 vorgestellt wurde. Der Hauptunterschied besteht jedoch darin, dass erstens auch ein Schreibzugriff unterstützt wird und zweitens, die Pfadangaben nicht über Zeichenketten, sondern mittels Datenmodellbaumknoten

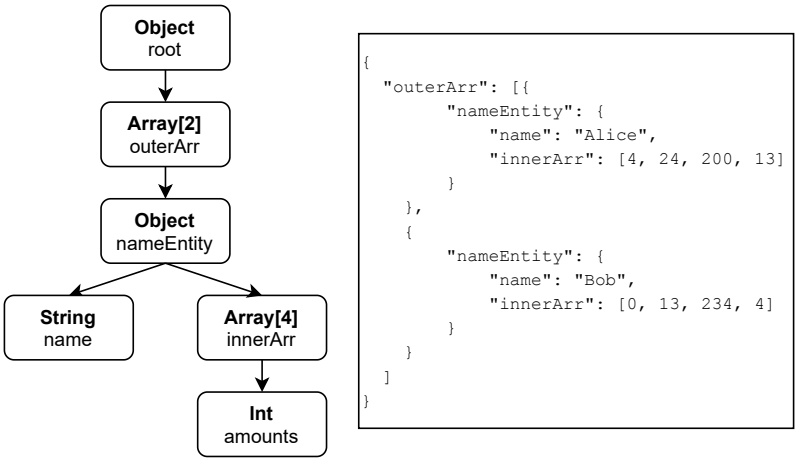

Abbildung 4.9.: Beispiel eines Datenmodells mit zugehörigen JSON-Beispieldaten, anhand derer im Folgenden die Funktionsweise des *Document*-Datenzugriffs erklärt wird.

definiert werden. Außerdem sieht die derzeitige Implementierung vor, dass ausschließlich auf primitive Typen zugegriffen werden darf. Diese Entscheidung soll verhindern, dass die zugrundeliegende Datenstruktur verändert wird und in Folge das *Document* nicht mehr mit seinem Datenmodell kompatibel ist. Im Folgenden wird die Funktionsweise des *Document*-Zugriffsobjekts näher beleuchtet. Um diese besser nachvollziehen zu können, wird ein neues Beispieldatenmodell eingeführt, das in Abbildung 4.9 gemeinsam mit einer konformen JSON-Dateninstanz abgebildet ist.

Schritt 1: Initialisierung des Zugriffobjekts

Die Idee des Zugriffsobjekts beruht darauf, dass es jeweils genau einen Datenpfad beschreibt. Auf diesen Datenpfad kann dann unter Angabe von *Document*-Instanzen beliebig oft über das Objekt zugegriffen werden. Diese Vorgehensweise hat den Vorteil, dass die für einen Pfad notwendigen Initialisierungsschritte nur einmal durchgeführt werden müssen, falls auf eine beliebige Anzahl von Dokumenten zugegriffen werden soll. Übergeben wird, wie oben schon erwähnt, ein Datenmodellknoten eines Datenmodells sowie eine Queue mit etwaigen Array-Indizes.

Zur Initialisierung des Objekts wird der Datenmodellknoten genutzt, um einen Baumpfad, bestehend aus allen Vorgängerknoten bis zur Wurzel, zu erstellen. Um diesen Schritt zu beschleunigen, wird bereits bei der Erstellung des Datenmodellbaumes für jeden Knoten einmalig eine Liste aller Vorgängerknoten berechnet und gespeichert, die so vom Zugriffsobjekt schnell verwendet werden kann. Angenommen das Ziel der Dokumentenabfrage im Beispiel sei es, den zweiten Array-Eintrag des `innerArr`-Arrays zu lesen oder zu bearbeiten, müsste man den Knoten `amounts` dem Zugriffsobjekt übergeben. Das Zugriffsobjekt würde

dann den Baumpfad mit den Knoten `root`, `outerArr`, `nameEnity`, `innerArr` und `amounts` initialisieren. Für Array-Indizes gilt die Konvention, dass negative Indizes den Wildcard-Operator anzeigen. Also müsste man als Index-Queue zum Beispiel `<-1, 1>` definieren. Hier wird davon ausgegangen, dass der Queue-Anfang, also das Ende aus dem Elemente entnommen wird, das linke Element der Liste ist.

Schritt 2: Aufruf der Document-Zugriffsfunktion

Nach der Initialisierung des Zugriffsobjekts kann die *Document*-Zugriffsmethode verwendet werden. Diese erwartet eine *Document*-Instanz als Parameter und optional ein Java-Objekt. Ist dieses Java-Objekt keine Null-Referenz, wird es verwendet, um alle Objekte, die in dem Dokument an der abgefragten Position gespeichert sind, mit diesem Objekt zu ersetzen. Auf diese Weise wird der Schreibzugriff auf *Documents* ermöglicht. Als Rückgabewert hat die Methode eine Liste aller Objekte, auf die der zuvor spezifizierte Baumpfad zeigt. Intern verwendet die Zugriffsmethode einen rekursiv definierten Algorithmus, der unter Verwendung des Baumpfades über die Datenstrukturen des *Documents* so lange iteriert, bis der Blattknoten des Pfades erreicht ist. Diese rekursive Vorgehensweise ist in Algorithmus 4.2 als Pseudo-Code dokumentiert.

Ein großer Teil dieses Algorithmus ist den Fallunterscheidungen von Objekt- und Arrayknoten geschuldet, die jeweils unterschiedlich gestaltete Zugriffe auf die *Document*-Datenstrukturen erfordern, je nachdem, ob auf Listen oder Maps zugegriffen werden soll. Davon abgesehen gibt es drei grundlegende Fallunterscheidungen, abhängig vom Typ des aktuell zu behandelnden Datenmodellknotens. Knoten, die einen primitiven Typ repräsentieren leiten das Rekursionsende ein, indem die Objekte der Rückgabeliste hinzugefügt werden und gegebenenfalls mit neuen Objekten überschrieben werden. Bei Objektknoten wird lediglich die nächste relevante Datenstruktur, gemäß angegebenem Baumpfad, extrahiert und der Algorithmus rekursiv unter Erhöhung des Baumpfad-Indexes aufgerufen. Für Arrayknoten geschieht dies ganz ähnlich, jedoch mit dem Unterschied, dass die Array-Index-Queue mitberücksichtigt werden muss sowie Wildcard-Operatoren unterstützt werden, indem so viele rekursive Aufrufe erfolgen, wie es die Array-Größe vorgibt. Im Folgenden werden zum besseren Verständnis die konkreten Parameter des Algorithmus beschrieben.

1. `returnObjects`: Eine Liste, die am Ende der Rekursion alle Objekte enthält, auf die das Zugriffsobjekt zugreifen sollte.
2. `nextNodeIndex`: Index, der als Zeiger auf einzelne Knoten der Baumpfadliste dient.
3. `currDataStructure`: Speichert die Datenstruktur, die vom nächsten Funktionsaufruf behandelt werden soll.
4. `arrIndex`: Variable für den nächsten Array-Index, der zu berücksichtigen ist. Wenn kein Array als nächster Knoten folgt, kann ein beliebiger Wert angegeben werden.

5. `objectToWrite`: Falls mit einer Objektinstanz belegt, wird dieses Objekt verwendet, um alle Objektreferenzen zu ersetzen, die sich aktuell an der abzufragenden Zugriffsposition des *Documents* befinden.

Algorithmus 4.2 wird folglich initial mit `nextNodeIndex = 0`, `currDataStructure` als Referenz des abzufragenden *Documents* und `arrIndex = -1` aufgerufen. Der Array-Index ist zu Beginn negativ, da gemäß der Datenmodelldefinition der erste Knoten immer einen Objekt-Typen repräsentiert. Für den ersten Rekursionsschritt müssen deshalb keine Array-Indizes berücksichtigt werden.

Der Beispielabfrage vom obigen Abschnitt folgend, greift der Algorithmus nacheinander auf die Datenstrukturen, die durch die Knoten des Baumpfades definiert werden, bis der Blattknoten `amounts` erreicht ist. Für den Knoten `outerArr` kommt der ausgewählte Wildcard-Operator zu tragen, der dafür sorgt, dass beide `nameEntity`-Knoten berücksichtigt werden. Für Baumknoten `innerArr` wurde hingegen ein definiter Index bestimmt, weshalb für diesen nur ein rekursiver Aufruf mit dem Index `1` gestartet wird. Nach Behandlung des Blattknotens wird als Ergebnis des Datenzugriffs demnach `[24, 13]` als Liste von *Double*-Objeken ausgegeben. Das entspricht der Zugriffsintention im Beispiel.

4.3. Validierung enigehender MQTT-Nachrichten

In Kapitel 2.1.2 wurde beschrieben, wie die MBP bisher mit ankommenden MQTT-Nachrichten umgegangen ist. Eine Validierung findet dabei als Nebeneffekt des JSON-Parsings der Nachrichten statt, mit dem Ziel die JSON-Strings in eine ValueLog-Instanz zu überführen. Ebenso wurde im vorherigen Kapitel die überarbeitete Struktur von ValueLogs zur Verwaltung komplexer Telemetriedaten in der Java-Anwendungslogik vorgestellt. Folglich müssen komplexe Daten, die die Anwendungslogik über den Message-Broker erreichen, in ebenfalls überarbeiteter Weise entgegengenommen werden, um die *Document*-Datenstruktur basierend auf der Nachricht korrekt herzustellen. Gleichzeitig muss erörtert werden, ob eine weiterführende Validierung dieser Daten gegen ihr Datenmodell sinnvoll ist und wenn ja, wie sie umgesetzt werden kann.

Zunächst ist anzumerken, dass alle notwendigen Berechnungsschritte für ankommende Telemetriedaten einen Flaschenhals für die Gesamtperformanz der weiteren Datenverarbeitung darstellen. Es sollte also Ziel sein, mit möglichst wenigen Schritten auszukommen. Eine Variante wäre deshalb ein Konzept zu verfolgen, dass ohne die Verwendung eines Datenmodells auskommt und Telemetriedaten allein auf Grundlage der vorliegenden JSON-Struktur in die richtige *Document*-Struktur überführt. Schließlich ist zumindest die Information über die Verschachtelung von Objekten und Arrays in analoger Weise auch direkt aus der JSON-Repräsentation zu entnehmen. Dies hätte den Vorteil, dass das Datenmodell so weder zu einer Baumstruktur deserialisert werden muss, noch zusätzlich iteriert werden muss. Besonders der Aufbau des Datenmodells kostet vergleichsweise viel Berechnungszeit, da die dafür verwendeten Algorithmen eine Worst-Case-Lufzeit

Algorithmus 4.2 Rekursive Funktion für den Document-Zugriff

```
treePathList ← List of data model nodes representing a path
indexQueue ← Queue of all specified array indices
procedure considerNextDataStructure(returnObjects, nextNodeIndex, currDataStructure, arrIndex,
objectToWrite)
    currNode ← treePathList[nextNodeIndex]
    if currNode is primitive then
        if currNode.parent is object then
            if objectToWrite ≠ null then
                currDataStructure[currNode.name] ← objectToWrite
            end if
            returnObjects ← returnObjects ∪ currDataStructure[currNode.name]
        else if currNode.parent is array then
            if objectToWrite ≠ null then
                currDataStructure[arrIndex] ← objectToWrite
            end if
            returnObjects ← returnObjects ∪ currDataStructure[arrIndex]
        end if
    else if currNode is object then
        if currNode.parent is object then
            nextDataStructure ← currDataStructure[currNode.name]
        else if currNode.parent is array then
            nextDataStructure ← currDataStructure[arrIndex]
        end if
        considerNextDataStructure(returnObjects, nextNodeIndex + 1, nextDataStructure, −1,
        objectToWrite)
    else if currNode is array then
        if currNode.parent is object then
            nextDataStructure ← currDataStructure[currNode.name]
        else if currNode.parent is array then
            nextDataStructure ← currDataStructure[arrIndex]
        end if
        nextArrIndex ← −1
        if indexQueue is not empty then
            nextArrIndex ← indexQueue.pop()
        end if
        if nextArrIndex < 0 then
            for all i ∈ {0, 1, 2, ..., currNode.size} do
                considerNextDataStructure(returnObjects, nextNodeIndex + 1, nextDataStructure, i,
                objectToWrite)
            end for
        else
            considerNextDataStructure(returnObjects, nextNodeIndex + 1, nextDataStructure,
            nextArrIndex, objectToWrite)
        end if
    end if
end procedure
```

in $O(n^2)$ haben, bei n Baumknoten. Die *Document*-Klasse des Java-MongoDB-Treibers unterstützt bereits eine Funktion automatisiert JSON-Strings in eine *Document*-Instanz umzuwandeln [Monc]. Sie würde sich somit prinzipiell ideal dafür eignen, in effizienter Weise, ValueLog-Instanzen aus den JSON-Telemetriedaten zu erstellen. Problematisch bei dieser Vorgehensweise ist jedoch, dass so Datentypen, die über Int, Double, String und Boolean hinausgehen nicht automatisch in korrekter Weise im *Document* berücksichtigt werden können. Dies wäre möglich, wenn die JSON-Daten dem sogenannten Extended JSON Format, spezifiziert von der MongoDB, entsprächen, bei dem Typen wie Binary und Date mit zusätzlichen semantischen Informationen kodiert werden [Monc]. Eine solche zusätzliche Kodierung müsste dann jedoch auch beim Erstellen der Extraktions-Software in den Operatorskripten vom Nutzer in korrekter Weise durchgeführt werden, was erstens zusätzliches Expertenwissen erfordert und zweitens gegen die grundlegende Idee des Datenmodells spricht, dass alle semantische relevanten Informationen getrennt von den konkreten Dateninstanzen gespeichert werden. Außerdem wird mit dieser effizienteren Variante so keine unmittelbare Konformitätsprüfung der komplexen IoT-Daten mit dem jeweiligen Datenmodell durchgeführt. Das würde dazu führen, dass unter anderem prinzipiell unerwartbare und fehlerhafte ValueLogs in der Datenbank abgespeichert werden könnten. Somit ist klar, dass sich diese Variante nicht zur Erstellung und Validierung von ValueLogs eignet und das Datenmodell stattdessen zu diesem Zweck herangezogen werden muss. Um die oben genannten Nachteile der Datenmodell-Variante teilweise zu kompensieren, wird jedoch ein Konzept des Datenmodellcaches eingeführt.

4.3.1. Datenmodell-Caching

Bei der Ankunft von Telemetriedaten muss zunächst das zum IoT-Objekt gehörende Datenmodell in der Datenbank gefunden werden und anschließend als Baumstruktur in der Anwendungslogik aufgebaut werden. Wie im Abschnitt zuvor beschrieben, wäre es relativ rechenintensiv für jede erhaltene Telemetrienachricht eine neue Baumstruktur zu erstellen. Vor allem, wenn jedes Mal erneut das gleiche Datenmodell zum ValueLog-Aufbau verwendet werden soll. Stattdessen wird ein Datenmodellcache implementiert, der für die MBP-Anwendungslogik als globaler Zugriffsservice für Datenmodelle dienen soll.

Der Cache ist im Wesentlichen eine Hashtabelle, bei der die Datenbank-Objekt-IDs der angebundenen IoT-Objekte als Schlüssel dienen. Als zugehöriger referenzierter Wert wird jeweils das Datenmodellobjekt gespeichert, das zu dem jeweiligen IoT-Objekt gehört. Das Datenmodellobjekt der Klasse DataModelTree bildet, wie in Abbildung 4.3 gezeigt, die fertig aufgebaute Baumstruktur des Datenmodells ab und wurde entsprechend bereits als valide validiert. Die Objekt-IDs wurden als Schlüssel gewählt, da sie stets mit in den MQTT-Nachrichten mitgesendet werden und sich somit direkt zur Datenmodellabfrage eignen. Wie Abbildung 4.4 zeigt, sind die Operatoren die Entitäten, die die Referenz des Datenmodells speichern. Durch das nun eingeführte direkte Mapping von IoT-ObjektId zu Datenmodell wird jedoch die Notwendigkeit aufgehoben, zusätzlich die Operator-Referenzen eines IoT-Objekts aufzulösen, was zusätzliche Rechenzeit spart. Die Objekt-ID

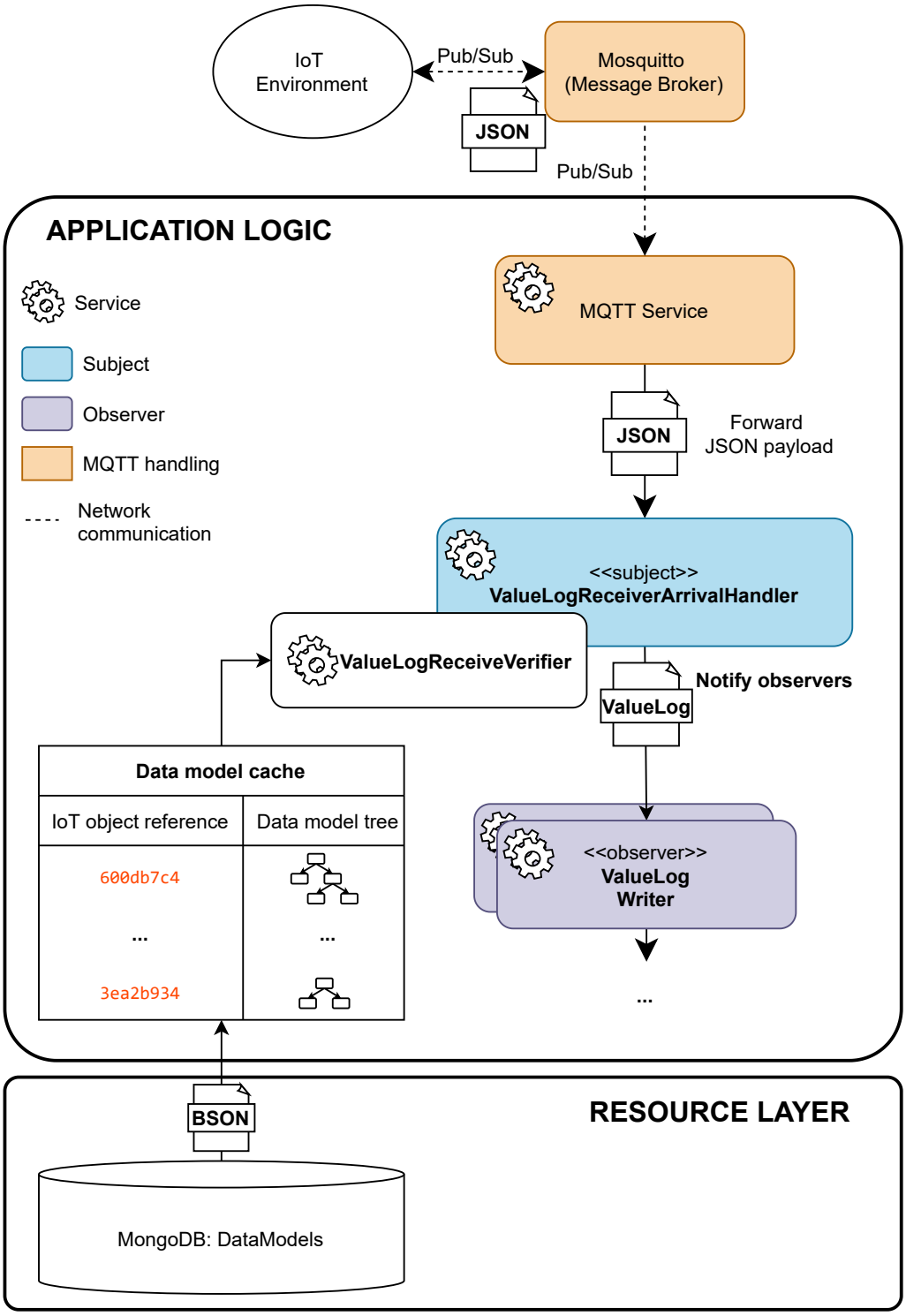

Abbildung 4.10.: Erweitertes Diagramm von Abbildung 2.4. Hinzugekommen ist ein Datenmodellcache und ein Service zur Verifizierung komplexer Telemetriedaten sowie zum Bau von *Documents* für ValueLogs.

wird von der MongoDB eindeutig und einmalig für ein Dokument vergeben [Mon20], weshalb es zu keinen Schlüsselkollisionen kommen sollte, die die Funktionsweise des Caches beschädigen könnten.

Ein Eintrag im Datenmodellcache wird immer dann erstellt, wenn eine MBP-Komponente erstmalig einen Datenmodellbaum, unter Angabe einer IoT-Objekt-ID, anfordert. Dann wird zunächst der Operator des Objekts ermittelt und dann das Datenmodell aus der Datenbank gelesen und in eine Baumstruktur überführt. Der Deserialisierungsmechanismus ist dabei der gleiche, wie er in Kapitel 4.1.4 vorgestellt wurde. Nach Aufnahme der Baumstruktur in den Cache werden alle weiteren Anfragen an ihn unter Verwendung der Hashtabelle beantwortet, wodurch eine schnelle Zugriffszeit auf das Datenmodell gewährleistet wird.

Schwieriger als das Schreiben von Datenmodellen in den Cache ist die Handhabung des Löschvorgangs von Einträgen. Das Aktualisieren von Einträgen kommt nicht als mögliche Operation infrage, da wie in Kapitel 4.1.5 beschrieben, das Editieren von Datenmodelle nicht unterstützt wird. Das Löschen von Einträgen soll verhindern, dass sich mit der Zeit zu viele Einträge im Cache sammeln, die möglicherweise überhaupt nicht mehr benötigt werden. Das gilt insbesondere für Datenmodelle, die von Nutzern eigentlich bereits gelöscht wurden. Um dem entgegenzutreten, verwaltet der Cache für jeden Eintrag zusätzlich einen Datumszähler, der angibt, wann zuletzt ein Eintrag abgerufen wurde. Über diese Information ist es möglich, in regelmäßigen Abständen Einträge zu entfernen, die nicht mehr verwendet werden oder gar nicht mehr existieren sollten. Beispielsweise könnte ein CronJob, ein Dienst, der auf Basis definierter Zeitintervalle eine Serveraktion durchführt, diese Funktion einmal am Tag aufrufen, um so alte Einträge der Nutzer zu entfernen. Obwohl diese Funktionalität für den Cache implementiert wurde, wird sie derzeit aufgrund der fehlenden Notwendigkeit, die MBP über lange Zeiträume am Stück laufen zu lassen, nicht verwendet. Ein Serverneustart leert den Cache ebenfalls.

4.3.2. Bau und Validierung von ValueLogs

Für den Bau von ValueLog-Instanzen wird der MBP ein neuer Service hinzugefügt, genannt *ValueLogReceiveVerifier*, der auch in Abbildung 4.10 abgebildet ist. Dieser Service bietet die Funktionalität unter Angabe eines Datenmodellbaumes und komplexer IoT-Daten, in Form eines JSON-Objekts, eine Instanz der Klasse *Document* (siehe Kapitel 4.2.1) herzustellen. Anders als vorhandene JSON-Parse-Funktionalitäten, die von der *Document*-Klasse angeboten werden, berücksichtigt der neue Service dabei auch alle im Datenmodell definierten Typen in vorgesehener Weise und übernimmt zu diesem Zweck die Überführung primitiver JSON-Typen in Java-Typen. Wie die Typenüberführung realisiert wird, ist in Kapitel 4.1.3 beschrieben. Das daraus bereitgestellte *Document* wird anschließend vom *ValueLogArrivalHandler* verwendet, um eine der MQTT-Nachricht entsprechende ValueLog-Instanz zu instanziieren. Gleichzeitig wird neben der Zusammenstellung des *Documents* auch eine Validierung des JSON-Objekts anhand des Datenmodells durchgeführt.

Algorithmus 4.3 Rekursiver Algorithmus zum Aufbau und der Validierung von Documents

```
procedure validateAndBuildNextChild(currNode, lastJsonArray, lastJsonObject,
lastDocument, lastList,nextArrayIndex)
    if lastJsonObject ≠ null then          // If last considered json element is an object
        if currNode.type is object then
            if lastJsonObject[currNode.name] exists then
                nextJsonObject ← lastJsonObject[currNode.name]
                nextDocument ← {}
                lastDocument ← lastDocument ∪ (currNode.name: nextDocument)
                for all child in currNode.children do
                    validateAndBuildNextChild(child, null, nextJsonObject,
                    nextDocument. null, −1)
                end for
            else
                return Validation error
            end if
        else if currNode.type is array then
            ...
        else if currNode.type is double then
            ...
            return
        else if currNode.type is decimal128 then
            ...
            return
        else if ... then
            ...
            return
        end if
    else
        // If last considered json element is an array
        ... // Analog to the first case but with accessing json arrays instead of json objects
    end if
end procedure
```

Der Algorithmus, um dies zu realisieren, ähnelt vom grundsätzlichen Aufbau, mit der Pre-Order-Traversierung des Datenmodellbaumes, stark den beiden bereits vorgestellten Algorithmen 4.1 und 4.2. Deshalb ist er als Algorithmus 4.3 nur grob skizziert, um die Grundstruktur vermitteln zu können.

Der Zugriff auf die JSON-Elemente erfolgt mittels gängiger JSON-Parsing-Bibliotheken, um besser von den syntaktischen Parsing-Aufgaben abstrahieren zu können. Für jeden traversierten Datenmodellknoten wird in Abhängigkeit des Knotentyps überprüft, ob an der erwarteten Stelle im JSON-Objekt sich tatsächlich ein typkompatibles Element

befindet und das betreffende Datenobjekt dann in eine *Document*-Repräsentation überführt. Sollte das nicht der Fall sein, schlägt die Validierung fehl. Im gezeigten Ausschnitt des Algorithmus wird für jeden Objektknoten eine neue *Document*-Instanz angelegt, die dann von den folgenden rekursiven Aufrufen der Kindknoten verwendet wird, um darin die entsprechenden Daten abzulegen. Die Rekursion endet jeweils, wenn Knoten primitiver Typen erreicht sind. Die Fälle für solche primitiven Knotentypen beschäftigen sich dann jeweils im Detail mit den typspezifischen Umwandlungsschritten, um beispielsweise eine Zeichenkette als Datumsobjekt berücksichtigen zu können. Schlägt die Validierung fehl, hat dies als einzige Konsequenz, dass die MQTT-Nachricht verworfen wird und nicht an weitere Services der MBP weitergeleitet wird.

4.4. Speicherung komplexer IoT-Daten in der Resourcenschicht

Bisher werden in der MBP alle ValueLogs über ein automatisches POJO-BSON-Mapping des Java-Treibers der MongoDB in die Datenbank geschrieben. Im vorherigen Abschnitt wurde beschrieben, wie die ValueLog-Klasse dahingehend für komplexe Daten angepasst wird. Das jeweilige `value`-Feld, einst ein Double-Objekt, wird ersetzt durch ein *Document*-Objekt, das die verschachtelten Datenstrukturen der komplexen Daten in geeigneter Weise abbilden kann. Auch auf dieses Objekt lässt sich das POJO-Mapping anwenden. Dadurch, dass bei der Festlegung der primitiven Datentypen für die unterschiedlichen Datenformaten, siehe Kapitel 4.1.3, darauf geachtet wurde, dass die Java-Typen ihren korrespondierenden BSON-Typen entsprechen, werden nun auch die verschiedenen Datentypen korrekt in eine BSON-Repräsentation überführt. In Konsequenz kann das bisherige ValueLog-Speichersystem der MBP ohne zusätzliche Änderungen auch für die neue ValueLog-Definition verwendet werden. Das hat den Vorteil, dass auch die Speicherstruktur der ValueLog-Collection, dem Entwurfspattern des Bucketing folgend, unverändert weiterbestehen kann. Dabei handelt es sich, wie in Kapitel 2.1.2 vorgestellt, bereits um die von MongoDB-Entwicklern empfohlene Vorgehensweise zur Speicherung großer Mengen von Zeitreihendaten, insbesondere für den Anwendungsfall des IoT. Statt des ursprünglichen *Double-Value*-Feldes eines ValueLogs in der Datenbank wird folglich nun ein eingebettetes Dokument gespeichert, analog zu der *Document*-Definition in der Java-Anwendungslogik.

Weiter scheint es relevant darauf einzugehen, inwiefern die Entscheidung, eine NoSQL als Datenbank im Kontext des IoT zu verwenden, überhaupt sinnvoll gegenüber traditionelleren Vorgehensweisen mit RDBMS ist. Denn ein großer Vorteil der NoSQL-Datenbank MongoDB, der erst jetzt mit den Änderungen an der MBP zur Unterstützung komplexer IoT-Daten zu tragen kommt, ist, dass sie ohne die Verwendung eines festen Datenbankschemas auskommt. Ohne diese Datenheterogenitätseigenschaft unter Verwendung relationaler Datenbanken müssten Kompromisse gefunden werden, wie IoT-Daten abgespeichert werden können. Beispielsweise könnte ein festes Schema definiert werden, bei dem die Spalte

für die Telemetriedaten generische JSON-Strings abspeichert oder alternativ auf andere Datenformate verweist. Problematisch ist hierbei jedoch, dass so die direkte Information über verwendete Datentypen verloren geht sowie direkte Datenbankabfragen auf Teildaten sich schwieriger gestalten. Obgleich es zumindest für den letzten Punkt auch RDBMS-Technologien gibt, die explizit Operationen zur Unterstützung von beispielsweise JSON bereitstellen [Pet17]. Dennoch sind dokumentenbasierte Datenbanken wie die MongoDB im Vorhinein prädestiniert dafür heterogene Daten abzuspeichern, weshalb dieser Punkt für eine Verwendung der MongoDB im IoT-Kontext der MBP spricht, statt dem Einsatz von RDBMS.

Eine weiteres Kriterium für die Eignung der Datenbank für die IoT-Daten der MBP ist die Performanz. IoT-Objekte senden potenziell in hoher Frequenz Telemetriedaten an die Plattform, die in Folge in ebenso hoher Frequenz in die Datenbank eingefügt werden müssen. Bei steigender Nutzeranzahl fallen auch anschließende Datenbankabfragen der Zeitreihendaten stärker in das Gewicht, die in der MBP vor allem für das Darstellen von Sensordaten in den Visualisierungsansichten des Frontends getätigt werden. Eine allgemeine Vergleichsstudie der MongoDB, als Vertreter einer NoSQL-Datenbank, mit MySQL, als Vertreter von RDBMS, findet sich beispielsweise in [GGPO15]. Dort kommen Győrödi et al. zu dem Ergebnis, dass alle vier Create Read Update and Delete (CRUD)-Operationen, getestet anhand eines Datensatzes mit 10000 Einträgen, eine kürzere Ausführungszeit unter Verwendung der MongoDB hatten. Rautmare und Bhalerao [RB16] wählen in ihrem Paper einen ähnlichen Vergleichsansatz wie das soeben erwähnte. Der Hauptunterschied ist jedoch, dass die Vergleichsaspekte konkret auf Anwendungen im Bereich des IoT bezogen werden. Bezüglich der Performanz hat bei ihren Tests die MongoDB nur bei Insert-Statements stetig kürzere Ausführungszeiten, als eine MySQL-Datenbank. Bei Select-Statements, also Lesezugriffen, hing die Ausführungszeit der MongoDB stark von der Art der Abfrage und der Anzahl vorhandener Dokumente ab. Bei großer Dokumentenzahl schnitt die MongoDB bei Select-Abfragen schlechter ab. Das Paper betont aber auch, dass sich NoSQL-Datenbanken horizontal leichter skalieren lassen, als RDBMS [RB16], wodurch sich diese Diskrepanz bei einer großen Anzahl an Datenbankeinträgen im Vergleich verringern könnte.

Zusammenfassend lässt sich sagen, dass die MongoDB als Datenbank zur Speicherung von komplexen IoT-Daten durchaus eine gute Wahl darstellt, aus den Hauptgründen der Unterstützung heterogener Daten und der Performanz von Schreiboperationen. Die Schreibperformanz ist besonders für die MBP wichtig, bei der Sensoren potenziell häufig schreiben müssen, jedoch Read-Operationen nur in meist größeren Zeitabständen erforderlich sind. Das liegt daran, dass die Anzahl voraussichtlicher gleichzeitiger MBP-Nutzer aufgrund der üblichen lokalen Systeminstallation gering ist, was zu einer niedrigeren Rate von Sensordatenabfragen führt. Angebundene Sensoren hingegen senden in kleinen Abständen kontinuierlich Daten an die MBP und lösen damit fortlaufend Schreiboperationen aus.

Im Kontext der Speicherung von komplexen Telemetriedaten muss bedacht werden, dass die maximal zulässige Dokumentengröße der MongoDB die Speicherung von Zeitreihendaten unter Verwendung des Bucket Patterns einschränkt. Wie in Kapitel 2.1.2 ausgeführt, werden

in der MBP je 80 ValueLogs zu einem Dokument zusammengefasst. Bei einer maximalen Dokumentengröße von 16 Megabytes stünde somit 200 Kilobytes pro ValueLog zur Speicherung zur Verfügung, wenn man vereinfachend davon ausginge, dass ein Dokument nur aus ValueLogs bestünde. Dies würde beispielsweise bedeuten, dass ValueLogs mit je einem String circa bis zu 50.000 Zeichen beinhalten dürfen, wenn man vom Worst-Case-Szenario ausgeht, dass für jedes Zeichen alle vier Bytes der UTF-8-Kodierung benötigt werden, die von BSON verwendet wird (siehe die BSON-Speizifikation in [Monb]). Für Sensoren, die gewöhnliche Metriken verschicken, sollte diese Speichergröße mehr als ausreichend sein. Sollen jedoch beispielsweise auch größere Binärdateien wie Bilder oder Videofragmente in einem ValueLog gespeichert werden, muss explizit darauf geachtet werden, dass die einzelnen ValueLogs diese Größe nicht überschreiten. Beispielsweise könnte man, um das zu gewährleisten, die über den MQTT-Message-Broker sendbare Nutzlast auf eine Größe von weniger als 200 Kilobytes beschränken. Diese Problematik der Berücksichtigung großer Binärdateien wird am Ende dieser Arbeit, im Ausblick in Kapitel 5, erneut aufgegriffen.

4.5. Visualisierung komplexer Sensordaten

Nachdem in den vorherigen Kapiteln die notwendigen Hintergrundmaßnahmen zur Verwaltung und Verarbeitung komplexer IoT-Daten in der MBP beschrieben wurden, soll es in den folgenden Kapiteln um die für den Nutzer sichtbaren Funktionalitäten der MBP gehen. Die Visualisierung von IoT-Daten ist eine dieser Funktionen. Wie die MBP Sensordaten bisher für numerische Sensorwerte dargestellt hat, wird in Kapitel 2.1.2 vorgestellt. Im Folgenden soll es um die notwendigen Anpassungen der MBP gehen, die die Visualisierung komplexer Daten ermöglichen. Das betrifft sowohl Änderungen an der serverseitigen Anwendungslogik, vor allem aber auch an der Benutzeroberfläche der MBP.

4.5.1. Problemstellung

Bisher verlassen sich alle Visualisierungsansichten der MBP auf die Prämisse, dass in allen Fällen ausschließlich einzelne numerische Sensorwerte als Zeitreihendaten visualisiert werden müssen. Mit der Einführung komplexer IoT-Daten verliert diese Annahme ihre Gültigkeit. Jetzt müssen prinzipiell beliebig komplex verschachtele Daten mit potenziell zahlreichen unterschiedlichen primitiven Datentypen visualisiert werden können. Um hierfür eine geeignete Lösung zu finden, ist es notwendig, zunächst zu wissen, welche Daten zwischen dem MBP-Server und dem Client zur Datenvisualisierung nach den Änderungen der letzten Kapitel ausgetauscht werden.

Für alle IoT-Objekt-Typen der MBP, Devices, Sensoren und Aktuatoren gibt es eine Detailansicht, die unter anderem die Visualisierung von ihnen gesendeter Sensorwerten ermöglicht. Wird diese Seite geöffnet, fragt sie den Server zunächst nach der digitalen

Objektrepräsentation dieses Objektes in der MBP ab. Diese wird als JSON-Objekt mit allen aufgelösten Datenbankreferenzen gesendet. Zu den Referenzen gehören dabei unter anderem der Operator inklusive seines enthaltenen Datenmodells. Die ValueLogs werden bei Bedarf vom Server abgefragt, wobei in der Anfrage bis zu einer Obergrenze von 2000 spezifiziert werden kann, wie viele der neusten ValueLogs maximal gesendet werden sollen. Die ValueLogs werden ebenfalls in der JSON-Repräsentation des BSON-Dokuments von der Datenbank gesendet, was für die komplexen Sensordaten bedeutet, dass immer alle Daten eines Sensors auf einmal für die Visualisierung zu berücksichtigen wären. Dies stellt ein Problem dar, denn das Frontend kann nicht für jedes denkbare Sensordatum, von denen es unendlich viele Ausprägungen geben kann, eine spezialisierte Visualisierung zur Verfügung stellen. Stattdessen muss ein Konzept entwickelt werden, mit welchem der Visualisierung eindeutig mitgeteilt wird, wie welche Daten für ein IoT-Objekt zu visualisieren sind. Dabei soll vermieden werden, dass innerhalb der Präsentationsschicht eine Heranziehung des Datenmodells notwendig ist, denn das würde erfordern, dass eine analoge Baumlogik, wie sie für die serverseitige Anwendungslogik implementiert wurde, auch für die Präsentationsschicht erforderlich ist. Dies würde nicht nur einen doppelten Entwicklungsaufwand bedeuten, sondern auch unnötige Redundanzen in das System bringen, die die klare Trennung der Aufgabenbereiche von Präsentationslogik und Anwendungslogik untergraben würde.

4.5.2. Modulare Visualisierung von Sensordaten

Das in Kapitel 4.1.2 definierte Datenmodell ermöglicht die Unterstützung von IoT-Daten, die nicht mehr alleine mit einer einzigen Art von Liniendiagramm darstellbar sind. Stattdessen sind arbiträre geeignete Visualisierungen dieser Daten denkbar, weshalb ein Konzept von modularisierbaren Visualisierungen unterstützt werden soll. Dabei sollen verschiedene Darstellungsformen in Form einzelner Modulen verfügbar sein, die um beliebig weitere erweitert werden können, sollten MBP-Entwickler die Notwendigkeit dazu sehen. Diese Module sollen von einem Nutzer je nach Bedarf zu seiner IoT-Objekt-Detailansicht hinzugefügt und auch wieder entfernt werden können.

Die Visualisierungsmodule haben alle voraussichtlich unterschiedliche Anforderungen an Daten, die sie visualisieren können. Denn beispielsweise ist klar, dass sich ein String schlecht in einem Liniendiagramm prädestiniert für numerische Werte darstellen lässt. Entweder müssten die Module also für vorliegende ValueLogs selbst herausfinden können, wie sie sich visualisieren lassen, oder den Modulen werden nur Daten in einer Form übergeben, von der sie sofort wissen, wie sie zu visualisieren sind. Die erste Variante würde erfordern, dass ein Mapping der Visualisierungsmodule zu den Telemetriedaten durch womöglich komplizierte Berechnungen in der Präsentationsschicht hergestellt werden. Aus bereits erwähnten Gründen im Problemstellungsabschnitt soll dabei aber das Datenmodell nicht erneut in eine Baumstruktur überführt werden müssen. Deshalb soll eine Variante verfolgt werden bei der die Anwendungslogik auf Serverseite vorgibt, welche Visualisierungsoptionen für ein vorliegendes Datenmodell existieren. Der Nutzer kann dann aus diesen Optionen

diejenigen heraussuchen, von denen er meint, sie passen am besten auf den vorliegenden Sensor und seine bereitgestellten Daten. Gleichzeitig sollen für jede Option dem Frontend Visualisierungsanleitungen mitgeliefert werden. Sie sollen angeben, wie innerhalb der Präsentationsschicht auf je einen ValueLog zugegriffen muss, damit die Daten extrahiert werden können, die von einem Visualisierungsmodul tatsächlich visualisiert werden können. Das hat den Vorteil, dass keine aufwendigen Berechnungen am Datenmodell oder an den konkreten ValueLogs in der Präsentationsschicht notwendig sind, da die Ergebnisse der semantisch relevanten Berechnungen bereits von der Anwendungslogik bereitgestellt werden. Eine Möglichkeit diese Extraktionsanleitungen für ValueLogs bereitzustellen, besteht in der Verwendung von JsonPath, einer Abfragesprache für JSON, die bereits im Grundlagenkapitel 2.2 eingeführt wurde. Da JsonPath Unterstützung durch Javascript-Bibliotheken erfährt, ist es für Visualisierungsmodule damit in einfacher Weise möglich ValueLogs so abzufragen, dass die zu visualisierenden Daten extrahiert werden.

Eine Alternative zu Datenzugriffsanleitungen mit JsonPath, die innerhalb des Browsers der Webanwendung ausgewertet werden müssen, ist, ValueLogs für verschiedene Module bereits serverseitig so bereitzustellen, dass nur noch die für die Visualisierung relevanten Daten gesendet werden. Das hat jedoch den Nachteil, dass so für mehrere gleichzeitig aktive Visualisierungsmodule einzeln Serverabfragen senden müssen, während bei einer Verarbeitung des gesamten ValueLogs in der Präsentationsschicht mit einem Verteilungsmechanismus, angelehnt an das Observer-Pattern, gearbeitet werden könnte, der jeweils nur eine Serveranfrage zur Bereitstellung der ValueLogs erfordert. Um diese Möglichkeit offenzuhalten, wurde sich nicht für diese alternative Lösungsmöglichkeit entschieden.

Datenmodelle für Visualisierungsmodule

Zunächst ist für das beschriebene Gesamtvorhaben zu klären, wie automatisiert herausgefunden werden kann, welche Daten sich durch welche Visualisierungsart visualisieren lassen. Hierfür eignet es sich, das Datenmodell der einzelnen IoT-Objekte heranzuziehen, das anschließend gegen eine Datendefinition einzelner Visualisierungsmodule verglichen wird. Dafür muss festgelegt werden, in welcher Weise definiert werden kann, welche Art von Daten von einem Visualisierungsmodul unterstützt werden. Die Idee, die dafür umgesetzt wird, beruht darauf, in gleicher Weise, wie Datenmodelle für Operatoren definiert werden, auch Datenmodelle für Visualisierungsmodule vorzudefinieren, die angeben, wie Daten auszusehen haben, um von dem Modul visualisiert werden zu können. Anschließend kann in den jeweiligen Datenmodellbäumen der Operatoren nach einem Teilbaum gesucht werden, der den jeweiligen Visualisierungsmodulbäumen entspricht. So kann herausgefunden werden, ob eine bestimmte Visualisierung die Telemetriedaten eines bestimmten IoT-Objekts darstellen kann und auf welche Weisen.

Das Problem, einen visualisierbaren Teilbaum in einem Datenmodellbaum zu finden kann als Spezialfall des Entscheidungsproblems der Inklusion ungeordneter Bäume (engl.: unordered tree inclusion) betrachtet werden, wie es von Kilpeläinen und Mannila in [KM95] formuliert wird. Das Problem stellt die Frage, ob aus zwei beschrifteten, unsortierten

Bäumen P und T, P aus T gewonnen werden kann, indem Knoten von T gelöscht werden. Das Löschen von Knoten mit Kindern hat zur Folge, dass diese Kindknoten auf den Elternknoten der nächsthöhere Baumebene übertragen werden [KM95]. Notiert man alle Lösungen dieses Problems, werden alle Teilbäume gefunden, die von der allgemeinen hierarchischen Struktur auf einen anderen Baum passen. So könnte man dann eine entsprechende Abbildung liefern, die, speziell angewandt auf das Datenmodell, eine Anleitung sein kann, wie auf ValueLogs eines bestimmten Datenmodells zugegriffen werden muss, um den gewünschten visualisierbaren Teilbaum zu erhalten. Ein großes Problem bei der Anwendung der Lösung des Inklusionsproblems ist jedoch, dass Kilpeläinen und Mannila zeigen, dass allein schon das reine Entscheidungsproblem, ohne also zwingend alle möglichen Lösungen zu notieren, für unsortierte Bäume NP-vollständig ist [KM95]. Der Datenmodellbaum wurde als ein solcher unsortierter Baum definiert, aus Gründen, die in Kapitel 4.1.2 nachlesbar sind. Besonders für große Bäume (Breite und Höhe) ist also mit einer sehr großen Algorithmenlaufzeit zu rechnen.

Da davon auszugehen ist, dass die Bäume von Datenmodellen in der Regel eher als klein anzusehen sind, könnte man über dieses Manko womöglich noch hinwegsehen. Ein weiteres großes Problem ist jedoch die Anzahl an möglicher Abbildungen des Visualisierungsmodulbaumes auf den Datenmodellbaum eines Operators. Die Abbildbarkeit von Knoten eines Datenmodelles auf die eines anderen ergibt sich nämlich aus den jeweiligen annotieren Datentypen dieser Knoten. Gelten diese als typkompatibel oder sind hinsichtlich ihres Types identisch, können die jeweiligen Knoten aufeinander abgebildet werden. Als Beispiel zeigt Abbildung 4.11 links einen Datenmodellbaum eines Operators, in welchem Teilbäume für ein Datenmodell eines Visualisierungsmoduls (rechts) gefunden werden sollen. Die Anzahl der Kombinationen für die verschiedenen möglichen Knotenabbildungen, die zu unterschiedlichen Datenvisualisierungen führen könnten, ergibt sich aus dem Binomialkoeffizienten $\binom{4}{2} = 6$. Denn das Aufteilen der zwei String-Felder der Visualisierung auf die Felder des ValueLogs mit dem linken Datenmodell, entspricht dem zweimaligen Ziehen aus einer vier-elementigen Menge ohne Reihenfolge und ohne Zurücklegen. Möchte man für die Visualisierung weiter die Bedingung aufheben, dass Visualisierungsfelder nicht mit jeweils den gleichen Daten belegt werden dürfen, entspräche dies sogar einem Urnenmodell mit Zurücklegen ohne Reihenfolge, also $\binom{N+n-1}{n} = \binom{4+2-1}{2} = 10$ möglichen Kombinationen. Mit zunehmender Knotenanzahl oder gar weiterer Objektverschachtelungen, die zu zusätzlichen Faktoren in der Berechnung der Kombinationsanzahl führen können, können also potenziell große Zahlen an möglichen Visualisierungsvarianten eines ValueLogs entstehen (zum Beispiel schon bei $\binom{8}{4} = 70$ Varianten). Deshalb stellt sich hier, abgesehen von der im obigen Abschnitt angesprochenen hohen Laufzeitkomplexität, die Frage, wie diese zahlreichen Kombinationen einem Nutzer zur Auswahl angeboten werden können. Oder überhaupt, in welcher Form diese potenziell großen Datensätze in der Anwendungslogik oder der Resourcenschicht berücksichtigt werden können und sollen.

Diese beiden Gründe, die hohe Komplexität, die nicht zuletzt auch tendenziell schwieriger zu schreibende Algorithmen erfordert und die große Anzahl möglicher Mapping-Kombinationen, führen zu der Entscheidung, dass die Datenmodelldefinition für Visualisierungsmodule stark eingeschränkt werden soll. Statt ganzen Datenmodellbäumen soll

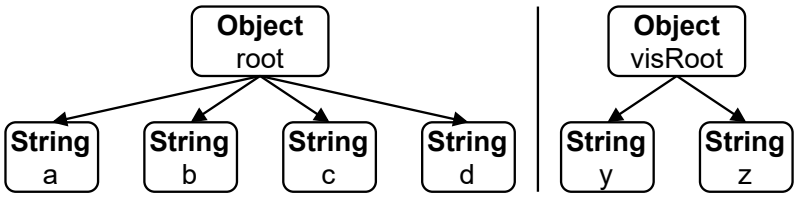

Abbildung 4.11.: Beispiel eines Datenmodelles eines Operators (links), in welchem Teilbäume in der Form eines Visualisierungsdatenmodells (rechts) gefunden werden sollen.

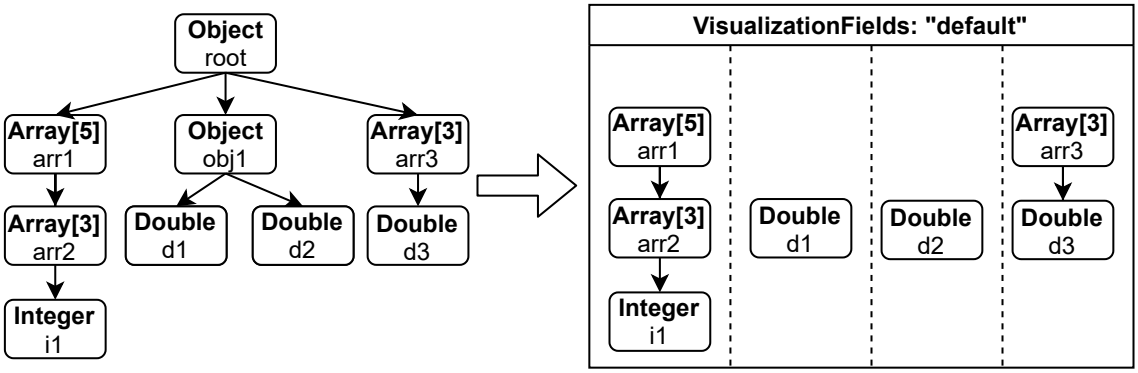

Abbildung 4.12.: Beispiel eines gedachten Datenmodells zur Modellierung von zulässigen Daten eines Visualisierungsmoduls und Umwandlung dieses Modells in eine datenfeldbasierte Speicherstruktur.

einem Konzept einzelner unterstützter Felder nachgegangen werden, die dann jeweils frei vom Nutzer mit Datenfeldern des ValueLogs belegt werden können. Ein Beispiel für solche Datenfelder einer Visualisierung ist in Abbildung 4.12 gezeigt. Eine Visualisierung, für die gemäß der ursprünglichen Intention ein komplexes Datenmodell zur Beschreibung der visualisierbaren Daten festgelegt wird, hat nun eine analoge Repräsentation in Form einzelner Baumpfade ohne Objektknoten. Alle Pfade werden dabei vollständig bis zu den Blattknoten, also jeweils bis zu den Repräsentanten der primitiven Typen, angegeben. Auf diese Weise können später in einem Mapping-Algorithmus von Visualisierungsdatenfelder auf Datenmodelle verfügbarer IoT-Objekte in effizienter Weise alle möglichen Mappings für jeden Baumpfad einzeln berechnet werden. Die daraus entstehenden unterschiedlichen Visualisierungsoptionen sind dann für jeden Baumpfad nach oben durch die Anzahl der Blattknoten des Operatoren-Datenmodells begrenzt. Dadurch kann der Nutzer später aus einer überschaubaren Anzahl an Visualisierungsoptionen pro zu visualisierendem Feld auswählen. Für Visualisierungsmodule wird festgelegt, dass immer alle Datenfelder, im Beispiel also alle vier Baumpfade, mit passenden Daten belegt sein müssen, damit diese visualisiert werden können. Wie dennoch etwas mehr Datenanforderungskulanz in das System eingebaut werden kann, wird im nächsten Abschnitt erklärt. Der genaue Algorithmus für das Datenmodell-Visualisierungsfelder-Mapping wird in einem späteren Abschnitt darauf aufbauend beschrieben.

Realisierung verschiedener Visualisierungseinstellungen

Das im vorherigen Abschnitt vorgestellte Beispiel eines Visualisierungsmodul mit vorgegebenen Datenfeldern ist nur für IoT-Daten einsetzbar, die mindestens ein zweidimensionales Integer-Array und einen Double-Wert enthalten. Über ein einziges Datenmodell pro Modul, in Form einzelner Datenfeld-Sammlungen hinaus, ist es wünschenswert, wenn zusätzliche Datenanforderungskulanz für Visualisierungsmodule realisierbar wäre, um möglichst viele unterschiedliche Daten mit einem einzelnen Modul visualisieren zu können, sollte sich die Darstellungsform dafür eignen. Dazu werden die folgenden Anforderungen an die Definitionsmöglichkeiten der Module gestellt:

1. **Datentypenkompatibilität.** Für einzelne Knoten der zulässigen Moduldatenfelder sollen im Fall von Knotentypen eines primitiven Datentyps, typkompatible Datentypen angegeben werden können, die dann in gleicher Weise als zulässiger Typ für das Datenfeld gelten. Besonders bei den numerischen Typen, *Int*, *Long*, *Double* und *Decimal128* ist dieses Feature sinnvoll einsetzbar. Denn wenn ein Visualisierungsmodul 128-Bit-Dezimalzahlen darstellen kann, ist es wahrscheinlich, dass auch alle anderen Zahlentypen darstellbar sind.
2. **Alternative Datenfeld-Anforderungsdefinitionen.** Für manche Visualisierungen ist es denkbar, dass sie abgesehen von Typenkompatibilitäten unterschiedliche Datenfeldansammlungen visualisieren können. Beispielsweise wäre das bei optionalen Datenfelder der Fall, für die ein Modul dann mehrere mögliche Datenfeldsammlungen definieren könnte, einmal mit dem optionalen und einmal ohne das optionale Datenfeld. Deshalb soll es möglich sein mehrere Datenfeldsammlungen, analog zu Beispiel in Abbildung 4.12, parallel nebeneinander zu definieren. In diesem Beispiel wird die konkrete Datenfeldsammlung unter dem Schlüssel `default` abgespeichert. Weitere Datenfeldsammlungen könnten parallel erstellt werden und mittels anderer Schlüsselbezeichner der Visualisierung zugewiesen werden.

Zur Realisierung dieser Eigenschaften wird jedes Visualisierungsmodul mithilfe drei verschiedener Klassen in der Java-Anwendungslogik modelliert. Diese sind in Abbildung 4.13 in Form eines UML-Diagramms zu sehen. Jedes Modul ist ein Objekt der Klasse `Visualization`, die eine Menge von `VisualizationFields`-Objekten enthält. Diese Objekte repräsentieren jeweils alle Datenfelder, die für eine Art der Datenvisualierung für das Visualisierungsmodul benötigt werden. Die Multiplizität `1..*` auf Seiten der `VisualizationFields` setzt Punkt zwei der obigen geforderten Eigenschaft um. Beispiel 4.12 zeigt schematisch ein einzelnes `VisualizationFields`-Objekt mit dem Namen `default`. Sowohl Objekte der Klasse `Visualization` als auch die `VisualizationFields`-Instanzen haben alle einen Namen, der zur eindeutigen Identifikation der jeweiligen Visualisierungen mit ihren jeweiligen Datenfeldern dienen.

Die einzelnen Datenfelder werden in Form eines Wurzelknotens eines Datenmodellpfades den `VisualizationFields`, innerhalb einer Map mit einem für das Modul eindeutigen Datenfeld-Bezeichner als Schlüssel, hinzugefügt. Um Punkt eins der beschriebenen wün-

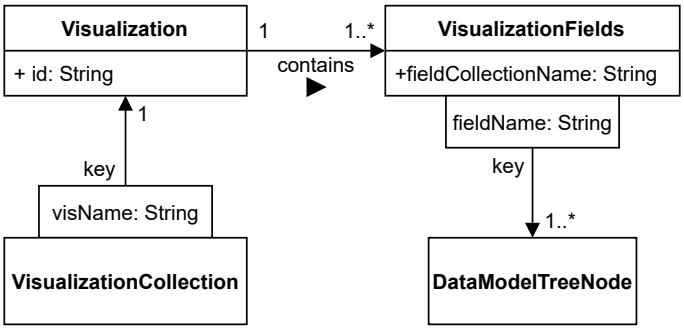

Abbildung 4.13.: UML-Klassendiagramm zur Modellierung der Klassen, die in der MBP-Java-Anwendungslogik verwendet werden, um Visualisierungsmodule abzubilden und innerhalb einer Konfigurationsklasse, der `VisualizationCollection`, als Konstanten zu definieren.

schenswerten Visualisierungsmodul-Eigenschaften zu erfüllen, wird jeweils eine Liste an Datenmodellwurzelknoten angegeben. Dies erlaubt es, mehrere einzelne Baumpfade mit unterschiedlichen Datentypen zu spezifizieren, die so mehrere kompatible Datentypvarianten definieren können.

Alle der Anwendungslogik bekannten Visualisierungen werden in einer Konfigurationsklasse als Konstanten von Entwicklern definiert. Dafür ist die `VisualizationCollection`-Klasse gedacht. In ihr werden die konkreten `Visualization`-Instanzen in einer Map gespeichert, mit einem unter den Visualisierungen eindeutigen Bezeichner. Die Bezeichner sollten sinnvoll gewählt und konsistent verwendet werden, damit sie dann auch im Frontend verwendet werden können, um herauszufinden, welche Daten welcher IoT-Objekte sich auf welche Weise visualisieren lassen. Statt der Verwendung einer Konfigurationsklasse wäre es alternativ auch denkbar, eine externe Konfigurationsdatei etwa mit JSON oder XML zu verwenden. Jedoch müsste diese dann stets beim Serverstart deserialisiert werden. Dieser zusätzliche Schritt erscheint nicht notwendig, weshalb auf diese Variante verzichtet wurde, vor allem da beispielsweise das Hinzufügen von neuen Visualisierungsmodulen im Frontend ohnehin mit Softwarecode-Anpassungen verbunden ist. Auch kann man darüber nachdenken, eine Datenbank-Collection für Visualisierungsmodule anzulegen, um diese dort zu persistieren, statt sie in der Anwendungslogik zu hartkodieren. Auch gegen diese Variante wurde sich entschieden, da diese Collection weder sehr viele Einträge haben würde, noch eine Bearbeitung der Datenbankeinträge zur Laufzeit notwendig wäre. Denn wie schon erwähnt, erfordern alle Änderungen am Visualisierungsmechanismus ohnehin Software-Anpassungen im Frontend und können deshalb nicht von einem User zur Laufzeit durchgeführt werden.

Abbildung 4.14 zeigt schematisch anhand eines Visualisierungsmodulbeispiels für geographische Koordinaten, wie konkrete Visualisierungsmodul-Instanzen innerhalb der `VisualizationCollection` definiert werden können.

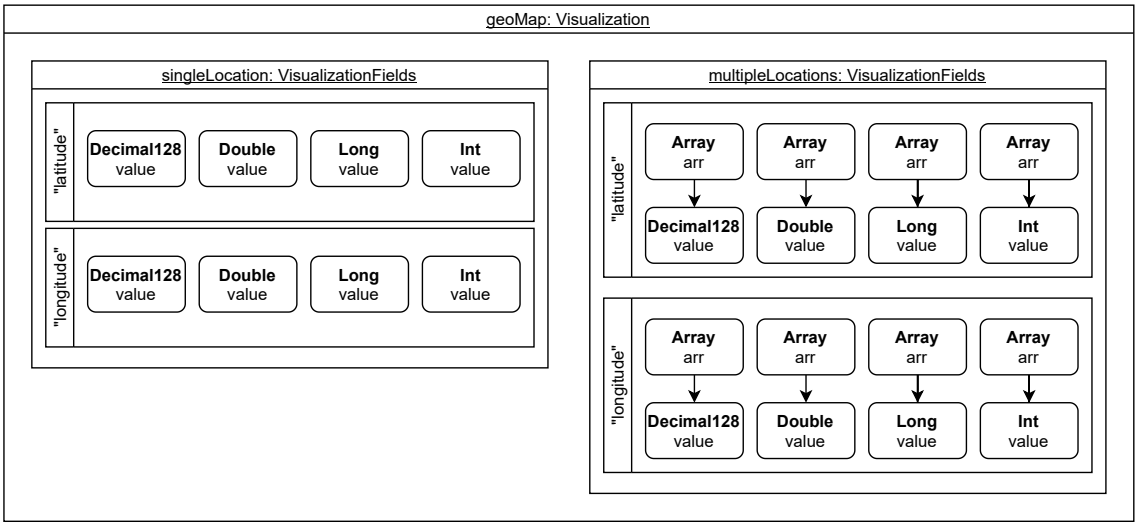

Abbildung 4.14.: Schemenhaftes Beispiel für eine Definition eines Visualisierungsmoduls zur Visualisierung von Geo-Koordinaten. Koordinaten müssen einem numerischen Datentyp entsprechen und können entweder zwei einzelne Datenfelder sein oder je zwei Arrays (jeweils für den Breiten- und Längengrad). Es wird in der Darstellung davon ausgegangen, dass etwaige nicht abgebildete Namen oder IDs, wie in Abbildung 4.13 definiert, den jeweiligen unterstrichenen Objektnamen entsprechen.

Datenstruktur zur Abbildung von Datenmodellen auf Visualisierungsmodule

Im letzten Abschnitt wurde beschrieben, wie Visualisierungsmodule für die Anwendungslogik definiert werden, um später anhand dieser Definitionen alle möglichen Visualisierungen für ein Datenmodell eines Operators berechnen zu können. Diese Berechnung benötigt jedoch als Ausgabe ebenfalls geeignete Datenstrukturen, die das Mapping von Visualisierungen zu dem jeweiligen Datenmodell repräsentieren kann. Ziel dieser Datenstrukturen ist es, für ein Datenmodell festzuhalten, welche Visualisierungen in der Lage sind dieses zu visualisieren und wie der Zugriff der Visualisierungen auf die ValueLogs mittels JsonPath aussehen muss, um die erwarteten Datenstrukturen abfragen und darstellen zu können.

Abbildung 4.15 ist ein UML-Diagramm, das alle Klassen zeigt, die an der Speicherung dieser Informationen beteiligt sind. Sollten die Struktur eines Datenmodell geeignet für die Darstellung innerhalb eines Visualisierungsmoduls sein, so wird für jedes dieser möglichen Module ein `VisMappingInfo`-Objekt in der DataModel-Klasse gespeichert. Zur Erinnerung: Das ist die Klasse, die letztlich angibt, wie Datenmodelle in der Resourcenschicht gespeichert werden. Da jede Visualisierung potenziell beliebig viele VisualizationFields haben darf (siehe Abbildung 4.13), kann ein `VisMappingInfo`-Objekt auch jeweils nur maximal so viele `VisualizationFieldMapping`-Objekte haben, wie sie für das jeweilige Modul definiert wurden. Allerdings ist das nur eine obere Grenze, denn es kann sein, dass das Datenmodell nur auf eine Teilmenge der definierten Sammlungen von Datenfelder der Module passt.

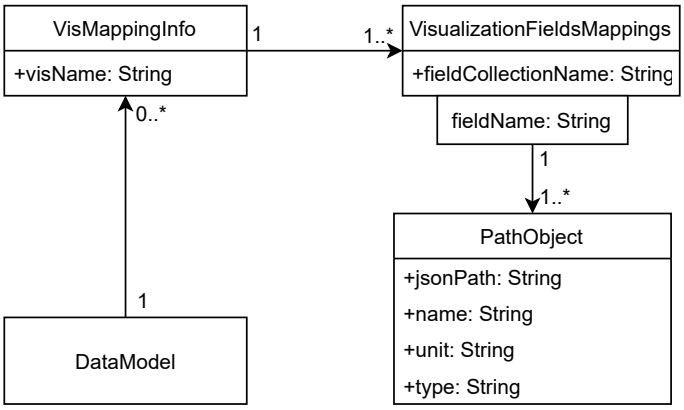

Abbildung 4.15.: UML-Klassendiagramm zur Veranschaulichung der verwendeten Datenstrukturen, um Operator-Datenmodelle und Visualisierungsmodul-Datenfelder aufeinander abzubilden. JsonPath wird als Zugriffsanleitung verwendet, um ValueLogs der IoT-Objekte passend für die Visualisierung im Frontend abfragen zu können.

Wie schon erwähnt gilt ein `VisualizationFields`-Objekt als passend für ein Datenmodell, wenn alle benötigten Datenfelder dieses Objektes vom Datenmodell abgedeckt werden können. Somit ist die Menge der `fieldName`-Mappings ausgehend von `VisualizationFields` -Objekten gleich der Mappings in den `VisualizationFieldsMappings`-Objekten. Während jedoch bei der Moduldefinition Feldnamen auf Datenmodellwurzelknoten abgebildet wurden, wird als Wertemenge nun eine Menge von `PathObject`-Ojekten verwendet. Diese Objekte repräsentieren alle konkreten Datenmodellblattknoten, die für das bestimmte Visualisierungsdatenfeld in Frage kommen und geben sowohl semantische Informationen zu diesem Knoten an als auch die JSON-Abfrageanleitung für diesen Knoten, in Form von JsonPath-Strings. Die semantischen Informationen werden benötigt, um dem Frontend Handhabungen mit der JSON-Repräsentation des Datenmodells zu ersparen, aber dennoch Zugriff auf visualisierungswerte Zusatzinformationen zu gewähren. Dazu zählt unter anderem der Name des Knoten, die optional vom Nutzer definierte Einheit und der Datentyp. Alle Attribute des `PathObjects` sind Strings, damit es später einfacher über das POJO-Mapping von Spring Boot als JSON-Nutzlast mittels der REST-API für die Präsentationsschicht bereitgestellt werden kann.

Wichtig für das Konzept ist, dass Visualisierungsnamen, Datenfeldsammlungs-Namen und einzelne Datenfeldernamen in gleicher Weise sowohl im Frontend als auch in der Anwendungslogik definiert sind. Denn diese Bezeichner dienen dem Frontend als Angabe, welche Visualisierung für welches IoT-Objekt zur Verfügung stehen soll, beziehungsweise auf welche Weisen Daten innerhalb einer Visualisierung visualisiert werden sollen. Werden diese Bezeichner serverseitig in der Anwendungslogik geändert, muss dementsprechend auch die Frontend-Implementierung angepasst werden und andersherum die Definition der Visualisierungsmodule.

Algorithmus zur Herstellung der Modul-Datenmodell-Abbildung

Nachdem die beiden Datenstrukturen vorgestellt werden, die für die Abbildung von Datenmodellen zu Visualisierungsmodulen benötigt werden, soll es nun um die konkrete Vorgehensweise zur Herstellung dieser Abbildung gehen. Dafür werden zunächst die Schritte erläutert, die vorbereitend vor Ausführung des Algorithmus durchgeführt werden.

Da die JsonPath-Angabe einzelner Knoten des Datenmodellbaumes eine Knoteneigenschaft ist, die potenziell öfters abgefragt werden soll, werden die Pfade für jeden Knoten schon bei der Validierung neuer Datenmodellbäume erstellt. Das bietet sich an, da die Erstellung der Pfade lediglich eine Pre-Order-Traversierung des Baumes erfordert, die wie in Kapitel 4.1.4 beschrieben, bereits genutzt wird, um beispielsweise den Datenmodell-Graphen auf etwaige nicht erwünschte Zyklen zu untersuchen. Beim Traversieren wird für jeden begegneten Knoten von der Wurzel ausgehend der JsonPath des Elternknotens um die entsprechenden Endungen der Kinder erweitert. Besonders Acht zu geben ist bei dieser Vorgehensweise auf die Array-Knoten im Datenmodell, da Arrays keine Namensbezeichner, sondern eckige Klammer im Pfad erfordern und dementsprechend die Kindknoten von Arrays nicht wie Kindknoten von *Objects* zu behandeln sind.

Abweichend von der JsonPath-Definition, wie in Kapitel 2.2 vorgestellt, wird für JsonPaths der Datenmodellbaumknoten eine erweiterte Syntax eingeführt. Diese hat das Ziel, die Arraygrößen in den Pfad zu kodieren, um damit primär dem Frontend in einfacher Weise mitteilen zu können, welche maximalen Arraygrößen als Nutzereingaben erlaubt sind. Diese Nutzereingabe ist für Arrays notwendig, da durch ein manuelles und dynamisches Einstellen der Array-Indizes es erspart bleibt, alle möglichen JsonPath-Array-Indizes-Kombinationen vorausberechnen zu müssen und diese dann dem Nutzer in vermutlich eher unübersichtlicher Weise darzustellen. Schließlich ist es auch der Sinn von Arrays, einen einfachen Zugriff auf große Anzahlen gleichförmiger Datenstrukturen zu gewähren. Um das zu realisieren, wird die Array-Größe für jeden Array im JsonPath in die eckigen Array-Klammern geschrieben, abgetrennt durch je zwei umklammernde `#`-Symbole. Für das in Abbildung 4.14 gezeigte mehrdimensionale Array im Datenmodellbaum würde der komplette Pfad bis zum Blattknoten `$['arr1'][#5#]['arr2'][#3#]` lauten. Die Rauten-Symbole dienen vor allem dazu, mittels String-Operationen später im Frontend die Arraygrößen erstens einlesen zu können und zweitens anschließend, um an den richtigen Stellen durch konkrete Nutzereingaben zu ersetzen. Wie diese Eingabe tatsächlich realisiert werden, wird in Kapitel 4.5.4 erläutert.

Algorithmus 4.4, 4.5 und 4.6 zeigen in Pseudo-Code, wie die Abbildung der Visualisierungsmodule auf ein Datenmodell hergestellt werden. Dazu wird Algorithmus 4.4 für alle Module innerhalb `VisualizationCollection`, siehe Abbildung 4.15, und für ein Datenmodell aufgerufen. Dieser ist dafür zuständig das `VisMappingInfo`-Objekt zusammenzubauen, indem für alle Datenfeldsammlungen des Moduls Algorithmus 4.5 aufgerufen wird, um wiederum eine Abbildung der einzelnen Visualisierungsdatenfelder auf Datenmodellbaum-

pfade herzustellen. Am Ende werden nur die Module zurückgegeben, die mindestens eine zum Datenmodell passende Datenfeldsammlung haben, was durch die If-Verzweigung in Algorithmus 4.4 umgesetzt wird.

Um mittels Algorithmus 4.5 einzelne Datenfeldsammlungen der Module auf das Datenmodell abzubilden, indem ein `VisualizationFieldsMapping`-Objekt zurückgegeben wird, wird über alle Datenfeldsammlungen des Moduls iteriert. Im Zuge dieser Iteration werden für jedes Datenfeld die alternativen Datenmodellbaumpfade betrachtet, die zum Zwecke der definierbaren Typkompatbilität eingeführt wurden. Für alle Pfade wird Algorithmus 4.6 aufgerufen, um zu überprüfen, ob im Datenmodell des IoT-Objekts ein solcher Pfad vorzufinden ist. Dies geschieht wiederum, indem zunächst die Array-Dimension des Baumpfades der Visualisierung bestimmt wird. Anschließend werden alle Blattknoten des Operator-Datenmodells iteriert und für jeden Knoten überprüft, ob ausreichend viele Arrays auf dem direkten Baumpfad von Blatt zu Wurzelknoten vorhanden sind. Dabei kommt es nur darauf an, dass mindestens so viele Array-Knoten im Operator-Datenmodellbaumpfad existieren wie auch der aktuell betrachtete Visualisierungs-Datenfeld-Baumpfad sie hat. Es ist also auch möglich, dass das Operator-Datenmodell für den jeweiligen Pfad mehr Arrays vorsieht. Auch wird davon ausgegangen, dass es für Visualisierungen keine Rolle spielt, welche Größe die Arrays haben. Bei Bedarf könnte man dieses Feature allerdings problemlos in das hier vorgestellte Konzept einfügen. Stimmen auch die Blattknotentypen von Visualisierung und Operator-Datenmodell überein, so wird das jeweilige `PathObject` erstellt und der Ergebnisliste hinzugefügt. Diese wird dann von dem aufrufenden Algorithmus 4.5, wie weiter oben beschrieben, verwendet, um sie zu `VisualizationFieldsMappings`-Objekten zusammenzufassen, die dann schlussendlich von Algorithmus 4.4 wiederum zu VisMappingInfo-Objekten verpackt werden.

Die soeben vorgestellte Gesamtprozedur wird immer bei der Erstellung von Datenmodellen in der MBP durchgeführt und dann mittels POJO-Mapping direkt in der Datenbank-Collection der Datenmodelle persistiert. Dadurch erhält das Frontend automatisch beim Aufruf der IoT-Objekt-Detailansicht ein JSON-Objekt mit allen benötigten Visualisierungsinformationen, um Telemetriedaten dieser IoT-Objekte modular und dynamisch auf Nutzerwünsche angepasst darzustellen. Wie die Umsetzung in der Benutzeroberfläche dafür konkret realisiert wurde, ist Thema des nachfolgenden Kapitels.

4.5.3. Benutzeroberfläche zur Visualisierung komplexer Sensordaten

In Kapitel 2.1.2 wurde bereits in kurzem Umfang die bisherige IoT-Objekt-Detailansicht des MBP-Browser-Frontends vorgestellt. Diese ist sehr stark auf die Konvention numerischer Telemetriedaten angepasst gewesen, weshalb zur Darstellung komplexer Daten einige neue Funktionalitäten hinzugefügt werden. Abbildung 4.16 zeigt die Detailansicht eines angebundenen Sensors. Die erste Änderung, die ins Auge fällt, ist unter anderem das Panel für die `Values Statistics`. Diese zeigt nun ausschließlich allgemeine Statistiken an, die datentypunabhängig für alle Telemetriedaten zur Verfügung stehen. Dazu gehören die Anzahl der bereits erhaltenen Datenpunkte, der erste jemals erhaltene Datenpunkt und

Algorithmus 4.4 Erstellung eines VisMappingInfo-Objekts für ein Visualisierungsmodul

```
procedure getVisMappingInfoPerVisualization(visualization)
    newVisMappingInfo ← {}
    for all visualizationFields ∈ visualization.visualizationFields do
        currVisFieldsMapping ← getVisualizationFieldsMapping(visualizationFields)
        if currVisFieldsMapping.pathObjectsMap.size = visualizationFields.size then
            // For all data fields a valid mapping exists
            newVisMappingInfo ← newVisMappingInfo ∪ currVisualizationFieldsMapping
        end if
    end for
    return newVisMappingInfo
end procedure
```

Algorithmus 4.5 Erstellung von VisualizationFieldsMapping-Objekten

```
procedure getVisualizationFieldsMapping(visFields)
    newVisualizationFieldsMapping ← {}
    for all dataFieldToPathObjectsMapping ∈ visFields do
        pathObjects ← {}
        for all dataModelNode ∈ dataFieldToPathObjectsMapping do
            pathObjects ← pathObjects
                ∪ getPathObjectsPerRootNode(dataModelNode)
        end for
        newVisualizationFieldsMapping ← newVisualizationFieldsMapping
            ∪ (dataFieldToPathObjectsMapping.key, pathObjects)
    end for
    return newVisualizationFieldsMapping
end procedure
```

der zuletzt erhaltene Datenpunkt, inklusive jeweils mit Datum des Erhalts. Alle weiteren Statistiken wie Lage- und Streuungsmaße wurden aus dieser Ansicht entfernt, da sie nur sinnvoll für einzelne numerische Daten zu ermitteln sind. Um dennoch weiterhin solche spezialisierten Statistiken zu ermöglichen, könnte man, als Weiterentwicklung der in dieser Arbeit vorgestellten Änderungen, ein separates Visualisierungsmodul für Statistiken anlegen, mit entsprechender Modul-Definition nach der Vorgehensweise, wie im letzten Kapitel beschrieben. Eine weitere Änderung, die die Statistiken betreffen, ist die Darstellung der einzelnen Datenpunkte für die zuerst und zuletzt erhaltenen Telemetriedaten. Da hier beliebig komplexe Daten dargestellt werden müssen, wird der jeweilige JSON-String des ValueLogs als Ganzes dargestellt. Zur übersichtlicheren Darstellung wird dieser als interaktive ein- und ausklappbare Baumstruktur angezeigt.

In der alten Visualisierungsansicht wurde standardmäßig immer das Liniendiagramm zur Darstellung historischer Sensorwerte angezeigt. Für komplexe Daten ist im Allgemeinen nicht klar, ob dieses Liniendiagramm überhaupt angezeigt werden kann. Deshalb wird im

Algorithmus 4.6 Mapping einzelner Visualisierungsdatenfelder zum Datenmodell mittels der Erstellung von PathObjects

```
leafNodesOfDataModelTree ← {n1, n2, ..., nn}
procedure GETPATHOBJECTSPERROOTNODE(rootNodeOfVisField)
    pathObjects ← {}
    visArrayDimension ← rootNodeOfVisField.getArrayPathDimension()
    leafNodeOfVisFieldPath ← rootNodeOfVisField.getLeaf()
    for all leafNode ∈ leafNodesOfDataModelTree do
        arrCount ← 0
        nextPathNode ← leafNode
        while true do
            if nextPathNode is an array then
                arrCount ← arrCount + 1
            end if
            if nextPathNode has a parent then
                nextPathNode ← nextPathNode.parent
            else
                break
            end if
        end while
        if leafNode.type = leafNodeOfVisFieldPath.type
          ∧ arrCount ≥ visArrayDimension                then
            pathObjects ← pathObjects
                ∪ (leafNode.name, leafNode.jsonPath, ..., leafNode.unit)
        end if
    end for
    return pathObjects
end procedure
```

Zuge der Einführung modularer Visualisierungen zunächst nur ein Menü zum Hinzufügen neuer Visualisierungsmodule angezeigt. Der Nutzer kann hier in einem Dropdown-Menü auswählen, welche Visualisierung er der Detailansicht hinzufügen möchte. Es stehen dabei nur die Visualisierungen zur Auswahl, die zuvor innerhalb der Anwendungslogik als passend für das verwendete Operator-Datenmodell errechnet wurden. Im schlechtesten Fall gibt es für bestimmte Telemetriedaten derzeit also keine Möglichkeit, sie zu visualisieren. Durch einen Klick auf den Plus-Button fügt der Nutzer das ausgewählte Modul der Ansicht hinzu. Es gibt keine Obergrenze, wie viele Module erstellt werden können, was dem Nutzer die Möglichkeit bietet, viele unterschiedlich konfigurierte Visualisierungen für seine Daten parallel einzustellen.

Nach Hinzufügen eines Moduls wird dieses als neues Panel dargestellt. Für jedes Modul stehen drei allgemeine Operationen zur Verfügung. Neben dem Aktualisieren des historischen Charts, was schon vor den neuen Änderungen möglich war, kann man nun

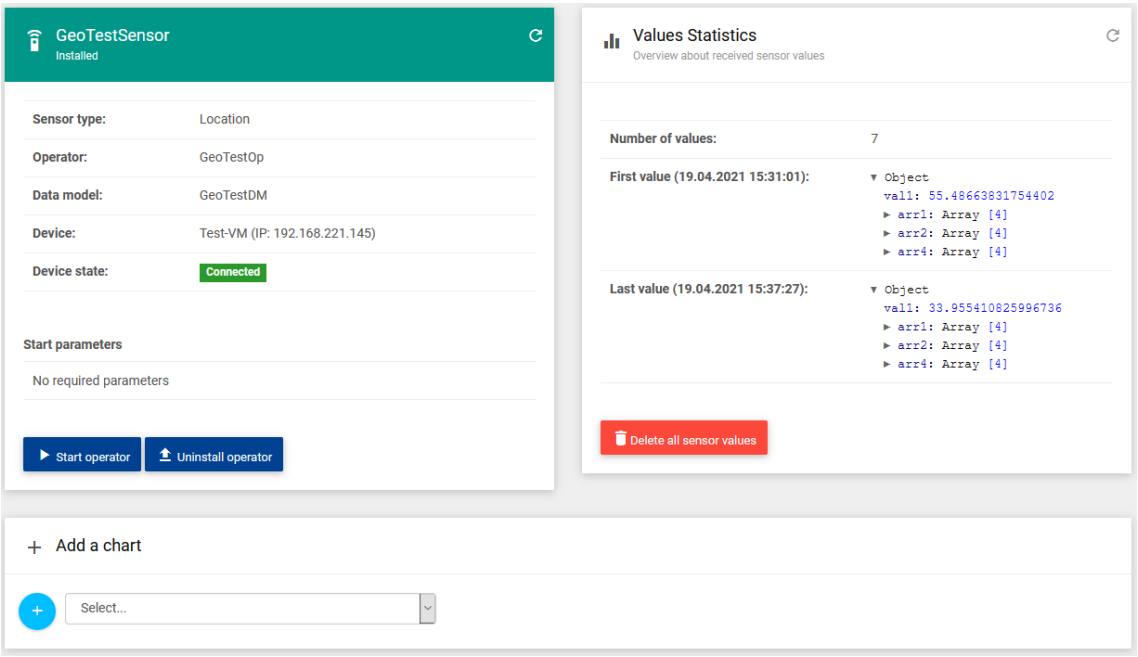

Abbildung 4.16.: Detailansicht eines an die MBP angebundenen Sensors. Für den Sensor sind noch keine Visualisierungsmodule hinzugefügt worden.

auch Einstellungen für ein Modul öffnen sowie das Modul wieder von der Detailansicht entfernen. Ein neu erstelltes Modul zeigt zu Beginn noch keine Daten an, da ihm nicht mitgeteilt wurde, welche Teildaten der komplexen Sensordaten es darstellen soll. Deshalb gibt es die Modul-Einstellungen-Option, um diese Information konfigurieren zu können. So wie die Darstellung der Sensorwerte an sich sind auch die verfügbaren Datenfelder für jedes Modul unterschiedlich. Je nachdem, ob es jeweils ein Satz von Datenfeldern gibt, im letzten Kapitel als `ViusalizationFields` bezeichnet, die sich für die Visualisierung der vorliegenden Sensorwerte eignen, werden die Datenfelder des Satzes als verfügbare Einstellungsoption angezeigt. Ansonsten wird die Option dem Nutzer gar nicht erst angeboten. Die einzelnen Datenfelder fordern einen JsonPath als Zeichenkette als Eingabe. Damit der Nutzer sich jedoch selbst kein Wissen über JsonPath aneignen muss und zudem nicht überlegen muss, welche Daten überhaupt visualisierbar sind, werden die errechneten passenden JsonPaths, wie im vorherigen Kapitel erklärt, dem Nutzer als Dropdown-Menü für jedes Datenfeld angezeigt, aus denen er auswählen kann. Wurden alle Datenfelder ausgewählt und die Eingabe bestätigt, verwendet das Visualisierungsmodul fortan die spezifizierten JsonPaths unter dem spezifizierten VisualizationFields-Identifikationsnamen, um JSON-ValueLogs geeignet abzufragen. Die so gewonnen Teildaten können von der Visualisierung anschließend angezeigt werden. Diese Einstellungen können vom Nutzer jederzeit geändert werden, wodurch es auf diese Weise auch möglich ist, beispielsweise für Visualisierungen die Datenachsen zu tauschen. Die Form der JsonPath-Eingabe wird in Abschnitt 4.5.4 im Detail vorgestellt.

Sobald der Operator des Sensors gestartet wird und deshalb Echtzeit-Daten zu erwarten sind, wird für jedes Visualisierungsmodul auf eine Live-Ansicht umgeschaltet. Echtzeitdaten und historische Daten sind dann, statt wie in der alten Visualisierung untereinander, nebeneinander in Form von Tabs angeordnet. Durch die Wahl des entsprechenden Reiters kann auf die live- beziehungsweise historische Ansicht für jedes Modul umgeschaltet werden. Anders als historische Charts, die manuell vom Nutzer mittels Knopfdruck aktualisiert werden müssen, aktualisieren sich die Live-Charts in regelmäßigen Zeitintervallen selbst. In der derzeitigen Implementierung der MBP sendet jedes Modul einzeln eine Serverabfrage, um die neusten ValueLogs zu erhalten. Eine performanz-optimierende Erweiterung wäre, dass für alle Module in regelmäßigen Zeitabständen nur einmal die ValueLogs abgefragt werden und diese dann, dem Observer-Pattern folgend, an alle Module verteilt werden zur weiteren Verarbeitung mit ihren eingestellten JsonPaths. Die Designentscheidung, die Auswertung der JsonPaths für jedes Modul einzeln im Frontend zu unternehmen, statt in der Anwendungslogik, ermöglicht diese Vorgehensweise.

4.5.4. Nutzereingabemethode für JsonPaths

Für die Eingabe von JsonPaths wäre ein einfaches Dropdown-Menü in allen Fällen ausreichend, unterstützten Datenmodelle nicht Array-Konstrukte, die zusätzlich die Angabe von Array-Indizes erfordern. Zu diesem Zweck wird, wie im letzten Kapitel erklärt, eine erweiterte JsonPath-Syntax eingeführt, die vom Frontend nun in geeigneter Weise genutzt werden soll. Dazu wurde eine neue Eingabemethode entwickelt, speziell für die Eingabe solcher erweiterter JsonPath-Zeichenketten. Diese ist beispielhaft in Abbildung 4.17 zu sehen. Sobald ein Pfad ausgewählt wird, der Arrays enthält, wird für jedes Array eine zusätzlich Dropdown-Eingabe dargestellt. Die Inhalte dieser Dropdown-Menüs sind die verfügbaren Array-Indizes, die auf Basis der in den #-Klammern angegebenen Array-Größen berechnet werden. Zusätzlich kann die Eingabemethode so vom Entwickler konfiguriert werden, dass Wildcard-Operatoren für Arrays erlaubt sind, wodurch auch das gesamte Array für die Visualisierung ausgewählt werden kann.

Zur Eingabe von JsonPaths wäre es prinzipiell genauso möglich, dass der Nutzer eigene JsonPaths konfigurieren kann, die dann auch weiterführende Sprachelemente wie Array-Slices oder Fliteroperationen verwenden können. Dann läge es allerdings in der Verantwortung des Nutzers korrekte JsonPaths anzugeben, die für die Visualisierung geeignete Daten bereitstellen können. In der hier vorgestellten Lösungsvariante wird deshalb die Möglichkeit manueller JsonPath-Eingaben zugunsten der Nutzerfreundlichkeit verwehrt. Eine denkbare Erweiterung wäre jedoch, die Eingabemethode in eine Freitexteingabe umschaltbar zu gestalten, durch beispielsweise einen Button neben dem Dropdown-Menü.

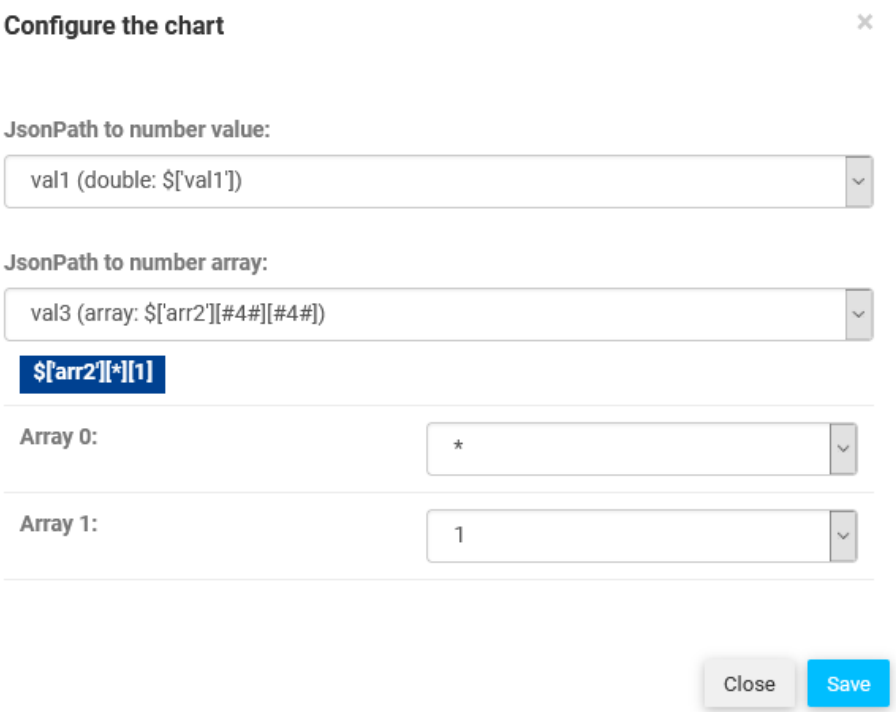

Abbildung 4.17.: Zwei JsonPath-Eingabefelder innerhalb einer Ansicht zum Konfigurieren eines Linien-Diagramms. Da der untere JsonPath Array-Felder enthält, müssen für ihn zusätzlich Array-Indizies spezifiziert werden, wobei der resultierende JsonPath als blau unterlegter Text zu sehen ist.

4.5.5. Vorstellung vorhandener Visualisierungsmodule

Für die Darstellung von Telemetriedaten der MBP wurden zwei Visualisierungsmodule implementiert, wobei das erste, Linien-Diagramme, gegenüber der alten MBP-Version lediglich angepasst und erweitert werden musste. Die beiden Module werden im Folgenden kurz vorgestellt.

Linien-Diagramme

Linien-Diagramme dienen zur Visualisierung von numerischen Zeitreihendaten. Auf der Abszisse ist die zeitliche Reihenfolge der Daten aufgetragen, während die Ordinate den jeweiligen Sensorwert anzeigt. Diese Darstellungsform ist die einzige vorhandene in der alten Version der MBP und bleibt leicht überarbeitet, in gleicher Weise auch für komplexe Sensordaten einsetzbar. Dazu muss ein JsonPath, auf ein numerisches Datenfeld des ValueLogs zeigend, angegeben werden. Zu den erlaubten numerischen Typen gehören Int, Long, Double und Decimal128, die wie in Kapitel 4.5.2 beschrieben, mittels Listen strukturell gleichartiger Datenmodellpfade als mögliche Typen für dieses Datenfeld

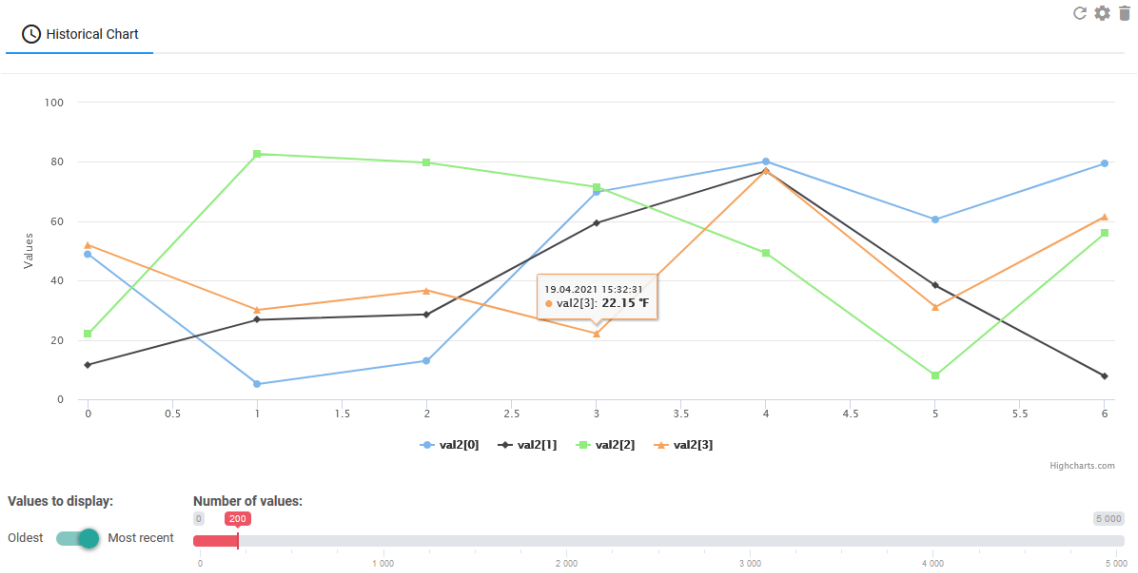

Abbildung 4.18.: Darstellung eines Arrays mit vier Double-Werten innerhalb der historischen Ansicht des Liniendiagramm-Visualisierungsmoduls.

definiert werden. Darüber hinaus kann ein Array numerischer Typen angezeigt werden, indem mehrere Linien für jeden Array-Eintrag dargestellt werden. Dieser Anwendungsfall ist in Abbildung 4.18 zu sehen.

Wurden in dem Datenmodell des IoT-Objekts Einheiten für die Datenfelder spezifiziert, werden diese zusammen mit dem jeweiligen Wert und der Sensorwertankunftszeit in einem Tooltip angezeigt, der erscheint, wenn man mit der Maus über einen Datenpunkt fährt. Die Zeitreihendaten sind in der Legende jeweils so benannt, wie der unterste Blattknoten des Datenmodells es vorgibt.

Weltkarte für geografische Koordinaten

Liniendiagramme haben als Moduleinstellungen jeweils nur ein Datenfeld pro verfügbarer Datenfeldsammlung. Um die Funktionsweise des Konzepts mit mehreren Datenfelder pro Datenfeldsammlung, wie in Kapitel 4.5.2 vorgestellt, zu demonstrieren, wurde zusätzlich eine geografische Karte als Visualisierungsmodul implementiert. Diese hat nur eine Datenfeldsammlung zur Verfügung, jedoch mit zwei Datenfeldern, je eines für den Breiten- und Längengrad. Für beide Felder sind, wie schon beim Liniendiagramm, alle verfügbaren numerischen Typen erlaubt.

In der historischen Diagrammansicht wird jeder Sensorwert als Markierung auf der Karte an der entsprechenden geografischen Position dargestellt. Um die Reihenfolge der Sensorwerte sichtbar zu machen, wird neben den Markierungen jeweils eine Zahl angezeigt, die die zeitliche Reihenfolge der Positionen angibt. Diese passt sich, je nachdem, wie viele Sensorwerte man anzeigen lässt, an die Menge aller aktuell visualisierten Werte

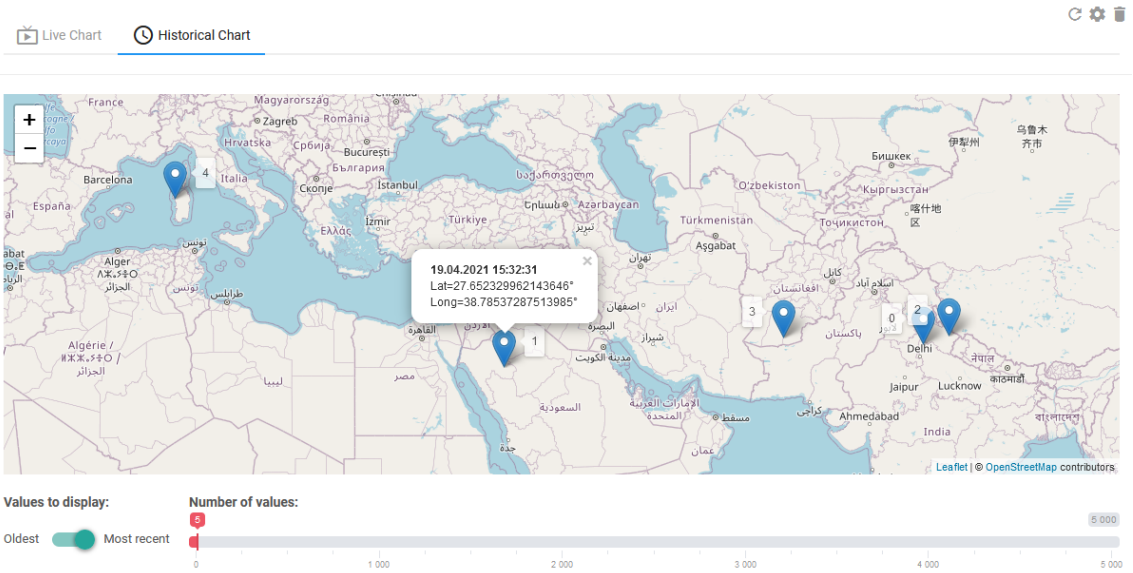

Abbildung 4.19.: Visualisierungsmodul zur Anzeige von Geo-Koordinaten auf einer Karte. Es werden jeweils nur so viele Datenpunkte angezeigt, wie mittels der unteren Slidebar eingestellt.

an. Klickt man mit der Maus auf eine Markierung, öffnet sich ein Textfeld, bei dem oben die genaue Zeit angezeigt, die dem entsprechenden ValueLog zugeordnet ist, sowie darunter die genauen Koordinaten. Der Kartenausschnitt passt sich automatisch so an, dass alle Sensorwerte zu sehen sind unter Anwendung der größtmöglichen Vergrößerung. Abbildung 4.19 zeigt die historische Kartenansicht mit visualisierten Beispieldaten.

Die Ansicht für die Echtzeitdaten der Karte kann maximal nur einen Sensorwert anzeigen, um die aktuelle Position des Sensors zu markieren. Ansonsten ist die Darstellung analog zu der historischen Kartenanzeige.

4.5.6. Persistierung von Visualisierungszuständen

Um zusätzlich die Nutzerfreundlichkeit der Visualisierungsansichten zu erhöhen, werden alle Visualisierungseinstellungen, die der Nutzer für eine IoT-Objekt-Detailansicht definiert, persistent in der Datenbank gespeichert. Das erspart den Nutzern, beim Aktualisieren der Seite oder bei erneutem Login, erneut alle vorher bereits definierten Detailansichten einstellen zu müssen. Die Einstellungen werden immer dann gespeichert, wenn der Nutzer ein Modul hinzufügt, entfernt oder die zur Anzeige verwendeten JsonPaths ändert. Geladen werden sie automatisch bei jedem Aufruf der Detailansicht. Zu diesem Zweck wird für jedes Modul eine eindeutige Identifikationsnummer generiert. Gemeinsam mit ihr wird der Visualisierungsname, der Name der verwendeten Datenfeldsammlung und alle Datenfeldnamen-JsonPath-Zuordnungen festgehalten. Die Identifikationsnummer dient zur eindeutigen Zuordnung eines Moduls, vor allem, wenn Daten zu diesem aktualisiert

werden müssen. Alle Daten werden direkt in dem Datenbank-Dokument des jeweiligen IoT-Objekts abgelegt. Für Sensoren also in der Sensor-Collection und Aktuatoren in der Aktuatoren-Collection. Diese Dokumente werden ohnehin jedes Mal beim Aufruf der Detailansicht vom Frontend angefordert, wodurch die Visualisierungs-Daten auch immer dann zur Verfügung stehen.

4.6. IoT-Regeln mit komplexen IoT-Daten

In Kapitel 2.1.2 wurde vorgestellt, wie die MBP die Definition und Auswertung von IoT-Regeln ermöglicht, mithilfe zugrunde liegender CEP-Technologien. Im Zusammenhang mit der Verarbeitung von IoT-Daten sind dabei drei notwendige Aktionen im Speziellen von Bedeutung. Dazu gehört die Erstellung eines Eventtyps, das Erstellen von EPL-Event-Abfragen und das Erstellen des Datenformats, das der Esper-Engine erlaubt, atomare IoT-Objekt-Events zur komplexen Eventdetektion zu verwenden. Die Unterstützung komplexer Daten erfordert Anpassungen in diesen drei Bereichen, die in den nächsten Unterkapiteln vorgestellt werden.

4.6.1. Anpassung der Eventtyp-Registrierung

Event-Typen dienen dazu, der Esper-Engine bekannt zu machen, welche atomaren Events berücksichtigt werden können, um diese innerhalb größerer komplexeren Events zu nutzen. Ein Eventtyp enthält dabei immer die Namen der Datenfelder des Events sowie deren Datentyp. Dieser Eventtyp kann der Esper-Engine entweder mittels einer EPL-Abfrage unter Angabe von Feld- und Datentyp-Namen erstellt werden, wie in Kapitel 2.1.2 vorgestellt, oder indem ein Pfad zu einer Java-Klasse angegeben wird, die vom Eventtyp abgebildet werden soll [Esp16]. In der bisherigen Lösungen der MBP wurden die zwei Eventdatenfelder für den Sensorwert und die Ankunftszeit stets als einzelne Felder vom Typ Double und Long als Eventtyp angegeben. Nun besteht hier mit der Einführung komplexer Daten das Problem, dass die Daten im Allgemeinen nicht als Datenfelder, gespeichert auf einer einzigen Objekt-Ebene vorliegen, sondern beliebig komplex verschachtelt sein können, inklusive primitiver Datentypen, die womöglich nicht von der Esper-Engine unterstützt werden. Deshalb ergibt es Sinn, sich zunächst anzuschauen, welche Möglichkeiten die Esper-Engine zur Modellierung komplexer Daten anbietet. Dabei wird sich im Folgenden stets auf die Esper-Dokumentation in [Esp16] berufen, wenn sich auf den Esper-Funktionsumfang bezogen wird.

Verfügbare Esper-Datentypen

Esper unterstützt die primitiven Datentypen *String*, *Boolean*, *Integer*, *Long*, *Double*, *Float* und *Byte*. Bis auf *Float* sind diese Datentypen also analog durch die zulässigen Typen der Datenmodelle repräsentierbar. Andersherum betrachtet fehlt Esper jedoch die Möglichkeit, mit Datumsangaben als Eventtyp-Datenfeld umzugehen. Deshalb werden die *Date*-Werte der ValueLogs für die Esper-Engine konventionell zu *Long*-Werten konvertiert, die das Datum und die Zeit in POSIX-Zeit angeben. Die MBP definiert zum leichteren Umgang eine Java-Enumeration mit allen verfügbaren CEP-Datentypen. In Abbildung 4.3 ist sichtbar, dass die Datenmodelle eine ähnliche Enumeration für die Definition der Datenmodelltypen verwenden. Um das geeignete Mapping zwischen den Datenmodelltypen und den CEP-Typen herzustellen, wird deshalb der Datenmodelltyp-Enumeration, genannt `IoTDataTypes`, zusätzlich eine Referenz auf Enumeration-Einträge der CEP-Typen hinzugefügt.

Um komplexe Eventtypen definieren zu können, unterstützt Esper Objekt- und Array-Strukturen. Der Zugriff auf verschachtelte Objekte kann dann mittels Angabe des Objektpfads in Form einer Punkt-Namen-Notation erfolgen. Der Zugriff auf Arrays geschieht durch die Angabe des Array-Indexes in eckigen Klammern. Es gibt mehrere Optionen, wie diese Objektstrukturen Esper bekannt gemacht werden können. Beispielsweise zählt dazu die Verwendung von Java-Maps, Java-Object-Arrays oder ein automatisches POJO-Mapping.

Finden eines geeigneten Esper-Abbildungsformats für ValueLogs

Da Esper ebenso wie die Datenmodell-Definition Objekte und Arrays als Datenstruktur unterstützt, könnte man denken, dass sich über eine analoge Verschachtelung dieser die komplexen IoT-Daten innerhalb von ValueLogs abbilden lassen. Das würde einen Zugriff auf die Datenfelder der Eventtypen im Wesentlichen analog zu den bereits verwendeten JsonPath-Abfragen ermöglichen, mit dem Unterschied, dass das Wurzelzeichen `$` weggelassen wird und nur die Punkt-Notation als Namenstrennzeichen erlaubt ist. Das große Problem hierbei ist jedoch, dass Esper keine mehrdimensionalen Arrays als Datenstruktur unterstützt. Dies steht zwar so nicht explizit in der Esper-Dokumentation, wird dort aber auch nicht als mögliches Anwendungsbeispiel aufgeführt. Außerdem haben für diese Arbeit durchgeführte Tests zur Definition multidimensionaler Arrays mit Esper diesen Verdacht erhärtet. Somit ist nicht in allen Fällen eine direkte Analogie von Datenmodell-Strukturen und Eventtyp-Strukturen gegeben. Man könnte zwar ähnliche Speicherstrukturen für mehrdimensionaler Array kreieren, indem man mehrere Objekte mit jeweils einem Arrayfeld verschachtelt, jedoch gestaltet sich der Zugriff auf das mehrdimensionale Array dann anders. Dies geschähe nämlich unter Angabe eines weiteren Objekt-Namens, für den es keine Repräsentation im Datenmodell gäbe. Dies ist nicht nur zusätzlich umständlich für die interne Handhabung in der Anwendungslogik, sondern ist auch weniger nutzerfreundlich. Denn die Erstellung von EPL-Abfragen zur komplexen Eventdefinition findet durch den User über das Frontend der MBP statt, der so unter Verwendung der manuellen Abfragendefinitions-Otion über diese komplexere Zugriffssyntax informiert sein müsste.

Um diesem Problem zu begegnen, wird sich für die Anpassung der MBP hier für ein, konzeptionell weniger elegantes, jedoch dafür nutzerfreundlicheres Konzept entschieden. Statt die Objekt- und Array-Datenstrukturen von Esper zu nutzen, werden alle primitiven Datenfelder auf der gleichen obersten Verschachtelungsebene des Eventtyps gespeichert. Die Namensschlüssel dieser Datenfelder werden dabei auf die eindeutigen JsonPath-Ausdrücke festgelegt, die bei Auswertung zu dem jeweiligen einzelnen Datenfeld in der JSON-Repräsentation der ValueLogs führen würde. Für Arrays bedeutet das, dass alle Array-Indizes-Kombinationen berechnet werden müssen, damit jedes einzelne Array-Element einem einzigen Datenfeld zuordenbar ist. Dieses Vorgehen hat gegenüber der Vorgehensweise mit echten komplexen Datenstrukturen zwei Vorteile. Der erste ist, dass auf diese Weise aus Nutzersicht auch das Verwenden von multidimensionalen Arrays für die IoT-Regeln möglich ist. Der zweite ist, dass JsonPath eine Syntax hat, die der Nutzer bereits aus den Visualisierungsansichten kennt. Somit ist die JsonPath-Syntax prinzipiell die einzige ValueLog-Zugriffsart, die er kennen muss, um alle etwas fortgeschrittenen Funktionalitäten der MBP nutzen zu können.

Registrierung von Eventtypen

Um auf die im letzten Abschnitt beschriebene Art und Weise die komplexen ValueLog-Datenstrukturen auf Eventtypen abzubilden, sind einige Berechnungsschritte notwendig. Zunächst wird über den Datenmodellcache für jedes zu registrierende IoT-Objekt das Datenmodell des Operators angefordert. Anschließend wird für jeden Blattknoten des Datenmodells ausgenutzt, dass in ihm bereits erweiterte JsonPaths gespeichert sind, wie sie schon für die Visualisierungsberechnungen benötigt wurden. Für jeden Blattknoten wird dann überprüft, ob Array-Knoten im Baumpfad zur Wurzel existieren und wenn ja, alle Index-Kombinationen der Arrays mithilfe der vorgegebenen Array-Größen berechnet. Die Syntax der erweiterten JsonPaths hilft anschließend, die jeweiligen JsonPath-Zeichenketten für das Blatt an den entsprechenden Array-Index-Stellen zu bearbeiten, um sie mit konkreten Indexangaben zu füllen. Diese Vorgehensweise wird aus Gründen der Konsistenz auch für eindimensionale Arrays umgesetzt, die eigentlich mit den Esper-Datentypen abbildbar wären. Sollte kein Array-Knoten im Baumpfad des Knotens existieren, kann der JsonPath direkt als Schlüssel für den Eventtyp verwendet werden. Um mögliche Namenskonflikte mit Sonderzeichen oder EPL-Schlüsselworte zu vermeiden, werden alle JsonPath- Datenfeldschlüssel umklammert mit ``-Apostrophen angegeben.

Neben einem Schlüssel braucht das Datenfeld noch eine Typangabe. Diese ergibt sich für jeden Knoten direkt aus der Angabe des Knoten-Typs, der wie oben schon beschrieben, ein direktes Mapping zu seinen entsprechenden CEP-Typen gespeichert hat. Eine Ausnahme bildet hierbei der Knoten-Typ Date, der als Long registriert wird. Listing 4.3 zeigt ein Beispiel-EPL-Statement zur Erstellung eines Eventtyps für ein IoT-Objekt, das das gezeigte Datenmodell von Abbildung 4.12 nutzt.

Listing 4.3 Beispiel eines EPL-Statements zur Erstellung eines Eventtyps in der MBP für das Datenmodell aus Abbildung 4.12.

```
    CREATE SCHEMA sensor_6038e55b178a3c5f509e8127(time long,
    `$['obj1'][d1']` double, `$['obj1'][d2']` double,
    `$['obj1'][0][0]` integer, `$['obj1'][0][1]` integer, ..., `$['obj1'][4][3]`
integer, ...)
```

Listing 4.4 Beispiel eines EPL-Statements zur Detektion von Events in der MBP mit komplexen IoT-Daten.

```
    SELECT * FROM pattern
    [every((event_0=sensor_6038e55b178a3c5f509e8127(`$['temp']['degree']` > 20) ->
    event_1=sensor_6038e15a178a3c5d509e2345 WHERE timer:within(2000)))]
    WHERE ((event_0.`$['humidity']` <= 50) AND (event_1.`$['temp']['num']` = 232))
```

4.6.2. Erstellung von EPL-Abfragen zur Eventdetektion

Das Erstellen der EPL-Abfragen, um komplexe Events auf Basis der atomaren IoT-Objekts zu erstellen, mit denen IoT-Regeln umgesetzt werden können, passt sich entsprechend auf die angewandten Änderungen der Eventtypen an. Statt der Konvention zu folgen, dass Sensorwerte mit EPL immer unter Angabe des `.value`-Schlüssels abgerufen werden können, muss jetzt der JsonPath zu den einzelnen Datenfelder der komplexen IoT-Daten angegeben werden, so wie sie von der Anwendungslogik als Eventtypen registriert wurden. Listing 4.4 zeigt beispielhaft für zwei atomare Sensorevents, wie das Beispiel-EPL-Statement von Listing 2.3 angepasst werden könnte, unter der Annahme, dass für die jeweiligen IoT-Objekte entsprechende Eventtypen bereits definiert wurden.

Da die EPL-Abfragen im Frontend der MBP primär mittels einer interaktiven Benutzeroberfläche erstellt werden, sind auch dort Anpassungen zur Unterstützung des Abfragens komplex verschachtelter Daten notwendig. Dazu wird an allen Stellen, an denen Bedingungen für Sensorwerte formuliert werden müssen, die JsonPath-Eingabemethode vorgeschaltet, die bereits für die Sensorwertvisualisierung in Unterkapitel 4.5.4 vorgestellt wurde. Da der Nutzer diese Eingabemethode bereits kennt, steigert dies die allgemeine Nutzerfreundlichkeit. Abbildung 4.20 zeigt beispielhaft die Eingabemethode beim Erstellen einer `WHERE`-Bedingung für ein EPL-Statement. Bei der Sensorwertvisualisierung wurden sogenannte `PathObjects` genutzt, um dem Frontend die jeweiligen JsonPaths für die JsonPath-Auswahl anzubieten. Für die IoT-Regeln wird in die Datenmodell-Dokumente der Datenbank für jedes Datenmodellknotenblatt des Datenmodellbaums ein `PathObject` hinterlegt, mit allen relevanten semantischen Informationen, unter anderem mit den erweiterten JsonPaths für die JsonPath-Eingabemethode.

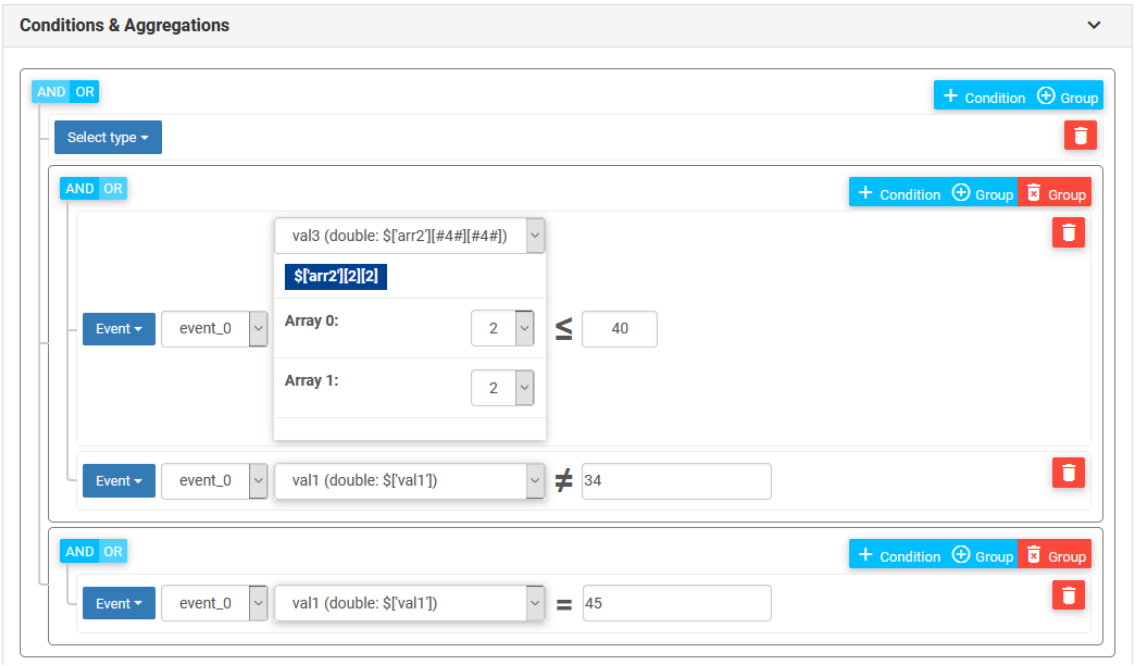

Abbildung 4.20.: Definition komplexer Bedingungen für IoT-Regeln mittels der Benutzeroberfläche. Neu ist die Möglichkeit, auf komplexe Daten eines Sensors zugreifen zu können, indem die jeweiligen Datenfelder mittels der JsonPath-Eingabemethode ausgewählt werden.

Alle eingetragenen Bedingungswerte werden für den Eingabebereich des `WHERE`-Statements anhand des Datenfeld-Typs direkt in der Benutzeroberflächenlogik validiert. So wird beispielsweise eine Fehlermeldung angezeigt, wenn ein Nutzer versucht, in ein Boolean-Feld eine Zeichenkette ungleich `true` oder `false` einzugeben. Auch ist die Typenberücksichtigung für die letztendliche Generierung des EPL-Statements notwendig. Denn Strings müssen stets in `""`-Klammern angegeben werden, um für Esper als solche interpretiert zu werden. Das Hinzufügen dieser Anführungszeichen wird von dem Benutzeroberflächenwerkzeug automatisch unternommen, wenn ein Datenfeld des Typs String ausgewählt wurde. Eine weitere Erweiterung, die jedoch im Rahmen dieser Arbeit nicht implementiert wird, ist, die erlaubten Operatoren auf Feldern je nach Datentyp einzuschränken. Denn beispielsweise wird ein Größer-Als-Vergleich nicht für String-Typen in Esper unterstützt. In der derzeitigen Version der MBP muss der Nutzer dies selbst beachten, auch wenn der Versuch, eine fehlerhaftes EPL-Statement zu definieren, von der Anwendungslogik mittels der esper-internen Statement-Validierung abgefangen und dem Nutzer rückgemeldet wird.

Eine weitere notwendige Änderung an der Benutzeroberfläche ist das Einstellen von Filtern für atomare Sensorevents. Am Beispiel von Listing 4.4 ist das die `>20`-Bedingung, die dafür sorgt, dass nur solche Events weiter berücksichtigt werden, die an der spezifizierten Datenfeldstelle einen Wert größer als 20 haben. In der alten MBP-Version stand für den Filter ausschließlich ein einzelnes konfigurierbares Feld zur Verfügung. Mit der Unterstützung komplexer Daten ist es jedoch denkbar, dass mehrere Datenfelder gleichzeitig als Filter

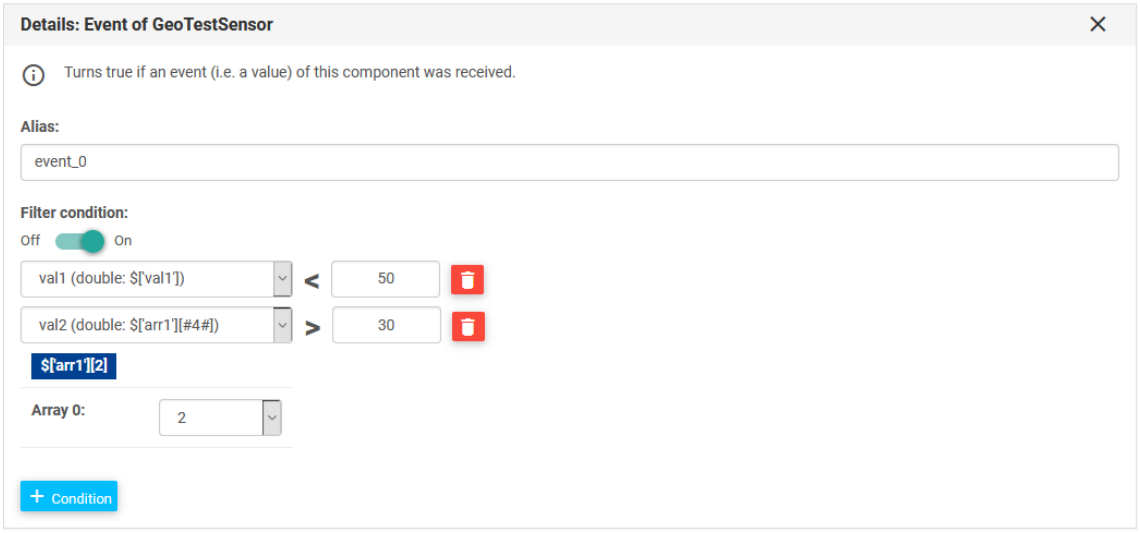

Abbildung 4.21.: Eingestellte Filter für einen Sensor bei der Erstellung von IoT-Bedingungen über das entsprechende Benutzeroberflächen-Werkzeug.

eingesetzt werden sollen. Dazu wurde die Option geschaffen, beliebig viele Filter für ein Eventtyp hinzufügen zu können. Dabei kann jedes vorhandene Sensor-Datenfeld ausgewählt werden und mittels der Vergleichsoperatoren =, !=, >, >=, < und <= Bedingungen formuliert werden. Mehrere Bedingungen werden dabei stets mit einem logischen Und (`AND` in EPL-Syntax) verknüpft. Zwar wären auch andere logische Operatoren möglich, jedoch wurde diese Entscheidung mit der Intention gefällt, dass Filterbedingungen und `WHERE`-Statements so für den Nutzer besser unterscheidbare Konstrukte darstellen. Filterbedingungen sollten eher gewählt werden, wenn Wertbedingungen über einzelne Felder eines einzigen Sensorevents formuliert werden sollen, ohne komplexe Bedingungen mit etwaigem Einbezug anderer Sensorevent-Datenfelder. Für alle anderen komplexeren Bedingungen sollten eher die `WHERE` -Statements verwendet werden, da hierfür auch das EPL-Abfragen-Modellierungstool die entsprechenden Optionen vorsieht und auch die Werte anderer Sensoren einbezogen werden können. Abbildung 4.21 zeigt ein Beispiel für eingestellte Filter eines Sensorevents.

4.6.3. Konvertierung ankommender ValueLogs zu CEP-Events

Als letzte anzupassende Abhängigkeit des CEP-Systems der MBP bleibt das Erstellen von CEP-Events aus ValueLogs, um die Esper-Engine über ankommende IoT-Daten ihrer jeweiligen registrierten Eventtypen zu informieren. In Grundlagen-Kapitel 2.1.2 ist bereits beschrieben worden, dass das CEP-Event im Wesentlichen eine Java-Map ist, die gemäß der Eventtypen-Definition alle eventrelevanten ValueLog-Informtionen kapselt, vor allem also die einzelnen Datenwerte der spezifizierten Datenfelder. Mit der Einführung komplexer Daten müssen nun alle Sensordaten von ihrer komplexen Struktur in eine unverschachtelte Java-Map umgewandelt werden, mit den jeweiligen JsonPath-Pfaden als Datenfeld-Schlüssel, wie sie für die Eventtypen definiert wurden.

Damit das in effizienter und einfacher Art und Weise umgesetzt werden kann, wird ein neuer Service eingeführt, der sich speziell mit der Umwandlung von ValueLogs zu besagten CEP-Events beschäftigt, genannt *CEP-ValueLog-Parser*. Dabei kommt vor allem der in Kapitel 4.2.2 beschriebene Datenzugriff auf *Documents* der ValueLogs zum Einsatz. Für jedes Datenfeld, das der CEP-Event-Map hinzugefügt werden muss, kann im Voraus, etwa schon während der Eventtypen-Registrierung, ein Zugriffsobjekt erstellt werden, das genau für dieses Datenfeld vorkonfiguriert ist. Dieses Zugriffsobjekt kann dann in einer Liste aller benötigten Datenfelder für jede einzelne Eventtyp-ID im Service abgespeichert werden. Da die Eventtyp-ID aus der eindeutigen Objekt-ID der IoT-Objekte hervorgeht, kann dann für jeden Sensorwert, der die MBP-Plattform erreicht, überprüft werden, ob ein Eventtyp vorliegt und wenn ja, die ValueLog-Leseanleitungen für diesen ValueLog in dem *CEP-ValueLog-Parser* herausgesucht werden. Anschließend wird iterativ jede Leseanleitung in Form der *Document*-Zugriffsobjekte, die wiederum einem JsonPath-Schlüssel des Eventtyps zugeordnet sind, auf das *Document* des ValueLogs angewandt und so der gewünschte Sensorwert gelesen. Dieser wir dann unter dem jeweiligen JsonPath-Schlüssel einem CEP-Event hinzugefügt, dass dann an die Esper-Engine zur weiteren Verarbeitung weitergeleitet werden kann.

Abbildung 4.22 ist ein Schaubild zur Veranschaulichung dieser Berechnungsschritte. Es ist dabei als Erweiterung von Schaubild 2.7 zu sehen, bei welchem das CEP-Event direkt vom CEP-Trigger-Service erstellt wurde. Nun übernimmt der *CEP-ValueLog-Parser* das Erstellen des CEP-Events unter Verwendung des Eventtyp-Leseanleitungs-Caches, der im rechten Block der Abbildung zu sehen ist. Dort ist für ein Beispiel angegeben, wie die Eventypen-IDs auf Leseanleitungslisten abgebildet werden, die wiederum für jeden einzelnen JsonPath-String ein entsprechendes vorkonfiguriertes Zugriffsobjekt speichern. Der Vorteil dieser vorinitialisierten Zugriffsobjekte, genannt `DocumentReader` ist, dass alle notwendigen Initialisierungsschritte jeweils nur einmal durchgeführt werden müssen und danach, wie schon in Kapitel 4.2.2 beschrieben, für den Lese- und Schreibzugriff auf beliebige *Document*-Objekte verwendet werden können. Dadurch wird die Gesamtperformanz der Umwandlungsschritte von komplexen Daten zu einer unverschachtelten Map verbessert, als wenn statt des Cachings jedes Mal bei der Ankunft neuer IoT-Daten erneut solche Leseobjekte erstellt werden müssten.

Soll ein Eventtyp wieder entfernt werden, etwa weil beispielsweise die zugehörige digitale Repräsentation des IoT-Objekts aus der MBP entfernt wurde, so bietet auch der Leseanleitungs-Cache die Funktion, Eventtypen wieder zu entfernen.

4.7. Anpassung des IoT-Environment-Modellierung-Tools

Die notwendigen Anpassungen zur Unterstützung von Environment Models, die bereits in Kapitel 2.1.2 vorgestellt wurden, beschränken sich im Wesentlichen auf Frontend-Anpassungen. Im Server-Backend genügt es, die neu definierte ValueLog-Version, mit den komplexen Sensordaten, mittels SSE an das Frontend des Klienten zu senden.

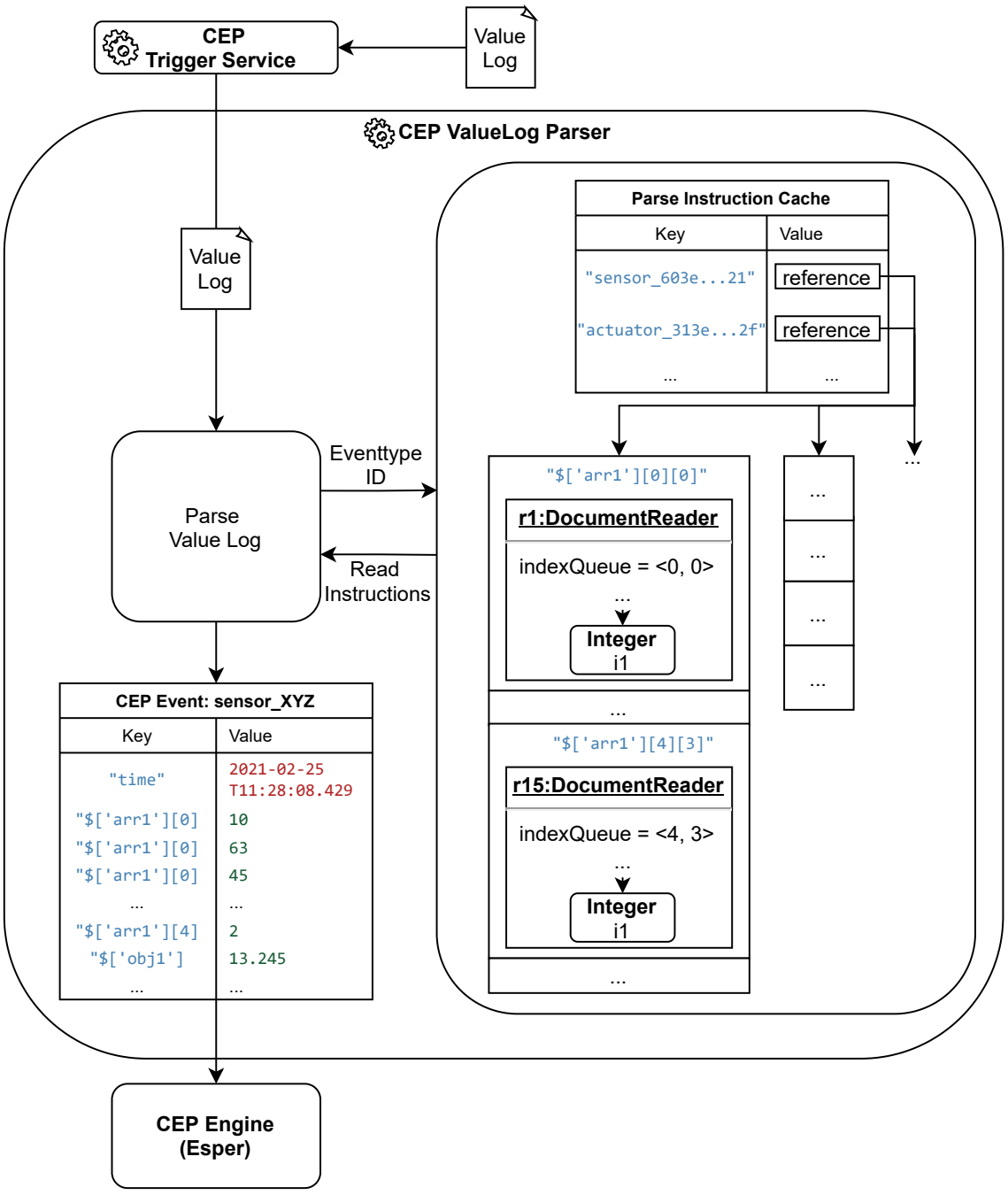

Abbildung 4.22.: Schematische Darstellung der Funktionsweise des *CEP-ValueLog-Parsers*. Aus einem ValueLog mit komplexen Daten wird ein CEP-Event in Form einer unverschachtelten Java-Map. Das Schema ist als Erweiterung des CEP-Event-Teils von Abbildung 2.7 zu sehen. Die Beispieldatenfelder orientieren sich am Datenmodell von Abbildung 4.12.

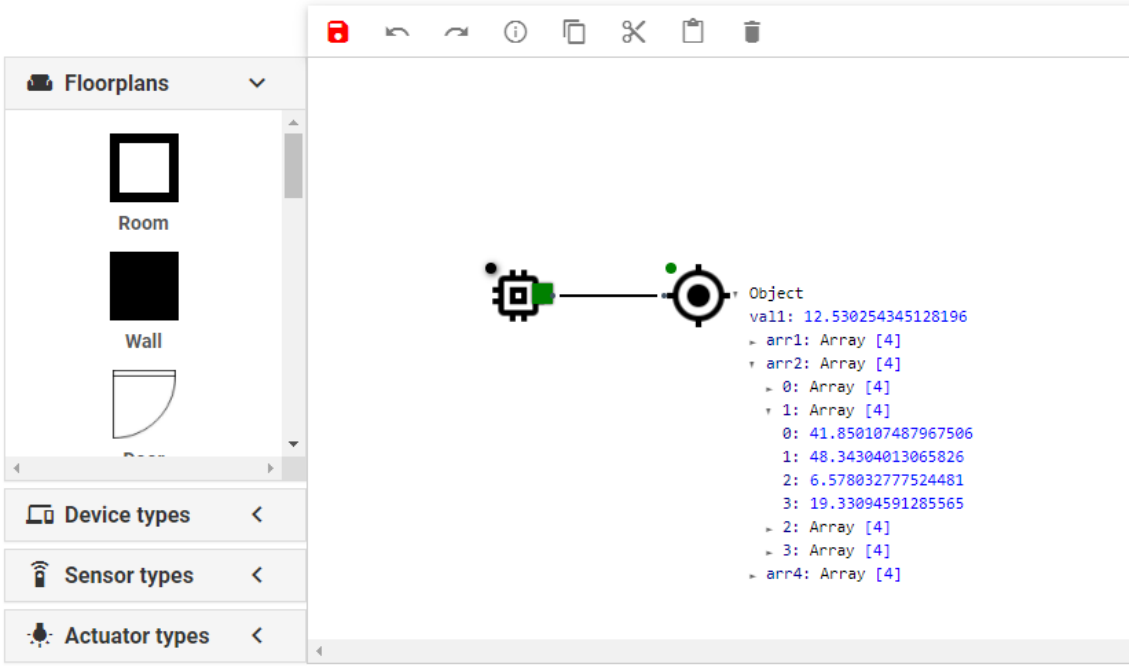

Abbildung 4.23.: Anzeige komplexer Sensordaten in der Ansicht eines Environment Models. Die mit Pfeilen markierten Baumknoten sind ausklappbar, um verschachtelte Daten anzeigen zu können.

Im Frontend sind Anpassungen notwendig, wenn es um die Darstellung einzelner Sensorwerte der modellierten IoT-Objekte geht. Statt einzelner Zahlenwerte müssen nun, sobald ein ValueLog mittels SSE an das Frontend übermittelt wurde, die komplexen Daten dieses ValueLogs visualisiert werden. Dazu wird die gleiche JSON-Objekt-Anzeige verwendet, wie sie schon bei der Visualisierung einzelner ValueLog-Datensätze in der Detailansicht der IoT-Objekte zum Einsatz kam (siehe Kapitel 4.5.3). Die komplexen Daten werden als ihre JSON-Repräsentation in einer aufklappbaren Baumansicht angezeigt, was in Abbildung 4.23 für eine beispielhaft modellierte IoT-Umgebung zu sehen ist. In der alten Version des Modelling-Tools verschwinden die eingeblendeten Sensorwerte nach kurzer Zeit, um anschließend neuere Sensorwerte anzeigen zu können. Für komplexe Daten ist dieses feste Zeitlimit aus Nutzerperspektive hinderlich, falls der dargestellte Datensatz zu groß ist, als dass er innerhalb dieser Zeitgrenze in Gänze gelesen werden kann. Um diesem Problem zu entgegnen, wurde die Erweiterung hinzugefügt, dass Sensorwerte mindestens so lange zu sehen sind, wie der Nutzer mit der Maus über die Baumanzeige fährt. Dadurch ist es für ihn möglich, den Sensorwert so lange wie notwendig zu begutachten und somit auch alle Zweige des JSON-Baumes durchzugehen. Nach wenigen Sekunden, in der die Maus still steht oder sich nicht mehr über der Anzeige befindet, wird der Sensorwert ausgeblendet und somit Platz für die nächste Anzeige ankommender Sensordaten geschaffen.

4.8. Anmerkungen zur Rolle von Monitoring Operatoren

Die im Hauptteil dieser Arbeit genannten Anpassungen der MBP, bezüglich der Einführung eines Datenmodellkonzepts zur Unterstützung komplexer IoT-Daten, beziehen sich stets auf die Operatoren zur Anbindung von Sensoren und Aktuatoren. Daneben gibt es in der MBP, wie in Kapitel 2.1.2 vorgestellt, jedoch noch Monitoring Operatoren, die zur direkten Überwachung angebundener Geräte dienen. Für diese Art von Operatoren, die eine Sonderrolle in der MBP einnehmen, wurde entschieden, die Notwendigkeit einer Datenmodelldefinition zu ersparen. Dafür gibt es vordergründig zwei Gründe. Der erste Grund ist, dass Monitoring Operatoren primär dafür gedacht sind, Geräteeigenschaften mit unmittelbaren Messwerten zu überprüfen, beispielsweise die Prozessortemperatur, ohne zusätzliche Sensoren anbinden zu müssen. Dies erfordert in den erdachten Anwendungsfällen lediglich ein Datenfeld zur Speicherung einer Gleitkommazahl. Der zweite Grund liegt darin, dass es viele zusätzliche Benutzeroberflächen-Abhängigkeiten der Monitoring-Operator-Ansichten zu den Sensorwerten existieren, die alle für komplexe Sensordaten umgebaut werden müssten. Der dadurch entstehende zusätzliche Entwicklungsaufwand wurde daher bewusst für diese Arbeit eingespart, um den zeitlichen Rahmen nicht zu sprengen, auch wenn die Unterstützung von Datenmodellen konzeptionell analog zu den Sensor- und Aktuator-Operatoren umsetzbar wäre.

Das hat zur Folge, dass für MQTT-Nachrichten, die sich durch einen `MONITORING`-Komponententyp auszeichnen, stets Sonderbehandlungen für Monitoring Operatoren überall dort im Backend notwendig sind, wo eigentlich die Datenmodelle für gewöhnliche Operatoren zum Einsatz kämen. Die Annahme der hierbei gefolgt wird, ist, dass Monitoring-Operatoren ihren Sensorwert in einem JSON-Objekt namens `value`, mit dem Schlüssel `value`, gespeichert haben. Das entspricht einer ähnlichen Handhabung, wie sie vor den Änderungen an der MBP vorlag, mit dem Unterschied eines zusätzlichen JSON-Objekts. Datenmodelle auch für Monitoring Operatoren zu unterstützen, stellt eine mögliche Erweiterung der MBP dar, die im Anschluss an die Änderungen durch diese Arbeit unternommen werden könnten, sollte dies gewünscht sein.

5. Zusammenfassung und Ausblick

Das gesetzte Ziel der Arbeit, Anpassungen an der MBP zu unternehmen, die die Unterstützung komplexer IoT-Daten ermöglichen, konnte erreicht werden. Die Operatoren der MBP wurden um das zentrale Konzept der Datenmodelle erweitert, die der Plattform die Datenstrukturen bekannt machen, die von angebundenen Geräten an die MBP gesendet werden. Bei dem Datenmodell handelt es sich um eine n-äre Baumstruktur, bei der die Knoten einzelne komplexe oder primitive Datentypen abbilden und dessen valide Gesamtstruktur von der Typisierung der einzelnen Knoten abhängt. Nutzer der MBP-Plattform können Datenmodelle mittels eines grafischen Benutzeroberflächenwerkzeuges erstellen und ihren jeweiligen IoT-Geräten zuweisen, deren Operatorenskripte anschließend datenmodellkonforme Nachrichten an die MBP senden müssen.

Es wurde gezeigt, wie das Datenmodell in effizienter Weise zur Validierung von Nachrichten und zur Typenkonvertierung generischer JSON-Typen herangezogen werden kann, um sowohl eine objektorientierte Datenrepräsentation in der Java-Anwendungslogik der MBP als auch eine analoge Speicherstruktur in der dokumentenbasierten MongoDB, zu erhalten. IoT-Daten werden unter Verwendung des Bucket Patterns in der Datenbank abgespeichert, um einen möglichst effizienten Datenzugriff zu gewährleisten, unter Berücksichtigung der technischen Besonderheiten der MongoDB und der potenziell großen Anzahl zu verwaltender Sensordaten. Für die Datenrepräsentation in der Anwendungslogik wurde ein Konzept entwickelt, das einen generischen Datenzugriff in Form von Lese- und Schreiboperationen basierend auf einzelner Datenmodellknoten als Leseschlüssel realisiert. Auf diese Weise können komplexe Daten in Abhängigkeit ihres Datenmodells generisch und autonom verarbeitet werden.

Ein Einsatzgebiet dieses Datenzugriffs sind die IoT-Regeln der MBP, die so angepasst wurden, dass nun jedes verfügbare Datenfeld eines komplexen Sensordatums für Bedingungen zur komplexen Eventdetektion verwendet werden können. Konkret können Nutzer der Plattform mittels einer neu entwickelten JsonPath-Eingabemethode verfügbare Datenfelder ihrer angebundenen IoT-Objekte auswählen und so komplexe Regeln definieren.

Eine weitere Verwendung von JsonPath findet sich in der Visualisierung von Sensordaten in der MBP, die im Rahmen dieser Arbeit in großem Umfang umgebaut wurde. Die fest vorgegebenen Visualisierungsansichten der MBP wurden von einem modularen Konzept aus frei konfigurierbaren Visualisierungsmodulen abgelöst. Das Datenmodell wird hier eingesetzt, um Vorberechnungen zur Visualisierungseignung zu unternehmen, die dem

Nutzer Konfigurationsentscheidungen abnehmen. Zur Visualisierung steht ein Linien-Diagramm und eine Kartenansicht zur Verfügung. Zuletzt wurden auch die Environment Models der MBP angepasst, die nun in der Lage sind komplexe Sensorwerte darzustellen.

Ausblick

Die getroffenen Anpassungen an der MBP ermöglichen weitere Erweiterungen und Verbesserungen, die nicht durch diese Arbeit umgesetzt werden konnten, jedoch für zukünftige Entwicklungsarbeiten in Erwägung gezogen werden können. Eine entsprechende Ideensammlung wird in diesem Abschnitt vorgestellt.

Unterstützung großer Binärdateien In Kapitel 4.4 wurde festgestellt, dass ein einzelnes Sensordatum eine maximale Speichergröße von ungefähr 200 Kilobytes haben darf, um die maximal zulässige Größe der MongoDB-Dokumente von 16 Megabytes nicht zu überschreiten. Diese Einschränkung macht es für die derzeitige Version der MBP unmöglich, größere Binärdateien beispielsweise in Form von Audio- oder Videodateien zu verwalten. Zur Speicherung solcher großer Binärdateien müsste deshalb entweder direkt das Dateiensystem des Servers verwendet oder eine geeignete Speicherkonvention großer Dateien innerhalb der MongoDB genutzt werden. Eine solche Konvention ist die sogenannte GridFS-Spezifikation der MongoDB, die große Binärdateien in eine Sammlung kleiner MongoDB-Dokumente, genannt Chunks, aufteilt und für jede große Datei ein zusätzliches Dokument anlegt, das die enthaltenen Chunks referenziert [Cho13]. Die vorhandenen Datenbanktreiber der MongoDB unterstützen Funktionen, die die GridFS-Konvention umsetzen [Cho13]. Zur Unterstützung großer Binärdateien in der MBP muss also überprüft werden, ob sich das angewandte Bucket Pattern für Zeitreihendaten mit einem entsprechendem Chunking-System in Einklang bringen lässt oder alternativ ein neues Chunking-Konzept basierend auf dem Bucket Pattern konzipiert und entwickelt werden muss. Auch muss ein Konzept erarbeitet werden, wie der Nutzer oder die Operatorskripte dem System mitteilen können, dass einzelne Zeitreihendaten als Gesamtdatei zu interpretieren sind. In diesem Zusammenhang der Übertragung von Binärdateien können oder müssen auch andere Nachrichtenprotokolle in Erwägung gezogen werden, wie zum Beispiel das Real Time Streaming Protocol (RTSP).

Stärkerer Einbezug semantischer Datenmodellinformationen Mit dem Konzept der Datenmodellbäume ist es möglich, für jeden Baumknoten beliebige semantische Informationen zu den modellierten Daten zu speichern. In der derzeitigen Version der MBP wird jedoch nur von der optionalen Angabe von Einheiten Gebrauch gemacht, die ausschließlich bei der Sensorwert-Anzeige in der Benutzeroberfläche Verwendung findet. An dieser Stelle sind vielseitige erweiterte Anwendungsfälle semantischer Informationen denkbar. Beispielsweise könnte der bereits existente Einheiten-Umrechnungsservice der MBP genutzt werden, um auch komplex verschachtelte Zahlen in kompatible Einheiten umrechnen zu

können. Weiter wäre es beispielsweise denkbar, auf Basis der bereitgestellten semantischen Informationen Vorentscheidungen bei der Visualisierung der Sensordaten zu treffen. Zum Beispiel könnte der MBP bereits so bekannt gemacht werden, dass Daten als geografische Koordinaten zu visualisieren sind, wenn zuvor entsprechende semantische Annotationen definiert wurden.

Unterstützung komplexerer JsonPath-Ausdrücke Für die Definition von CEP-Regeln und die Konfiguration von Visualisierungsmodulen wäre als zusätzliche Erweiterung der JsonPath-Eingabemethode denkbar, dass auch komplexere JsonPath-Ausdrücke, wie zum Beispiel der Einsatz von Array-Slices und Filteroperationen unterstützt werden. Bei den CEP-Regeln müsste dazu zusätzlich die Implementierung der Anwendungslogik angepasst werden, um einen erweiterten JsonPath-Zugriff auf die einzelnen unverschachtelten Datenfelder des CEP-Event-Objekts zu ermöglichen.

Größere Auswahl an Visualisierungsmodulen Die derzeit vorhandenen Visualisierungsmodule können um weitere erweitert werden, um beispielsweise auch nicht-numerische Datentypen anzeigen zu können.

Gemeinsame ValueLog-Abfrage für Visualisierungsmodule Statt dass jedes Visualisierungsmodul die zu visualisierenden Daten einzeln vom Server abfragt, können für alle Module, einer Art Observer-Pattern folgend, die Daten einmal abgefragt werden und anschließend zur Auswertung der jeweiligen JsonPaths an die Module verteilt werden.

Anpassungen des Testing Tools Wie in Kapitel 2.1.2 erwähnt, wurde das Testing Tool für die Anpassungen der MBP im Rahmen dieser Arbeit nicht behandelt. Damit dieses auch nach der Einführung des Konzepts der Datenmodelle wieder funktioniert, müssen vor allem Anpassungen an der Benutzeroberfläche und bei der Generierung des Testreports getroffen werden.

Literaturverzeichnis

[Bra+14] T. Bray et al. *The javascript object notation (json) data interchange format*. 2014. DOI: 10.17487/RFC7159 (zitiert auf S. 26, 51, 53, 54, 56).

[Cho13] K. Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media, Inc., 2013 (zitiert auf S. 21, 110).

[Esp16] EsperTech Inc. *Esper Reference - Version 5.4.0*. 2016. URL: http://esper.espertech.com/release-5.4.0/esper-reference/pdf/esper_reference.pdf (zitiert auf S. 32, 99).

[Fou] openHAB Foundation e.V. *OpenHab*. URL: https://www.openhab.org/ (zitiert auf S. 44).

[Fri19] J. Friesen. „Extracting JSON Values with JsonPath". In: *Java XML and JSON*. Springer, 2019, S. 299–322 (zitiert auf S. 39).

[G K02] C. N. G. Klyne. *Date and Time on the Internet: Timestamps*. Hrsg. von Network Working Group. Juli 2002. URL: https://tools.ietf.org/html/rfc3339 (zitiert auf S. 117).

[GBF+16] J. Guth, U. Breitenbücher, M. Falkenthal, F. Leymann, L. Reinfurt. „Comparison of IoT platform architectures: A field study based on a reference architecture". In: *2016 Cloudification of the Internet of Things (CIoT)*. IEEE. 2016, S. 1–6 (zitiert auf S. 17).

[GGPO15] C. Győrödi, R. Győrödi, G. Pecherle, A. Olah. „A comparative study: MongoDB vs. MySQL". In: *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*. 2015, S. 1–6. DOI: 10.1109/EMES.2015.7158433 (zitiert auf S. 80).

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994, 293ff. ISBN: 9780321700698 (zitiert auf S. 27).

[Goo21] Google Cloud. *Google Cloud IoT Solutions*. 4. März 2021. URL: https://cloud.google.com/solutions/iot (zitiert auf S. 41, 42).

[Hic09] I. Hickson. „Server-Sent Events". In: *W3C Working Draft WD-eventsource-20091222, latest version available at https://www.w3.org/TR/eventsource/* (2009) (zitiert auf S. 35).

[IPV] IPVS – University of Stuttgart. *Multi-purpose Binding and Provisioning Platform (MBP)*. URL: https://github.com/IPVS-AS/MBP (zitiert auf S. 19).

[Jay] Jayway by Devoteam. *Jayway JsonPath - A Java DSL for reading JSON documents*. URL: https://github.com/json-path/JsonPath (zitiert auf S. 39).

[Jos06] S. Josefsson. *RFC4648: The Base16, Base32, and Base64 Data Encodings*. Internet Engineering Task Force - Networking Group. Okt. 2006 (zitiert auf S. 55).

[KM95] P. Kilpeläinen, H. Mannila. „Ordered and unordered tree inclusion“. In: *SIAM Journal on Computing* 24.2 (1995), S. 340–356 (zitiert auf S. 83, 84).

[Lig17] R. A. Light. „Mosquitto: server and client implementation of the MQTT protocol“. In: *Journal of Open Source Software* 2.13 (2017), S. 265 (zitiert auf S. 23).

[Luc98] B. Luckham David C; Frasca. „Complex event processing in distributed systems“. In: *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford* 28 (1998), S. 16 (zitiert auf S. 31).

[Lut13] M. Lutz. *Learning python: Powerful object-oriented programming*. O'Reilly Media, Inc., 2013, S. 301 (zitiert auf S. 66).

[Mat20] H. I. Mat Keep. *Performance Best Practices: MongoDB Data Modeling and Memory Sizing*. MongoDB, Inc. 2020. URL: https://www.mongodb.com/blog/post/performance-best-practices-mongodb-data-modeling-and-memory-sizing (zitiert auf S. 21).

[MG11] P. M. Mell, T. Grance. „The NIST definition of cloud computing“. In: *National Institute of Science and Technology, Special Publication* 800 (2011), S. 145 (zitiert auf S. 41).

[Mic] Microsoft Azure. *Digital Twins Definition Language (DTDL)*. URL: https://github.com/Azure/opendigitaltwins-dtdl/blob/master/DTDL/v2/dtdlv2.md (zitiert auf S. 43, 46).

[Mic21] Microsoft. *IoT Plug and Play documentation*. 2021. URL: https://docs.microsoft.com/en-us/azure/iot-pnp/ (zitiert auf S. 41, 43).

[Mona] MongoDB. *Mongo-Java-Driver 3.6.0 API: Class binary*. URL: https://mongodb.github.io/mongo-java-driver/3.6/javadoc/org/bson/types/Binary.html (zitiert auf S. 55).

[Monb] MongoDB Inc. *BSON specification*. URL: http://bsonspec.org/ (zitiert auf S. 21, 55, 81).

[Monc] MongoDB, Inc. *Class Document*. URL: https://mongodb.github.io/mongo-java-driver/4.1/apidocs/bson/org/bson/Document.html (zitiert auf S. 69, 75).

[Mon20] MongoDB, Inc. *The MongoDB 4.4 Manual*. 2020. URL: https://docs.mongodb.com/manual/ (zitiert auf S. 21, 77).

[Mon21] MongoDB, Inc 2008-present. *BSON Types*. 2021. URL: https://docs.mongodb.com/manual/reference/bson-types/ (zitiert auf S. 53, 54).

[MSK19] C. M. de Morais, D. Sadok, J. Kelner. „An IoT sensor and scenario survey for data researchers“. In: *Journal of the Brazilian Computer Society* 25.1 (2019), S. 1–17 (zitiert auf S. 17).

[OAS14] OASIS. „MQTT Version 3.1.1“. In: *Oasis Standard* (29. Okt. 2014) (zitiert auf S. 23, 27).

[Oraa] Oracle. *Class SimpleDateFormat. Date and Time Patterns*. URL: https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html (zitiert auf S. 117).

[Orab] Oracle. *Java SE 11 & JDK 11 - Class String*. URL: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html (zitiert auf S. 54).

[Pet17] D. Petković. „JSON integration in relational database systems“. In: *Int J Comput Appl* 168.5 (2017), S. 14–19 (zitiert auf S. 80).

[PRS+16] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, D. Vrgoč. „Foundations of JSON schema“. In: *Proceedings of the 25th International Conference on World Wide Web*. 2016, S. 263–273 (zitiert auf S. 57).

[RB16] S. Rautmare, D. Bhalerao. „MySQL and NoSQL database comparison for IoT application“. In: *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*. IEEE. 2016, S. 235–238 (zitiert auf S. 80).

[SHS+20] A. C. F. da Silva, P. Hirmer, J. Schneider, S. Ulusal, M. T. Frigo. „MBP: Not just an IoT Platform“. In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2020, S. 1–3. DOI: 10.1109/PerComWorkshops48775.2020.9156156 (zitiert auf S. 17, 19, 31).

[The21] The Apache Software Foundation. *Apache Avro™ 1.10.2 Documentation*. 2021. URL: https://avro.apache.org/docs/1.10.2/ (zitiert auf S. 61).

[VF13] O. Vermesan, P. Friess. *Internet of things: converging technologies for smart environments and integrated ecosystems*. River publishers, 2013 (zitiert auf S. 17).

[VG11] A. Vukotic, J. Goodwill. *Apache Tomcat 7*. Springer, 2011 (zitiert auf S. 22).

[Vie17] A. Viebrantz. *Build a Weather Station using Google Cloud IoT Core and MongooseOS*. Medium.com. 16. Okt. 2017. URL: https://medium.com/google-cloud/build-a-weather-station-using-google-cloud-iot-core-and-mongooseos-7a78b69822c5 (zitiert auf S. 42).

[VMw21] VMware, Inc. *What Spring can do*. 2021. URL: https://spring.io/ (zitiert auf S. 22).

[Wal16] C. Walls. *Spring Boot in action*. Manning Publications, 2016 (zitiert auf S. 22).

[Wal19] R. Walters. *Time Series Data and MongoDB: Part 2 – Schema Design Best Practices*. MongoDB, Inc. 2019. URL: https://www.mongodb.com/blog/post/time-series-data-and-mongodb-part-2-schema-design-best-practices (zitiert auf S. 28, 30).

Alle URLs wurden zuletzt am 28. 04. 2021 geprüft.

A. Tabellen

A.1. Zulässige Formate für Zeichenketten des Datentyps Date

Datums-/Zeitmuster	Beispiel
dd-MM-yyyy	10-03-2021
yyyy-MM-dd	2021-03-10
dd.MM.yyyy	10.03.2021
yyyy-MM-dd HH:mm:ss	2021-03-10 10:43:50
yyyy-MM-dd HH:mm:ssXXX	2021-03-10 10:43:50-08:00
yyyy-MM-dd HH:mm:ss.SSS	2021-03-10 10:43:50:421
yyyy-MM-dd HH:mm:ss.SSSXXX	2021-03-10 10:43:50-08:00
yyyy-MM-dd'T'HH:mm:ssXXX	2021-03-10T10:43:50-08:00
yyyy-MM-dd'T'HH:mm:ss.SSSXXX	2021-03-10T10:43:50:421-08:00
dd.MM.yyyy HH:mm:ss	10.03.2021 10:43:50
dd.MM.yyyy HH:mm:ss.SS	10.03.2021 10:43:50:421
dd.MM.yyyy HH:mm:ssXXX	10.03.2021 10:43:50+04:00
dd.MM.yyyy HH:mm:ss.SSSXXX	10.03.2021 10:43:50:421+04:00

Tabelle A.1.: Übersicht über alle erlaubten Zeichenkettenformate für Datums-, beziehungsweise Zeitangaben für Datenmodelle der MBP. Die Muster folgen der Syntax, wie von Oracle in [Oraa] definiert. Angaben zu Zeitzonen, repräsentiert durch das xxx-Pattern, folgen der ISO8601 [G K02].

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift