

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

Maverick Studer

Technical Report – STL-TR-2021-04 – ISSN 2364-7167



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

Maverick Studer: Konzeption und Implementierung einer Microservices-Architektur zur Sensordatenverarbeitung und -aggregation für IoT-Anwendungen
Technical report id: STL-TR-2021-04

First published: December 2021

Last revision: September 2021

Internal review: Markus Esch

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

Title image source: ©colourbox.de, <https://www.colourbox.de/bild/iot-internet-york-bild-27985693>



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. <http://creativecommons.org/licenses/by-nc-nd/4.0/>

Bachelor-Thesis

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der Hochschule für Technik und Wirtschaft des Saarlandes

im Studiengang Praktische Informatik

der Fakultät für Ingenieurwissenschaften

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

vorgelegt von

Maverick Studer

betreut und begutachtet von

Prof. Dr. Markus Esch

Saarbrücken, 20. September 2021

Selbständigkeitserklärung

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

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

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

Saarbrücken, 20. September 2021

Maverick Studer

Zusammenfassung

Von Jahr zu Jahr führt das Wachstum des Internet of Things zur Erfindung moderner Technologien. Wie lassen sich diese nutzen, um Daten aus heterogenen Sensornetzen zu verarbeiten? Welche Herausforderungen ergeben sich für Systeme mit verschiedensten Sensoren und wie kann man diesen begegnen? Die Thesis beantwortet diese und viele weitere Fragen, welche bei der Konzeption eines Systems zur Sensordatenverarbeitung auftreten.

Das Ziel der Arbeit ist die Findung einer Architektur zur Verarbeitung von Sensormesswerten und der Bildung von Aggregaten. Möglichst große Flexibilität bei der Anbindung von Sensoren stellt eine Kernanforderung dar. Die Aggregatbildung soll in Echtzeit und über längere Zeiträume erfolgen. Hierbei wird auch der Aggregation eine große Dynamik zugesichert. Die Aggregate können einfache Berechnungen oder auch komplexe Formeln darstellen. Es werden Konzepte sowie bestehende Lösungen vorgestellt und anhand der Anforderungen bewertet. Anschließend wird die Umsetzung des Konzepts basierend auf einer Microservices-Architektur beschrieben. Einen hohen Stellenwert hat die Zielplattform Kubernetes, welche der Orchestrierung von Containern dient. Es werden die Vorteile von Kubernetes als Plattform für ein System bestehend aus zusammenarbeitenden Microservices herausgestellt. Ein großer Fokus wird auf die Darstellung der einzelnen Microservices und deren Zusammenwirken gelegt. Dadurch kann ein tieferes Verständnis für das Architekturkonzept sowie dessen Vor- und Nachteile gewonnen werden. Ebenso werden verwendete Technologien, deren Anwendung sowie relevante Implementierungsdetails dargestellt. Beispielsweise wird erläutert, wie Prometheus und Grafana zur Visualisierung der gewonnenen Daten eingesetzt werden kann. Zum Schluss erfolgt eine Evaluation der erreichten Lösung anhand eines Benchmarkings der Microservices.

Danksagung

Ich danke Herrn Prof. Dr. Markus Esch, der das Projekt ermöglicht und betreut hat. Insbesondere möchte ich mich für das gute Feedback in den Projekt-Meetings bedanken. Trotz der Ausnahmesituation, das Projekt vollständig von zu Hause aus durchführen zu müssen, hatte ich keine Nachteile. Durch die ausgezeichnete Betreuung und Erklärungen fühlte ich mich stets den Aufgaben gewachsen.

Ich danke ebenso Herrn Dominic Büch für die Unterstützung bei Fragen und Problemen. Immer wenn mir Aspekte unklar waren oder ich mehr zu einem Thema wissen wollte, konnte ich mich auf Herrn Büch verlassen.

Zudem möchte ich allen Korrekturlesern meiner Thesis für die Kritik und die Verbesserungsvorschläge danken.

Einen weiteren Dank möchte ich an meine Familie und Freunde aussprechen, die mich während meines Studiums und insbesondere während der Thesis begleitet haben.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Aufbau	2
1.3	Projektplanung	3
2	Technische Grundlagen	5
2.1	Microservices	5
2.1.1	Funktionsumfang	5
2.1.2	Unabhängigkeit	5
2.1.3	Widerstandsfähigkeit	6
2.1.4	Stateless und stateful Microservices	6
2.2	Container	6
2.2.1	Docker	6
2.2.2	Docker Hub	7
2.3	Kubernetes	7
2.3.1	Kubernetes Objects	8
2.3.2	Pods	8
2.3.3	ReplicaSet	8
2.3.4	Deployments	8
2.3.5	ConfigMaps und Secrets	9
2.3.6	Volumes	9
2.3.7	Cluster Networking	10
2.3.8	Services	11
2.3.9	Metrics API	11
2.3.10	Horizontal Pod Autoscaler	11
2.3.11	Minikube	12
2.4	Helm	12
2.5	RabbitMQ Message Broker	13
2.5.1	Routing mit AMQP	13
2.5.2	MQTT-Protokoll	13
2.6	MinIO	14
2.7	Memcached	14
2.8	MySQL	14
2.9	REST	15
2.10	Prometheus	16
2.10.1	Prometheus Operator	16
2.10.2	Prometheus Adapter	16
2.11	Grafana	16
2.12	Maven	17
2.13	Spring	17
2.13.1	Spring Boot	18
2.13.2	Multi-modulare Projekte	18
2.13.3	Spring Data JPA	18
2.13.4	Spring Web MVC	19

2.14	Hibernate	19
2.15	OpenAPI 3.0	19
2.16	Java Microbenchmark Harness	19
3	Analyse	21
3.1	Zielsetzung	21
3.2	Anforderungen	21
3.2.1	Funktionale Anforderungen	21
3.2.2	Nichtfunktionale Anforderungen	22
3.3	Lösungsansatz	23
3.3.1	Prototyp Plugin-Architektur	24
3.3.2	Prototyp Microservices	24
3.3.3	Vergleich der Lösungsansätze	26
4	Verwandte Arbeiten	27
4.1	Hadoop MapReduce	27
4.2	Stream Processing	28
4.3	Autoscaling	29
5	Konzeption	31
5.1	Top-Level Design	31
5.1.1	Sensoranbindung	32
5.1.2	Backend im Kubernetes-Cluster	32
5.2	Low-Level-Design	37
5.2.1	Entitätsmodellierung	37
5.2.2	Aufbau Microservices	39
5.2.3	Automatische Skalierung	40
5.2.4	Historische Aggregation	42
5.2.5	Frontend - Visualisierung durch Dashboards	43
6	Implementierung	47
6.1	Sensoranbindung für die Raspberry Pis	47
6.2	Erweiterbare Code-Basis	48
6.2.1	Entitäten und Modelle	48
6.2.2	Generalisierung grundlegender Abläufe	50
6.2.3	Konfiguration	51
6.2.4	Verarbeitungsschritte	51
6.3	Microservices zur Rohdatenverarbeitung	57
6.3.1	Nutzung der Code-Basis	57
6.3.2	Kubernetes-Deployment	57
6.4	Microservices zur Aggregation	58
6.4.1	Nutzung der Code-Basis	58
6.4.2	Aggregation gleicher Datentypen	58
6.4.3	Aggregation verschiedener Datentypen	59
6.4.4	Aggregation komplexe Formeln	59
6.4.5	Kubernetes-Deployment	59
6.5	Microservices zur historischen Aggregation	59
6.5.1	Nutzung der Code-Basis	59
6.5.2	Kubernetes-Deployment	60
6.6	Maven Archetypes zur Erstellung neuer Microservices	60

6.7	Microservice zur Bereitstellung an Prometheus	61
6.7.1	Weiterführung der Implementierung	62
6.7.2	Kubernetes Deployment	62
6.8	Microservice Monitoring	62
6.9	Hilfsmodule	62
6.9.1	Simulationsmodul	63
6.9.2	Modul zur Verwaltung von Stamm- und Metadaten	63
6.10	Deployment	64
6.10.1	Minikube Testumgebung	64
6.10.2	Produkktivsystem	65
6.11	Grafana Dashboards	65
7	Evaluation	67
7.1	Anforderungserfüllung	67
7.2	Auswertung Microservices	67
7.2.1	Rohdatenverarbeitung	69
7.2.2	Aggregation	70
7.2.3	Prometheus Exporter	72
7.2.4	Ausblick Skalierung	74
8	Zusammenfassung und Ausblick	75
8.1	Zusammenfassung	75
8.2	Ausblick	76
	Literatur	77
	Abbildungsverzeichnis	85
	Tabellenverzeichnis	86
	Listings	86
	Abkürzungsverzeichnis	87
A	Weiterführende Informationen zur Analyse	91
A.1	Use Cases	91
B	Weiterführende Informationen zur Implementierung	93
B.1	Temperature Microservice Deployment	93
B.2	Microservices ServiceMonitor	94
B.3	Code der Taupunktberechnung	95
B.4	Code der Wärmefluss- und Kondensationsberechnung	96
B.5	Deployment Prometheus Exporter Microservice	99
B.6	Übersicht Paketstruktur	100
B.7	Klassendiagramm Entitäten	101
B.8	Klassendiagramm Modelle	102
B.9	Klassendiagramm Konfiguration	104
B.10	Klassendiagramm Broker	105
B.11	Klassendiagramm Handler	106
B.12	Klassendiagramm Storage	107
B.13	Klassendiagramm Exporter	108
B.14	Prometheus Metriken	109

B.15	Grafana Dashboards	111
B.15.1	Übersicht Microservices	111
B.15.2	RabbitMQ Dashboard	113
B.15.3	Übersicht Messwerte	114
B.16	Deployment RabbitMQ Testumgebung	117
C	Weiterführende Informationen zur Evaluation	119
C.1	Benchmarking Microservice Deployment	119
C.2	Ergebnisse Benchmarking Rohdatenverarbeitung	120
C.2.1	Auszug JMH-Ergebnisdatei	120
C.2.2	Durchschnittswerte über mehrere Iterationen	123
C.3	Ergebnisse Benchmarking Prometheus Exporter	124
C.4	Ergebnisse Benchmarking Prometheus Exporter Optimierung	125

1 Einleitung

Das *Internet of Things (IoT)* erhält zunehmend Einzug in das alltägliche Leben. Nach zahlreichen Studien wird IoT als eine der wichtigsten modernen Technologien betrachtet, deren Einfluss in den kommenden Jahren nur noch weiter wachsen wird [63]. IoT umfasst bereits Bereiche wie Bildung, Transport, Industrie, Medizin, Handel und viele mehr [71]. In den letzten Jahren ist die Zahl an internetfähigen Geräten gestiegen. Für 2025 werden 75 Billionen *Smart Devices* prognostiziert [5]. Gleichmaßen sind auch IoT-Anwendungen, sogenannte *Smart Products*, immer populärer geworden. Dieser Trend soll sich nach Marktforschern auch weiterhin fortsetzen [63].

Mit dem Wachstum des Internet of Things verwandelt sich unsere Welt zunehmend in einen *Smart Planet* und unsere Städte in *Smart Cities*. Unter einer Smart City kann man einen urbanen Raum verstehen, der von Smart Devices umgeben ist und auf diesen aufbaut [8]. Das Ziel ist es, das städtische Leben zu vereinfachen, indem durch den Einsatz von Technik Abläufe automatisiert beziehungsweise optimiert werden [35]. Dabei werden drei Prinzipien verfolgt, die Gewinnung, Bereitstellung und Verarbeitung von Daten. Zur Verfügung stehende Informationen werden über Sensoren gewonnen. Durch Vernetzung werden sie dort bereitgestellt, wo sie sinnvoll eingesetzt werden können. Um einen möglichst großen Wissensgewinn zu erzielen, werden die Daten zusätzlich verarbeitet und analysiert [51].

Auch die Hochschule für Technik und Wirtschaft des Saarlandes (htw saar) beschäftigt sich mit der Erforschung und dem Einsatz von IoT-Technologien. In diesem Rahmen wurde vom Labor für Verteilte Systeme, auch *Distributed Systems Lab (DSL)* genannt, das *Smart City Testfeld* ins Leben gerufen. Das Testfeld soll ein städtisches Umfeld simulieren. Hier finden bereits diverse Roboter, Sensoren und Anwendungen ihren Einsatz. Durch aufbauende Forschungsprojekte sollen weitere zentrale Anwendungsszenarien von Smart Cities abgebildet und erforscht werden.

1.1 Motivation

Die Thesis beschäftigt sich mit der Konzeption und Implementierung eines Systems zur Verarbeitung von Messwerten von Smart Devices. Es ist eine Architektur aufzubauen, die Daten von IoT-Sensoren in Echtzeit aggregieren kann. Das System soll eine wichtige Rolle im Testfeld spielen. Zukünftige Arbeiten können es nutzen oder darauf aufbauen. Es ist wichtig, eine möglichst erweiterbare sowie generische Lösung zu erstellen, sodass verschiedenste Sensortypen angebunden werden können. Hierbei handelt es sich um eine zentrale Anforderung, da insbesondere bei einem Testfeld häufig Änderungen vorgenommen werden.

Das Forschungsgebiet IoT ist sehr modern. Es werden ständig neue Technologien bereitgestellt, welche für eine Umsetzung infrage kommen. Es soll gesichtet werden, inwiefern der neuste Stand der Technik eine Problemlösung erleichtert. Gleichmaßen sollen die Vorteile der neuen Technologien herausgestellt werden. Durch die Recherche von verwandten Arbeiten zeigte sich, dass diese nicht eine Mehrheit der Gesamtanforderungen abdecken. Somit konnte keine der gefundenen Lösungen zur Umsetzung verwendet werden. Dementsprechend war die Ausarbeitung einer eigenen Lösung notwendig.

1.2 Aufbau

Die Arbeit erläutert die wesentlichen Schritte, die zur Konzeption und Implementierung durchgeführt wurden. Dabei wird auf die einzelnen Projektphasen eingegangen.

In Kapitel zwei werden erstmals wichtige technische Grundlagen erläutert, die zum Verständnis der Thesis notwendig sind. Hier werden die verwendeten Technologien erläutert. Es erfolgt eine Darstellung der Prinzipien von *Microservices*. Anschließend werden Informationen zu *Containern* und *Kubernetes* dargelegt, welche die Bereitstellung der Anwendungen betreffen. Es wird die Funktionalität des *Message Brokers* erläutert, der die Verteilung der Daten zwischen den Anwendungen übernimmt. Eine Erklärung der verwendeten Protokolle wird gegeben. Ebenso werden die Technologien zur Datenhaltung, *MinIO*, *MySQL* und *Memcached*, angesprochen. Eine weitere wichtige Technologie ist *Representational State Transfer (REST)*. Die Grundlagen dieser Technologie werden beschrieben. Ebenfalls wird auf *Prometheus* und *Grafana* eingegangen. Hierbei handelt es sich um die Anwendungen zur Visualisierung der erhobenen Daten. Zum Schluss werden die wichtigsten Frameworks zur Implementierung vorgestellt.

Das dritte Kapitel befasst sich mit der Zielsetzung und der Analyse der Anforderungen. Die Anforderungen werden kategorisiert und priorisiert. Im Anschluss wird beschrieben, wie der Lösungsansatz erarbeitet wurde. In diesem Kontext wird auf die zwei grundsätzlichen architekturellen Möglichkeiten eingegangen, die *Microservices-Architektur* und die *Plugin-Architektur*. Es wird zu jedem Ansatz kurz der jeweilige Prototyp vorgestellt. Anschließend erfolgt ein Vergleich beider Ansätze.

Die verwandten Arbeiten werden im vierten Kapitel beschrieben. Hier wird auf zwei Technologien eingegangen, die zu einer Umsetzung der Lösung infrage gekommen wären. Es wird erläutert, welche Anforderungen mit diesen jedoch nicht umsetzbar sind. Ebenso wird eine eigene Entwicklung gerechtfertigt. Außerdem wird eine Arbeit erläutert, die zur Umsetzung einer automatischen Skalierung als Referenz verwendet wurde.

Im fünften Kapitel wird auf die Konzeption eingegangen. Es wird sich also mit der Umsetzung der Anforderungen befasst. Zuerst wird das *Top-Level-Design* erläutert, welches sich mit der Abbildung des Systemaufbaus beschäftigt. Hier wird die Interaktion der Komponenten erläutert und deren Funktionsumfang festgelegt. Im nächsten Schritt wird das *Low-Level-Design* ausgeführt. Der Aufbau und die Funktionalität der einzelnen Komponenten wird dargestellt. Die Abbildung der Entitäten wird beschrieben. Anschließend werden weitere Anforderungen konzipiert. Als Erstes wird hier das Autoscaling erläutert. Dabei handelt es sich um die automatische Anpassung der Architektur an Änderungen bezüglich der Anzahl eingehender Sensormessungen. Danach wird ein Plan für die Abbildung der historischen Aggregation der Daten dargestellt. Zum Schluss wird erklärt, wie die Visualisierung durch Dashboards realisiert werden kann.

In Kapitel sechs wird die Implementierung vorgestellt. Die Umsetzung der Anforderungen wird dargelegt. Zuerst wird die gemeinsame erweiterbare Code-Basis beschrieben. Im Anschluss wird jeweils die Entwicklung der einzelnen *Microservices* erläutert. Hier wird insbesondere die Nutzung der Code-Basis zur Minimierung des Implementierungsaufwands dargestellt. Zudem wird auf die Integration der Komponenten in die Testbeziehungswise Produktivumgebung eingegangen. Es wird die Bereitstellung über *Kubernetes* illustriert. Der grundlegende Gedankengang wird aufgezeigt, während wichtige Implementierungsdetails explizit angesprochen werden.

Das siebte Kapitel befasst sich mit der Evaluation der Lösung. Zuerst wird herausgestellt, welche Anforderungen zu welcher Zufriedenheit umgesetzt werden konnten. Dann wird die Tauglichkeit der Umsetzung dargestellt. Es werden Techniken zum Benchmarking der Anwendungen erläutert. Messungen zur Bestimmung der Performanz werden

vorgestellt und visualisiert. Zuletzt wird eine Analyse und Bewertung der erzielten Ergebnisse durchgeführt.

Abschließend erfolgt im achten Kapitel eine Rekapitulation der wichtigsten Punkte. Es wird ein Fazit gezogen und ein Ausblick für zukünftige Weiterentwicklungsmöglichkeiten gegeben.

1.3 Projektplanung

Zur Projektplanung wurden Arbeitspakete gebildet und in einem Projektstrukturplan organisiert. Es wurden die übergeordneten Projektphasen Planung, Analyse und Entwurf, Implementierung sowie Bereitstellung und Test gebildet. Dann wurden die einzelnen Arbeitspakete den Projektphasen zugeordnet. Im nächsten Schritt wurde aus dem Projektstrukturplan ein Gantt-Diagramm erstellt. Hierzu wurde die Software *GanttProject* [32] verwendet. Das Gantt-Diagramm setzt die Arbeitspakete in eine zeitliche Reihenfolge und stellt Abhängigkeiten heraus. Zudem können so auch Termine für geplante Meilensteine festgelegt werden. Es fanden alle zwei Wochen Rücksprachen mit den Betreuern statt, wo über den aktuellen Projektstatus gesprochen und Entscheidungen festgehalten wurden. Zu diesen Terminen wurde ebenfalls eine Betrachtung der Zeitplanung durchgeführt. Dadurch war es möglich bei Abweichungen vom Plan rechtzeitig zu reagieren.

2 Technische Grundlagen

Es folgt die Erklärung der technischen Grundlagen. Hierbei werden die zur Umsetzung verwendeten Technologien erläutert. Ebenso werden wichtige Prinzipien dargestellt. Als Erstes werden die Grundsätze von Microservices beschrieben. Anschließend wird auf *Container* und *Kubernetes* eingegangen. Diese Technologien sind für die Bereitstellung der Microservices von zentraler Bedeutung. Danach erfolgt die Darstellung der Anwendungen zur Datenhaltung. Hier werden *MinIO*, *Memcached* und *MySQL* angesprochen. Im Anschluss werden die wichtigsten Prinzipien von *REST* dargestellt. *REST* definiert die Kommunikation zwischen den Microservices und der Präsentationsschicht. Darauffolgend wird die Anwendung von *Prometheus* und *Grafana* zur Datenvisualisierung besprochen. Zuletzt werden Technologien zur Implementierung illustriert. Es wird *Maven* als Build-Management-Tool vorgestellt. Ebenfalls werden die Frameworks *Spring* und *Hibernate* erläutert. Der *OpenAPI 3.0* Standard zur Dokumentation von *REST*-Schnittstellen wird beschrieben. Dann erfolgt eine Erklärung von *JMH*. Dieses Framework wird zur Evaluation der Anwendungen verwendet.

2.1 Microservices

Microservices können gemeinsam eine komplexe Anwendung bilden [21]. In diesem Fall spricht man von einer Microservices-Architektur. Hierbei hat jeder einzelne Service einen klar definierten und begrenzten Funktionsumfang. Die fundamentalen Grundsätze eines Microservices werden im Folgenden dargestellt.

2.1.1 Funktionsumfang

Ein Microservice erfüllt eine einzige Aufgabe. Er stellt die Lösung eines spezifischen Problems dar. Hierbei kapselt er eine Einheit der Geschäftslogik. Dadurch wird er in seiner Komplexität begrenzt. Ein Gesamtablauf wird in logische Teilschritte zerlegt. Jeder Microservice erfüllt einen Verarbeitungsschritt. Hierbei sind Änderungen einfacher auszuführen, da sie auf Service- und nicht auf System-Ebene erfolgen.

2.1.2 Unabhängigkeit

Ein Microservice muss unabhängig von anderen Microservices verändert werden können. Es können Abhängigkeiten zu anderen Microservices bestehen. Hier muss jedoch eine lose Kopplung vorliegen. Die Koordination soll ausschließlich über Kommunikation erfolgen [70]. Das Erfüllen dieser Anforderung führt zudem zu einem weiteren Grundsatz, der Skalierbarkeit. Ein Microservice kann als modulare Einheit beliebig repliziert werden. In einer Microservices-Architektur lassen sich die einzelnen Services flexibel nach der zu verarbeitenden Last skalieren. Somit können Engpässe vermieden werden [39]. Die Unabhängigkeit verringert ebenso den Aufwand und die Schwierigkeit der Bereitstellung. Statt einer großen Anwendung werden kleine Services bereitgestellt. Updates und Patches erfolgen in der Regel für einen einzelnen Service und haben dadurch einen wesentlich kleineren Umfang [83].

2.1.3 Widerstandsfähigkeit

Ein Microservice muss widerstandsfähig sein. Ausfälle oder unvorhergesehene Ereignisse dürfen nicht zum Systemabsturz führen. Die oberste Priorität ist, die Verfügbarkeit auch im Fehlerfall zu gewährleisten [41]. Dies ist wichtig, da jeder Microservice einen kritischen Prozess in einem größeren Ablauf realisiert. Fällt ein Service aus, so kann der Gesamtablauf nicht ausgeführt werden.

2.1.4 Stateless und stateful Microservices

Microservices können entweder *stateless* oder *stateful* sein [44]. Der Unterschied liegt im Anwendungskern. Ein zustandsloser Microservice beantwortet Anfragen, ohne dass er Informationen als Zustand speichern oder abrufen muss. Jede Operation startet mit einer gleichen Ausgangslage ohne den Einfluss von zuvor erfolgten Ereignissen. Ein zustandsbehafteter Microservice muss Informationen über mehrere Anfragen hinweg persistieren, um eine Verarbeitung durchführen zu können. Es ist zu betonen, dass ein zustandsloser Microservice auch Daten von außerhalb zur Verarbeitung hinzuziehen kann. Er kann auf externe Anwendungen zugreifen, heißt Datenbanken oder sonstige Speicher anfragen. Diese Daten stellen keinen Zustand dar. Sie können auf Basis der Anfrage selektiert werden.

Stateless Microservices bieten einige Vorteile. Zustandslosigkeit fördert Widerstandsfähigkeit. Bei Fehlern oder Abstürzen kann ein Zustand verloren gehen, was die Funktionalität beeinträchtigt. Hat ein Microservice keinen Zustand, so entstehen auch keine Auswirkungen durch Neustarts. Zudem ist ein stateless Microservice beliebig horizontal skalierbar. Wird ein Zustand gehalten, so muss eine Kommunikation zwischen Replikaten erfolgen, um diesen synchron zu halten. Ohne Zustand können Anfragen an beliebige Instanzen verteilt werden. Somit wird die Lastverteilung, man spricht meist von *Load Balancing*, unkompliziert [102].

2.2 Container

Container stellen eine Form der *Betriebssystem-Virtualisierung* dar. In einem Container kann eine Anwendung beliebiger Größe ausgeführt werden. Diese wird entkoppelt von der unterliegenden Infrastruktur. Dadurch hat die Anwendung immer das gleiche Verhalten, egal wo und wann der Container ausgeführt wird [16].

Der Container beinhaltet den Code der Anwendung sowie alles, was zu deren Ausführung benötigt wird. Dabei kann es sich um die Laufzeitumgebung, Konfigurationen, Bibliotheken und Ähnliches handeln [114][25]. Ein *Container-Image* stellt ein solches Software-Paket dar. Es ist ein Abbild eines Containers und erleichtert dessen Bereitstellung. Immer wenn Änderungen an einem Container erfolgen sollen, muss ein neues Image erstellt und der Container neu erzeugt werden [16].

2.2.1 Docker

Docker ist eine *Container-Laufzeitumgebung*. Container-Images werden hier als Dateien, die sogenannten *Dockerfiles*, verwaltet. Dockerfiles können als Schablonen verstanden werden, die zur Erstellung von Containern nutzbar sind. Dabei sind sie portabel speicherbar und mehrfach anwendbar [114]. Eine wichtige Eigenschaft von Docker Images ist der Aufbau in Schichten. Jede Schicht entspricht einer Instruktion im Dockerfile [3].

In Listing 2.1 ist ein Beispiel für ein Dockerfile zu sehen. Zuerst wird eine Build Stage initialisiert, indem *OpenJDK* als Plattform genutzt wird. Anschließend wird eine Datei vom

Typ *Java Archive (JAR)* aus dem lokalen Verzeichnis in die Container-Umgebung kopiert. Hierbei handelt es sich um ein Dateiformat, welches alle kompilierten Java Klassen und deren Ressourcen zu einer ausführbaren Datei vereint. Der Befehl zum Starten der JAR wird als Hauptbefehl des Images gesetzt. Dadurch wird der Container ausführbar. Startet man einen Container mit dem Image, so führt dieser das entsprechende Java-Programm aus. Zentrale Dockerfile-Befehle sind in Tabelle 2.1 beschrieben [26].

```
1 FROM openjdk:14-alpine
2 ARG JAR_FILE=target/*.jar
3 COPY ${JAR_FILE} app.jar
4 ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Listing 2.1: Beispiel Dockerfile

Tabelle 2.1: Docker-Befehle

Docker-Befehl	Beschreibung
FROM	Initialisiert eine neue Build Stage, setzt das Standard-Image für nachfolgende Befehle
LABEL	Setzt einen Bezeichner zwecks Verwaltung
RUN	Führt Kommandos in einer neuen Schicht aus
CMD	Dient zur Ausführung der Software des Images
EXPOSE	Definiert die Ports, die zur Laufzeit abgehört werden
ENV	Setzt Umgebungsvariablen
ARG	Setzt eine Variable im Scope des Build-Prozesses
ADD, COPY	Kopiert lokale Dateien in den Container
ENTRYPOINT	Setzt den Hauptbefehl des Images
VOLUME	Macht Speicherbereiche verfügbar

2.2.2 Docker Hub

Docker Hub ist ein *Repository*, also ein Verzeichnis, für Container-Images. Es dient der zentralen Verwaltung und ermöglicht das Speichern und Teilen [24]. Man kann zuvor kompilierte Dockerfiles unter einem *Tag* ablegen. Anschließend können sie über diesen Tag abgerufen und von überall als Container gestartet werden. Docker Hub stellt ebenfalls Funktionalität zur Versionierung und zum Teilen von Images bereit.

2.3 Kubernetes

Kubernetes (k8s) basiert auf einem *Cluster Management System* namens *Borg* [105]. Es ermöglicht die Verwaltung eines Computerclusters, also einem Zusammenschluss mehrerer Rechner [79]. Kubernetes wird zur Orchestrierung von Containern verwendet. Hierbei wird die Automatisierung der Verwaltung, Bereitstellung und Skalierung realisiert [49]. Im Vergleich zu Docker, das auf einer einzelnen Maschine ausgeführt wird, ist Kubernetes eine Anwendung für eine Vielzahl an Maschinen. Es wird das Zusammenwirken einer großen Zahl an Containern auf verschiedenen Computersystemen verwaltet.

2.3.1 Kubernetes Objects

Kubernetes bildet den Zustand des Clusters anhand von Objekten ab. Diese Objekte beschreiben die laufenden Anwendungen, deren Ressourcen und Regeln. Kubernetes gewährleistet die Existenz der Objekte. Jedes Objekt hat die Eigenschaften *spec* und *status* [104]. *Spec* beschreibt die erwünschten Eigenschaften des Objekts, während *status* den aktuellen Zustand angibt. Die Erstellung von Objekten wird häufig mithilfe von *YAML-Dateien* durchgeführt. *YAML Ain't Markup Language (YAML)* [101] ist eine auf *Unicode* basierte Sprache zur Datenserialisierung. Sie findet in vielen Bereichen ihre Anwendung und wird vor allem für die Definition von Konfigurationen verwendet.

Namespaces ermöglichen die Arbeit von mehreren Nutzern oder Teams an einem oder vielen Projekten [65]. Kubernetes-Objekte können einem Namespace zugeordnet werden und sind dann nur innerhalb dieses sichtbar. *Labels* können Objekten hinzugefügt werden, um diese in Gruppen einzuteilen [50]. Dadurch wird die Identifikation und somit auch die Verwaltung der Objekte erleichtert.

2.3.2 Pods

Bei Kubernetes werden Container in *Pods* ausgeführt. Pods wiederum können auf einem virtuellen oder physischen System laufen [66]. Ein Pod stellt die kleinste verwaltbare Einheit in Kubernetes dar. Er ist als Gruppe von Containern definiert. Alle dem Pod zugehörigen Container haben den gleichen Kontext. Sie teilen sich also System- und Netzwerkressourcen sowie Speicher. Ein Pod bildet einen logischen Host ab, er umfasst also eng zusammenhängende Anwendungen. Der meistverbreitete Anwendungsfall bei der Erstellung von Pods ist das „one-container-per-pod“-Modell. Hierbei ist ein Pod als *Wrapper* um einen einzelnen Container zu verstehen. Unter *Wrapper* versteht man hierbei eine Hülle um den Container, der eine zusätzliche Abstraktionsebene bildet. Jeder Pod soll hier nur einen einzigen Container beinhalten [74].

2.3.3 ReplicaSet

Ein *ReplicaSet* ermöglicht die Verwaltung einer Gruppe von Pods. Hierbei wird insbesondere gewährleistet, dass immer eine bestimmte Anzahl an Pods verfügbar sind [80]. Die direkte Verwendung dieser Objekte ist in der Praxis eher selten. Stattdessen werden meist *Deployments* verwendet. Diese bauen auf *ReplicaSets* auf. Zudem werden zusätzliche Funktionalitäten und Verwaltungsmöglichkeiten bereitgestellt.

2.3.4 Deployments

Als *Deployment* ist ein Objekt zu verstehen, das den gesamten Lebenszyklus einer Anwendung definiert. Es wird der gewünschte Zustand beschrieben und Kubernetes kümmert sich um dessen Umsetzung [22]. In Listing 2.2 ist ein Beispiel zu sehen, welches zur weiteren Erklärung referenziert wird.

Das *Deployment* definiert die Eigenschaft *spec*. In diesem Fall wird ein *ReplicaSet* erzeugt. Hier kann über *replicas* die Anzahl an gewünschten Pods festgelegt werden. Es werden Replikate, also Kopien des gleichen Pods instanziiert. Stürzen Pods ab oder werden auf sonstige Art beendet, so reagiert Kubernetes mit dem Starten neuer Pods. Die Anzahl an Replikaten kann auch nach der Bereitstellung beliebig angepasst werden. Kubernetes macht es damit möglich eine horizontale Skalierung durchzuführen. Der *Selektor* gibt an, wie die zugehörigen Pods erkannt werden können. Die Erkennung kann über ein gesetztes Label erfolgen. Mit dem *template-Feld* wird die Definition der zu erzeugenden Pods beschrieben.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: kubedemo
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: kubedemo
10   template:
11     metadata:
12       labels:
13         app: kubedemo
14     spec:
15       volumes:
16         - name: counter-storage
17           persistentVolumeClaim:
18             claimName: counter-storage-claim
19     containers:
20       - name: kubedemo
21         image: user/kube-demo
22         envFrom:
23           - configMapRef:
24             name: app-config
25           - secretRef:
26             name: app-secrets
27         ports:
28           - containerPort: 8080
29             name: http
30         volumeMounts:
31           - mountPath: "/app/data/counter"
32             name: counter-storage

```

Listing 2.2: Beispiel Deployment

2.3.5 ConfigMaps und Secrets

Kubernetes stellt zwei verschiedene Objekte zur Verwaltung von Konfiguration bereit. Für nicht vertrauliche Daten sind *ConfigMaps* zu verwenden [15]. *Secrets* beinhalten sensible Informationen, wie beispielsweise Passwörter oder Zertifikate [85]. Diese werden verschlüsselt und somit geschützt. Die Informationen werden als Schlüssel-Wert-Paare definiert und können so später referenziert werden. Der Vorteil in der Verwendung von Objekten, um Konfiguration zu speichern, liegt in der Entkopplung von den Containern. Die Portabilität ist gewährleistet. Konfigurationen sind zentral einmal definiert und können beliebig wiederverwendet werden. Wie in Listing 2.2 dargestellt, können Pods auf Inhalte von ConfigMaps und auf Secrets über Umgebungsvariablen zugreifen. Zudem kann ein Zugriff über Aufrufparameter oder Dateien im Speicher erfolgen.

2.3.6 Volumes

Der Speicher innerhalb eines Containers ist flüchtig. Das heißt, dass Dateien, die von Anwendungen im Speicher abgelegt werden, nach der Beendigung des Containers verloren sind [108]. Um diesem Problem zu begegnen, macht Kubernetes die Definition von persistenten Speicherbereichen möglich, die über die Container- beziehungsweise Pod-Lebenszeit hinaus existieren. Man bezeichnet einen solchen Speicher als *Persistent Volume (PV)* [72]. Ein Persistent Volume kann durch einen Administrator statisch oder dynamisch über *Storage Classes* bereitgestellt werden.

2.3.6.1 Storage Classes

Eine *Storage Class* bildet eine Klasse von Speichern, die dieselben Eigenschaften aufweisen. Sie stellt unter anderem einen *Provisioner* zur Verfügung, der die Verteilung von Speicher durchführt [98]. Eine statische Bereitstellung kann über eine YAML-Datei erfolgen, wie in Listing 2.3 zu sehen ist. Das Persistent Volume muss dem Pod zugeordnet werden. Dazu wird eine Anfrage gestellt. Man spricht von einem *Persistent Volume Claim (PVC)* [72]. Sobald ein passendes Volume verfügbar wird, ordnet Kubernetes dieses dem PVC zu. Bei der dynamischen Bereitstellung erhält die Storage Class die Anfrage und erzeugt gegebenenfalls ein PV, welches dann durch Kubernetes zugeordnet wird. Wie ein zugehöriger PVC beschrieben wird, kann in Listing 2.4 nachvollzogen werden.

Ein PV kann einem Pod zugeordnet werden. Diese Zuordnung erfolgt über die Referenzierung des PVCs in der Definition des Pods. Das Einbinden von Speichern bezeichnet man als *Mounten*. Dabei wird das PV den Containern unter einem Pfad verfügbar gemacht. Die konkrete Zuweisung ist in Listing 2.2 zu sehen.

```
1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: counter-storage-volume
5    labels:
6      type: local
7  spec:
8    storageClassName: manual
9    capacity:
10     storage: 10Mi
11    accessModes:
12     - ReadWriteOnce
13    hostPath:
14     path: "/app/data/counter"
```

Listing 2.3: Beispiel PersistentVolume

```
1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: counter-storage-claim
5  spec:
6    storageClassName: glusterfs
7    accessModes:
8     - ReadWriteOnce
9    resources:
10     requests:
11      storage: 10Mi
```

Listing 2.4: Beispiel PersistentVolumeClaim

2.3.6.2 GlusterFS

GlusterFS ist ein skalierbares Netzwerk-Dateisystem [37]. Es ist für den Einsatz in einem Cluster mit mehreren Knoten konzipiert. Hierbei verbindet es die Speicher der verschiedenen Knoten zu einer Einheit. GlusterFS kann bei Kubernetes als Storage Class eingesetzt und zur dynamischen Speicherbereitstellung verwendet werden.

2.3.7 Cluster Networking

Im Cluster ordnet Kubernetes jedem Pod eine eigene IP-Adresse zu. Es werden Container-Ports von Host-Ports getrennt. Dadurch kann die Kommunikation zwischen Containern

innerhalb eines Pods erfolgen. Die Kommunikation zwischen Pods wird durch Kubernetes verwaltet [14]. Möchte man von außerhalb auf Ressourcen zugreifen, so kommen Services zum Einsatz. Die Verwendung von Services ermöglicht darüber hinaus ein *Load Balancing*, also eine Lastverteilung zwischen Pods.

2.3.8 Services

Ein Service beschreibt eine logische Gruppe von Pods und die Regeln für den Zugriff auf diese. Über einen Selektor wird die Zusammensetzung der Gruppe festgelegt. Ebenso kann dieser zur Definition des Zugriffs verwendet werden [86]. Ein Beispiel für einen Service ist in Listing 2.5 wiedergegeben.

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: kubedemo-service
5    labels:
6      app: kubedemo
7  spec:
8    type: NodePort
9    selector:
10     app: kubedemo
11    ports:
12     - port: 8080
13       name: http

```

Listing 2.5: Beispiel Service

Kubernetes ermöglicht die Definition verschiedener Service Typen [86]. Ein als *ClusterIP* definierter Service macht ihn nur von innerhalb des Clusters erreichbar. Legt man einen Service mit dem Typ *NodePort* an, so ist er über jede Node-IP auf einem speziellen Port erreichbar. Dieser Port wird zufällig zugewiesen oder kann explizit gewählt werden. Dabei liegt er stets zwischen 30000 und 32767. Ein Service vom Typ *NodePort* ermöglicht auch die Ansprache der Pods von außerhalb. Über die Definition als *LoadBalancer* bekommt der Service eine Anwendung zugeordnet, die die Lastverteilung übernimmt. Durch Zuordnung des *ExternalName*-Typs wird dem Service ein *DNS-Name* zugewiesen. Das Kubernetes *Domain Name System (DNS)* ermöglicht eine Zuordnung von sprechenden Namen zu IP-Adressen im Cluster. Es realisiert somit eine Namensauflösung.

2.3.9 Metrics API

Die *Metrics API* stellt aktuelle Metriken von Nodes beziehungsweise Pods bereit [82]. Metriken können diverse Daten abbilden. Sie können zum Beispiel Informationen über die Anwendung oder das ausführende System bereitstellen. So sind Daten wie die Anzahl an gesendeten Anfragen, die CPU-Auslastung oder der verwendete Speicher oft interessant. Die Metrics API ermöglicht ebenso die Bereitstellungen externer sowie eigener Metriken [42][19]. Diese können über den Pfad */apis/metrics.k8s.io/* abgefragt werden. Der *Metrics Server* aggregiert die Metriken für das Cluster. Er muss je nach Art des Cluster-Deployments separat aufgesetzt werden [82].

2.3.10 Horizontal Pod Autoscaler

Der *Horizontal Pod Autoscaler (HPA)* ermöglicht eine automatische horizontale Skalierung von Pods in einem *ReplicaSet*. Er arbeitet als *Kubernetes-Controller* und als *API-Ressource*. Dadurch kann er die Anzahl an benötigten Pods anhand beobachteter Metriken regulieren. In Listing 2.6 ist die Definition eines HPA zu sehen.

```
1  apiVersion: autoscaling/v2beta2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: temperature-hpa
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: temperature-deployment
10   minReplicas: 1
11   maxReplicas: 10
12   metrics:
13     - type: Object
14       object:
15         metric:
16           name: rabbitmq_queue_messages_ready
17           selector: {matchLabels: {queue: "temperature"}}
18         describedObject:
19           apiVersion: "/v1"
20           kind: Service
21           name: rabbitmq-service
22       target:
23         type: Value
24         value: 20
```

Listing 2.6: Beispiel Horizontal Pod Autoscaler

In diesem Beispiel wird ein Deployment skaliert. Es wird eine minimale und maximale Anzahl an Pods definiert. Zudem wird die zu beobachtende Metrik beschrieben. Hier zu sehen ist die Skalierung anhand des Typs *Object*, welches über den Selektor definiert wird. In diesem Fall handelt es sich um eine Ressource, die über den RabbitMQ-Service bezogen wird. Diese gibt die Größe der Queue für Temperaturmesswerte an. Alternativ können hier auch Metriken der Pods verwendet werden. Dazu muss der Typ als *Pods* statt als *Object* definiert sein. Hierbei kann der Name der referenzierten Metrik analog angegeben werden. Diese Metrik wird dann von jedem Pod des Deployments abgefragt. Es wird anschließend ein Durchschnitt über alle Werte gebildet, welcher zur Skalierung verwendet wird. Der abgefragte oder berechnete Wert wird zur Durchführung der Skalierung mit einem Zielwert verglichen. Je nach Abweichung von diesem Zielwert sorgt der HPA für das Hinzukommen beziehungsweise Wegfallen von Pods. Dadurch erfolgt eine automatische Anpassung an die aktuelle Last. Ein manueller Eingriff bleibt erspart.

2.3.11 Minikube

Minikube ist eine Implementierung eines Kubernetes-Clusters, die als lokale Testumgebung zum Entwickeln und Experimentieren dienen soll [112]. Minikube führt ein Single-Node Cluster innerhalb einer virtuellen Maschine aus. Es unterstützt die meisten Kubernetes Features. Gleichzeitig bleibt das Aufsetzen eines großen ressourcenintensiven Rechner-Clusters erspart.

2.4 Helm

Helm ist ein *Kubernetes Package Manager*. Es ist ein Tool zur Verwaltung von Kubernetes-Anwendungen [118]. Dazu nutzt Helm die sogenannten *Helm Charts*. Diese legen die Definition und Konfiguration eines Kubernetes-Deployments fest. Zudem können diese zur Verwaltung verschiedener Versionen genutzt werden. Helm ermöglicht darüber hinaus die Migration zwischen Versionen durch Upgrades und Downgrades. Die Charts

verbergen die Komplexität eines Deployments und stellen essenzielle Parameter zur Konfiguration bereit. Durch die Publikation von populärer Software als Helm Charts soll ein mühseliges Kopieren und Einfügen von Deployments vermieden werden [40].

2.5 RabbitMQ Message Broker

RabbitMQ ist eine Erlang-Implementierung eines AMQP-Brokers [27]. *Advanced Message Queuing Protocol (AMQP)* ist ein Netzwerkprotokoll auf dem *Application Layer* für *Message Oriented Middleware (MOM)*. MOM umfasst Programme, die eine Kommunikation zwischen Anwendungen durch Nachrichtenübertragung realisieren. Das AMQP-Protokoll definiert die Interaktion mit dem Broker sowie das Format von Nachrichten und Kommandos [106]. Ein Message Broker ist eine Software, die einen Austausch von Nachrichten zwischen Anwendungen ermöglicht. Man bezeichnet eine Anwendung als *Producer* oder *Publisher*, wenn sie Daten zum Broker versendet. Von einem *Consumer* oder *Subscriber* spricht man, wenn Nachrichten vom Broker empfangen werden. Die Nachrichten haben einen *Payload* und ein *Label*. Unter Payload ist der Inhalt der Nachricht zu verstehen. Über das Label wird ein Routing zu den Empfängern durchgeführt [46].

2.5.1 Routing mit AMQP

AMQP verwendet *Exchanges* und *Queues*, um ein Message Routing durchzuführen [23]. Einen Exchange kann man als Zwischenstation verstehen. RabbitMQ definiert drei verschiedene Arten von Exchanges. Man kann einen *Direct Exchange* definieren, einen *Fanout Exchange* oder einen *Topic Exchange* [27]. Ein Direct Exchange stellt ein Unicast Routing dar. Die Zuordnung erfolgt Punkt zu Punkt über das Label. Ein Fanout Exchange leitet an alle gebundenen Queues weiter. Die Verteilung ist analog zu einem Multicast [58]. Ein Topic Exchange ist auf *Publish-Subscribe-Messaging* ausgelegt. Die Weiterleitung wird basierend auf dem Abgleich zwischen dem Label und einem Pattern durchgeführt. Anschließend werden die Nachrichten einer oder mehrerer Queues zugeordnet. Über *Bindings* können Zuordnungen definiert werden. Sie legen Regeln für eine Weiterleitung fest, zum Beispiel über einen *Routing Key* [58]. Kommt eine Nachricht bei dem Exchange an, gleicht er das Label mit dem Routing Key ab. Der Abgleich zeigt für welche Queue die Nachricht bestimmt ist. Die Queues speichern die Nachrichten, bis sie von einem Consumer verarbeitet werden. Um Nachrichten zu erhalten, muss sich ein Consumer bei einer Queue registrieren.

2.5.2 MQTT-Protokoll

Message Queuing Telemetry Transport (MQTT) ist ein simples, leichtgewichtiges Nachrichtenprotokoll, welches auf dem *Publish-Subscribe-Pattern* basiert [43]. Es ist von seiner Grundfunktionalität AMQP sehr ähnlich, hat aber einen geringeren Funktionsumfang. MQTT ist besser geeignet zur Anbindung von simplen Sensoren oder Aktoren. Insbesondere unter eingeschränkten Bedingungen, wie in Netzwerken mit schlechter Konnektivität oder bei der Verwendung leistungsschwacher Hardware, kann MQTT überzeugen [52]. Ein wichtiger Unterschied ist, dass MQTT keine Exchanges und Queues verwendet. Die Weiterleitung erfolgt direkt an alle Subscriber. Dieser Vorgang ist performanter, jedoch verliert man die Möglichkeit des flexiblen Routings.

RabbitMQ kann durch das Aktivieren einer Erweiterung mit MQTT kommunizieren. Das heißt, es wird ermöglicht Clients über MQTT an den Broker anzubinden.

2.6 MinIO

MinIO ist eine Kubernetes-native *Object Storage-Lösung* [59]. Sie dient dem Speichern von unstrukturierten Daten. Es wird ein mit *Amazon S3* kompatibles *Application Programming Interface (API)* bereitgestellt [61]. *Amazon Simple Storage Service (Amazon S3)* zählt zum Umfang der *Amazon Web Services (AWS)* [13]. Hierbei handelt es sich um eine skalierbare *Object Storage-Lösung*, welche über einen *Web Service* genutzt werden kann. *MinIO* stellt einen Server, einen Client und ein *Software Development Kit (SDK)* bereit. Der Server ist so konzipiert, dass er hardwareunabhängig und skalierbar ist. Damit ist er optimal für das Deployment als Container geeignet. Er lässt sich somit auch durch Orchestrierungssoftware wie Kubernetes bereitstellen und verwalten [60]. Das SDK steht in vielen Programmiersprachen bereit und macht den Zugriff auf den *Object Storage* für Anwendungen möglich.

Man kann die angelegten Objekte über das SDK auslesen oder über das bereitgestellte Webinterface betrachten. Eine weitere Möglichkeit die Daten verfügbar zu machen, ist durch das Teilen mit *Presigned URLs*. Diese ermöglichen die gezielte Veröffentlichung einzelner Objekte. Dazu kann mit dem SDK eine *Uniform Resource Locator (URL)* generiert werden, die den Zugriff auf einzelne Datensätze ermöglicht [87]. Die URLs können je nach Definition unterschiedliche Gültigkeitsdauer haben.

JavaScript Object Notation (JSON) ist ein menschenlesbares, leichtgewichtiges Datenformat zum Austausch von Informationen zwischen Anwendungen. Es besteht aus einer Sammlung von Schlüssel-Wert-Paaren, welche auch in Listen beziehungsweise Feldern organisiert sein können [45]. *JSON* findet bei der Serialisierung, also der Abbildung von strukturierten Daten in ein serielles Format Einsatz. So kann es zum Beispiel verwendet werden, um Objekte in einem *Object Storage* als Dateien abzulegen. Client-Anwendungen müssen die Serialisierung durchführen. Bei erneutem Zugriff können die Dateien von den Clients wieder in die ursprünglichen Objekte deserialisiert werden.

2.7 Memcached

Memcached ist ein *Objekt-Caching System* [121]. Hierbei werden Daten als Schlüssel-Wert-Paare abgelegt. Da die Daten im Speicher behalten werden, ist eine hohe Zugriffsgeschwindigkeit gewährleistet. Der Speicher kann bei *Memcached* auf mehrere Server verteilt werden. Diese kennen sich jedoch untereinander nicht. Es ist keine Synchronisation zwischen den Servern notwendig [69]. Dadurch ist das System gut skalierbar. Speicherzugriffe können über Clients erfolgen. Diese müssen dazu die bereitgestellte API nutzen. Zur Vereinfachung des Zugriffs ist ein SDK bereitgestellt, welches eine Implementation von Clients ermöglicht.

2.8 MySQL

MySQL ist ein *Open-Source Relationales Datenbankmanagementsystem (RDBMS)* [64]. Insbesondere wird es häufig zur Speicherung von Daten aus *Web Services* eingesetzt. Es eignet sich durch sein Client-Server-Modell als Datenhaltung für verteilte Systeme. Relationale Datenbanken werden oft zur Persistenz von strukturierten Daten verwendet. Durch die Organisation der Daten in Tabellen und Relationen ist eine Strukturierung möglich. Die *Structured Query Language (SQL)* ermöglicht eine gezielte Abfrage von Daten.

2.9 REST

REST ist die Kurzform für *Representational State Transfer*. Hierbei handelt es sich um einen Architektur-Stil für Netzwerk-Anwendungen [29]. REST stellt Anforderungen an die Eigenschaften eines Dienstes. Roy Fielding legt im Rahmen seiner Dissertation [30] mehrere Prinzipien fest, die eine REST-Architektur ausmachen. Diese werden im Folgenden kurz erläutert.

- **Client-Server** - Der Server stellt einen Dienst zur Verfügung. Clients müssen ihm Anfragen stellen, um diesen Dienst in Anspruch zu nehmen. Es liegt also zwischen Client und Server eine Unabhängigkeit vor.
- **Zustandslosigkeit** - Eine Nachricht enthält alle Informationen, die zur Verarbeitung benötigt werden. Sowohl Server als auch Client können die Nachricht interpretieren und verarbeiten. Zwei Nachrichten sind immer komplett unabhängig voneinander. Die Verarbeitung einer Nachricht ist ein in sich geschlossener Prozess. Durch die Zustandslosigkeit wird eine horizontale Skalierung des Servers ermöglicht.
- **Cache** - Clients können Antworten in einem Cache zwischenspeichern. Falls zu einem späteren Zeitpunkt dieselbe Anfrage erneut gestellt wird, so kann die Antwort aus dem Cache genommen werden. Dadurch kann die Netzwerklast reduziert werden.
- **Uniform Interface** - Die Einheitlichkeit der Schnittstelle ist eine zentrale Anforderung. Es wird eine einfache Nutzung unabhängig von der Ausprägung der Schnittstelle garantiert. Hierzu werden Informationen als Ressourcen definiert, die über einen *Uniform Resource Identifier (URI)* identifiziert werden. Ressourcen können verschiedene Datenrepräsentationen haben. Sie können modifiziert oder gelöscht werden. Zur Manipulation von Ressourcen sind Standardmethoden zu verwenden. Die wichtigsten Methoden sind in Tabelle 2.2 aufgeführt.
- **Multi-Layer** - Das System ist mehrschichtig aufgebaut. Die Schichten sind logisch voneinander getrennt und interagieren nur über Schnittstellen. Dadurch wird eine Kapselung möglich. Die Skalierbarkeit wird zusätzlich gefördert.
- **Code on Demand** - Optional kann Code an Clients ausgeliefert werden. Dies soll nur bei Bedarf erfolgen. Clients können den Code dann ausführen, wodurch ihre Funktionalität dynamisch erweitert wird.

Tabelle 2.2: HTTP-Commands

HTTP-Command	Beschreibung
GET	Die Ressource wird vom Server angefordert.
POST	Eine neue Ressource wird eingefügt.
PUT	Die Ressource wird angelegt. Falls sie bereits vorhanden ist, wird eine Änderung ausgeführt.
DELETE	Die Ressource wird gelöscht.

2.10 Prometheus

Prometheus ist eine *Monitoring-Anwendung* für Systeme und Services [125]. Ziele zur Überwachung sind statisch konfigurierbar oder können über *Service Discovery* hinzugefügt werden [68]. Prometheus grenzt sich von anderen Monitoring-Systemen ab, da es auf einem *Pull-Mechanismus* basiert. Die meisten anderen Systeme erwarten hingegen einen *Push* der Daten. Bei einem push-basierten Konzept müssen die Anwendungen die Daten selbst einpflegen, während bei einem pull-basierten Modell die Daten aktiv von den Endpunkten abgefragt werden. Prometheus definiert ein Datenmodell basierend auf *Time Series Data*, also Zeitreihendaten. Diese können über den Namen der Metrik und Labels in Form von Schlüssel-Wert-Paaren identifiziert werden [68]. Zur Abfrage der Daten stellt Prometheus eine eigene Sprache, genannt *Prometheus Query Language (PromQL)* bereit. Diese ermöglicht die Abfrage und Aggregation von Daten in Echtzeit [77]. Ergebnisse können dann direkt über Prometheus in Tabellen oder Graphen angezeigt werden. Außerdem wird eine HTTP API bereitgestellt, die eine Auswertung durch externe Systeme ermöglicht [38][77]. Der Kern des Systems ist der *Prometheus Server*, welcher das Sammeln der Daten in festgelegten Intervallen durchführt. Ein weiterer Bestandteil sind die Client-Bibliotheken, welche es ermöglichen, selbst Metriken zu erstellen und zu publizieren [68]. Der *Push Gateway* stellt eine optionale Systemkomponente dar. Dieser ermöglicht das Einpflegen von Metriken aus kurzlebigen Jobs. Er kann also eingesetzt werden, wenn ein Pull der Daten nicht möglich ist [126].

2.10.1 Prometheus Operator

Der *Prometheus Operator* bietet eine Kubernetes-native Bereitstellung von Prometheus und dessen Komponenten [123] an. Er vereinfacht die Integration von Prometheus in ein Kubernetes-Cluster sowie das Monitoring von Kubernetes-Ressourcen. Dazu ermöglicht er insbesondere die Definition von *ServiceMonitors*. Hierbei handelt es sich um eine Ressource, die festlegt, wie ein Monitoring von Kubernetes-Services zu erfolgen hat. Der Prometheus Operator erzeugt dann basierend auf der Definition des *ServiceMonitors* Konfigurationen zur entsprechenden Abfrage der Anwendung [123].

2.10.2 Prometheus Adapter

Der *Prometheus Adapter* dient als Bindeglied zwischen Prometheus und Kubernetes. Er kann den *Metrics Server* in Clustern, die Prometheus zum Monitoring nutzen, ersetzen [119]. Die Funktion des Adapters ist die Erweiterung der Kubernetes Metrics API [76]. Hierbei werden die durch Prometheus gesammelten Metriken hinzugefügt. Diese können dann auch von anderen Kubernetes-Ressourcen abgefragt werden. So kann beispielsweise der HPA die Metriken über die API auslesen und zur Skalierung verwenden.

2.11 Grafana

Grafana ist eine Software zur Visualisierung und Analyse von Daten. Sie ermöglicht den Zugriff auf Metriken unabhängig von der Datenhaltung [34]. Mit Grafana ist es möglich, dynamische und wiederverwendbare Dashboards zu erstellen [117]. Prometheus wird häufig als Datenquelle für Grafana verwendet. Die Einbindung der Daten kann über das Grafana Webinterface durchgeführt werden. Anschließend können die Metriken nach der Prometheus Logik selektiert werden. Prometheus stellt zur grafischen Darstellung nur simple Anzeigen bereit. Mit Grafana kann eine vielfältigere und ansprechendere Visualisierung erfolgen.

2.12 Maven

Apache Maven ist ein Tool zur Verwaltung von Projekten. Hierbei realisiert es die Automatisierung des Build-Prozesses und die Verwaltung von Abhängigkeiten [57]. Maven nutzt zum Management eines Projekts eine zentrale Datei namens *Project Object Model (POM)*. Diese Datei beinhaltet Informationen in der Auszeichnungssprache *Extensible Markup Language (XML)*, welche das Build-Management konfigurieren. Dazu zählen Informationen zum Projekt selbst, zum Ablauf des Build-Vorgangs sowie benötigte Abhängigkeiten. Maven gibt eine Standard-Verzeichnisstruktur vor, welche die Navigation in komplexen Projekten vereinfacht. Außerdem legt es einen klaren Aufbau für Projekte mit mehreren Modulen und gemeinsamen Abhängigkeiten fest. Zudem wird durch die definierten Lebenszyklen eine Abbildung der Iterationen eines Softwareprojekts ermöglicht. Der *default*-Lebenszyklus besteht aus folgenden Phasen [56]:

1. **validate** - Validation der Projektstruktur
2. **compile** - Kompilieren des Quellcodes
3. **test** - Ausführen der Unit-Tests
4. **package** - Packen in eine ausführbare Datei
5. **verify** - Ausführen der Integrationstests
6. **install** - Installieren im lokalen Repository
7. **deploy** - Kopieren zum Remote-Repository

Maven kann zudem flexibel mit Plugins erweitert werden. Dadurch kann der Funktionsumfang vergrößert werden. Es steht ein breites Spektrum an Plugins bereit. Beispielsweise existiert ein Plugin zur Definition von eigenen *Maven Archetypes*. Hierbei handelt es sich um Templates zur Erstellung von Projekten. Diese definieren Dateien, welche für ein Projekt benötigt werden. Außerdem legen sie den Aufbau und die Konfiguration fest. Es können Platzhalter verwendet werden. So kann beispielsweise ein Paketname während der Erzeugung angegeben und in den Klassen eingefügt werden. Unter Verwendung des Archetypes kann ein Projekt automatisch generiert werden. Dazu muss der Archetype im lokalen Maven Repository installiert werden. Anschließend kann ein einzelner parametrisierbarer Befehl aufgerufen werden. Die Parameter werden bei der Generierung der Dateien für die entsprechenden Platzhalter eingesetzt. Maven stellt einige standardmäßige Archetypes bereit, bietet jedoch auch die Möglichkeit eigene Archetypes zu erstellen.

2.13 Spring

Spring ist ein Java-Framework, welches die Entwicklung von Geschäftslogik und Anwendungen erleichtert. Hierbei stellt Spring Funktionalität für verschiedenste Anwendungsfälle bereit [97]. Ein Kernaspekt von Spring ist die Entkopplung der Ressourcen. Dazu stellt Spring das Prinzip der *Dependency Injection* bereit. Spring verwaltet die Ressourcen und teilt sie Objekten zu, die von ihnen abhängig sind. Dadurch muss selbst keine aktive Initialisierung und Zuweisung von Ressourcen durchgeführt werden. Ein von Spring verwaltetes Objekt nennt man *Bean*. Die Verwaltung der Spring Beans übernimmt der *Inversion of Control (IoC)-Container*. Dieser führt die Erzeugung und Konfiguration der Objekte sowie die Dependency Injection durch [17]. Er verwaltet somit den kompletten Lebenszyklus der Spring Beans.

Ebenso stellt Spring viele Vorlagen bereit, die eine Arbeit mit APIs erleichtern. Es wird auch Logik für den Zugriff auf populäre Datenhaltungslösungen angeboten. Großer Wert wird auf die Kapselung von logischen Schichten gelegt. In diesem Sinn erleichtert Spring die Anwendung des *Data Access Object (DAO)*-Patterns, welches die Trennung von Datenhaltungs- und Anwendungsschicht vorschreibt [20]. Auch für die Netzwerkkommunikation bestehen Integrationen von APIs diverser Technologien. So gibt es beispielsweise auch für AMQP eine Spring-Integration, welche die Nutzung des Protokolls vereinfacht. Ein weiterer Kernaspekt von Spring ist das flexible Testen mit Mocks und dynamischen Testprofilen [97].

2.13.1 Spring Boot

Spring Boot ist eine Erweiterung des Spring Frameworks, welches der Entwicklung eigenständiger Anwendungen dient. Es wird insbesondere zur Implementierung von Microservices und sonstigen Webanwendungen eingesetzt. Spring Boot vereinfacht die Erstellung, indem es die Konfiguration der Anwendung und der Infrastruktur automatisiert [93]. Durch die *Spring Starters* bildet es typische Anwendungsfälle ab und bietet für diese vordefinierte Templates. Zudem ermöglicht es Anwendungen ohne Abhängigkeiten von Web Servern auszuführen. Web Server werden automatisch durch die Initialisierung eingebettet. Spring Boot stellt des Weiteren Lösungen für Themen, wie Sicherheit, Metriken, Monitoring und ausgelagerte Konfiguration bereit [92].

2.13.2 Multi-modulare Projekte

Ein Spring Projekt kann wiederum verschachtelt weitere Projekte beinhalten. Man spricht von einem *multi-modularen Projekt*. In der Regel beinhaltet das Elternprojekt Konfigurationen zum Gesamtaufbau und seinen Kindprojekten. Diese Konfiguration ist bei einer Verwaltung mit Maven in der POM zu finden. Die POM des Elternmoduls referenziert die Kindmodule, welche wiederum selbst durch eigene POMs beschrieben werden [62]. Hierbei muss das Elternmodul keine konkrete Anwendung darstellen, sondern umfasst lediglich zentrale Informationen und Definitionen zu Gemeinsamkeiten. Spring ermöglicht ebenso Module untereinander zu referenzieren. Dadurch kann auf Funktionen anderer Module zugegriffen werden. Dies ist insbesondere für das Schreiben von einer gemeinsamen Code-Basis oder von Hilfsbibliotheken nützlich.

2.13.3 Spring Data JPA

Spring Data Java Persistence API (JPA) ist Teil des größeren Spring Data Frameworks. Es ermöglicht die Implementation von *JPA Repositories* [96]. Diese stellen die Datenzugriffsschicht eines Projekts dar. Durch Spring Data JPA kann der Aufwand für das Schreiben von Abfragen und anderen Operationen minimiert werden [95]. Die Verbindung zur Datenhaltung wird durch das Framework realisiert. Ebenso wird die Verwaltung der *Entitäten*, also der Datenobjekte, übernommen. Außerdem wird die Definition von Abfragen über festgelegte Namenskonventionen ermöglicht. Eine konforme Benamung führt zur automatischen Erstellung der Abfrage durch das Framework. Es ist also lediglich notwendig, für eine gewünschte Funktion den der Konvention entsprechenden Namen auszuwählen. Alles Weitere erledigt der *Query Derivation Mechanismus* automatisch [94].

Für Abfragen und Operationen, die zu komplex sind, um sie mit dem Benamungsschema abzubilden, kann die *Java Persistence Query Language (JPQL)* verwendet werden. JPQL ermöglicht die Definition von Datenbankzugriffen basierend auf den vorhandenen Entitäten [100]. Dies bringt den Vorteil, dass die jeweilige Abfragesprache der Datenbank nicht

verwendet werden muss. Stattdessen werden die Daten über das Objektmodell abgefragt. Unter anderem mit dieser Funktionalität ermöglicht Spring Data den flexiblen Austausch der Datenhaltungslösung. Unabhängig von dem verwendeten System kann Code für beliebige Create, read, update and delete (CRUD)-Operationen implementiert werden. Durch die Entkopplung von spezifischen Abfragesprachen ist ein Austausch ohne große Programmänderungen möglich.

2.13.4 Spring Web MVC

Spring stellt mit dem *Web MVC Framework* Funktionalität zur Erstellung von Webanwendungen bereit [111]. Im klassischen *Model View Controller (MVC)-Pattern* wird eine Webanwendung in drei Schichten unterteilt. Die Model-Schicht befasst sich mit den anzuzeigenden Daten. Der Controller bildet die Geschäftslogik ab. Die View stellt die Ansicht der Daten dar. Spring Web MVC vereinfacht die Erstellung aller drei Komponenten. Ebenso wird Logik bereitgestellt, die den Zugriff auf Controller und sonstige Webschnittstellen erleichtert. Dazu werden vorgefertigte *REST-Clients* bereitgestellt, die als Basis zur Kommunikation dienen.

2.14 Hibernate

Hibernate ist ein *ORM-Framework* [115]. ORM steht für *Object Relational Mapping*. Bei ORM geht es um die Speicherung von Daten über die Laufzeit der *Java Virtual Machine (JVM)*, also über das Programmende hinaus [113]. Diese Persistenz erfolgt häufig in relationalen Datenbanken, wie zum Beispiel MySQL. *Hibernate* ist zudem eine Implementation der *Java Persistence API*. JPA ist eine Spezifikation, die das Speichern und Abfragen von Daten festlegt. Als Spezifikation ist JPA kein Framework, sondern benötigt einen Provider um eingesetzt werden zu können. Mit *Hibernate* als Provider wird die Entwicklung von persistenten Klassen durch das Hinzufügen von *Annotations* ermöglicht. Hierbei erreicht es eine gute Performanz, indem unter anderem auf Technologien wie *Lazy-Loading* zurückgegriffen wird. Bei *Lazy-Loading* werden nicht alle Daten sofort geladen. Stattdessen werden Daten dann geladen, wenn ein Zugriff erfolgt. *Hibernate* lässt sich außerdem gut skalieren. Es kann in Clustern mit großen Datenmengen und vielen Nutzern eingesetzt werden [115].

2.15 OpenAPI 3.0

Die *OpenAPI-Spezifikation*, auch bekannt als *Swagger-Spezifikation*, ermöglicht die Beschreibung von REST-APIs [2]. Endpunkte und deren Operationen sowie Eingaben und Ausgaben werden dokumentiert. Damit kann der korrekte Umgang mit der API gewährleistet werden. Zudem kann aus der API direkt Client Code erzeugt werden, indem *Swagger Codegen* genutzt wird [1]. Ebenfalls wird eine Oberfläche, die *Swagger UI* zum Testen und Arbeiten mit der Schnittstelle bereitgestellt. Diese ermöglicht das sofortige Ausführen von Operationen. Dabei bleibt das Schreiben von Anfragen oder die Implementierung von Clients zum Testen erspart [78].

2.16 Java Microbenchmark Harness

Java Microbenchmark Harness (JMH) ermöglicht die Implementation und Analyse von Benchmarks für die *JVM*. Es bietet für das Benchmarking unter anderem die Messung

2 Technische Grundlagen

von Gesamtdurchsatz und Durchschnittszeiten an [122]. Dazu wird eine Funktion mehrfach ausgeführt und jeweils die benötigte Zeit zur Ausführung bestimmt. Es wird hierbei garantiert, dass die Logik in der JVM ungestört ablaufen kann. Des Weiteren werden zahlreiche Konfigurationsmöglichkeiten bereitgestellt. Es ist möglich die Dauer des Tests, die Anzahl an Wiederholungen oder auch die verwendeten Systemressourcen, wie zum Beispiel die Anzahl an Threads, festzulegen. Zudem können Warmups definiert werden, welche dem System Zeit zum Anlaufen bieten. Dadurch kann eine realitätsnahe Messung gewährleistet werden. Durch Konfiguration kann eine klare Abgrenzung getätigt werden, was in die Berechnung der Ausführungszeit einfließen soll. Vor, nach oder zwischen den Messungen können Routinen ausgeführt werden, welche bei den Messungen nicht berücksichtigt werden [47]. Dadurch können Zustände gesetzt werden, welche für das Benchmarking benötigt werden, ohne dass sich dies auf die Verarbeitungszeit auswirkt. JMH ermöglicht somit das gezielte Messen von Ausführungszeiten von Programmteilen und die Evaluation der Geschwindigkeit. Es gibt dabei ebenfalls Aufschluss über die Genauigkeit der erfolgten Messungen. Dazu werden Abweichungen vom Mittel erkannt und Werte für Fehlerspannen bestimmt.

3 Analyse

Im Folgenden werden die wesentlichen Schritte zur Ausarbeitung der Projektanforderungen dargelegt. Zuerst wurde das grundlegende Projektziel definiert. Anschließend wurden die Anforderungen erarbeitet. Es wurde ein erster Lösungsansatz angefertigt. Dazu wurden zwei Prototypen erstellt und verglichen. Abschließend wurde ein Prototyp ausgewählt, welcher als Basis für die Konzeption verwendet wurde.

3.1 Zielsetzung

Das Ziel der Thesis ist der Aufbau einer Architektur zur Verarbeitung von Sensordaten. Eine grundlegende Anforderung an diese ist die möglichst flexible Erweiterung. Neue Funktionalität soll mit möglichst geringem Aufwand angebunden werden können. Es ist eine Basis zu schaffen, um neue Sensortypen mit verschiedensten Datenformaten anzubinden. Ebenso müssen Konzepte geschaffen werden, um Daten zu aggregieren. Es ist angedacht, gleiche Sensorwerte zu gruppieren und einfache Aggregate in Echtzeit zu berechnen. Darüber hinaus soll eine Gruppierung von Sensoren verschiedener Art zur Bildung neuer Messwerte realisierbar sein. Ein weiteres Ziel ist es, eine historische Datenverarbeitung zur Berechnung von zeitlichen Aggregaten zu konzipieren. Die aufbereiteten Daten sollen schließlich in einem Dashboard bereitgestellt werden.

3.2 Anforderungen

Die Anforderungen wurden über den Zeitraum der Analysephase gesammelt und ausgearbeitet. Zuerst wurden *Use Cases* erstellt. Dazu wurden Akteure und deren Aufgaben definiert. Die festgelegten Anwendungsfälle sind im Anhang A.1 ausgelagert. Im nächsten Schritt wurden aus den Use Cases funktionale und nichtfunktionale Anforderungen abgeleitet. Diese wurden wiederum in *Must-*, *Should-* und *Nice-To-Have-Anforderungen* unterteilt. Dadurch konnten die Anforderungen priorisiert und in eine zeitliche Abfolge gebracht werden.

3.2.1 Funktionale Anforderungen

Die funktionalen Anforderungen bilden die angestrebte Kernfunktionalität der Lösung ab und sind wie folgt festgehalten worden.

3.2.1.1 Must-Have-Anforderungen

Flexible Anbindung von neuen Sensortypen: Die Anbindung muss unabhängig vom Aufbau der Sensordaten sein. Für jeden Sensor müssen Rohdaten gesammelt und gespeichert werden.

Stamm- und Metadaten verwalten: Es muss zudem möglich sein, Metadaten zu Sensoren einzupflegen. Eine Gruppierung von Sensoren zwecks Berechnung von Aggregaten muss ebenso realisiert werden. Hierbei ist die Verwaltung diverser Stammdaten notwendig.

3 Analyse

Echtzeitdatenverarbeitung: Einige Aggregate sollen in Echtzeit berechnet werden. Dazu zählen Gruppierungen, also Zusammenschlüsse von Sensoren gleichen Typs. Hier ist es angedacht, eine Durchschnittsbildung sowie eine Findung von Minima und Maxima durchzuführen. Es ist außerdem die Möglichkeit offenzuhalten, andere Berechnungen auszuführen.

Historische Daten verarbeiten: Historische Daten sollen ebenfalls verarbeitet werden. Dazu soll eine separate Routine geschaffen werden, die beispielsweise am Tages-, Monats- und Jahresende alle Daten analysiert und neue zusammengefasste Werte berechnet.

Sensoranbindung: Es sollen die vorhandenen *GrovePi-Sensoren* angebunden werden. Dabei handelt es sich um die Modelle *Temperature&Humidity*, *Water* und *Airquality*. Zur Anbindung soll der bereitgestellte *Raspberry Pi* verwendet werden. Dieser kann mit den Sensoren verknüpft werden und so Daten auslesen. Aufseiten des Raspberry Pi sind somit Anwendungen zu realisieren, die die Sensoren abfragen und die Messwerte über MQTT an den Message Broker senden. Für alle vorhandenen Sensoren sind Microservices zu erstellen, die diese Sensordaten verarbeiten. Die Erstellung neuer Anwendungen soll durch die bestehende Architektur möglichst einfach gestaltet sein.

Bereitstellung der Daten: Die Rohdaten und verarbeiteten Daten müssen bereitgestellt werden, damit sie in einem Dashboard visualisiert werden können.

3.2.1.2 Should-Have-Anforderungen

Echtzeitdatenverarbeitung: Es sollen Gruppen verschiedener Sensortypen gebildet werden können. Ein Beispiel wäre hier eine Gruppierung von Temperatur- und Luftfeuchtigkeitsdaten zu Tauwerten. Dazu muss immer jeweils ein Temperatur- und Luftfeuchtigkeitssensor zu einer Gruppe zusammengefasst werden. Liegen zu beiden Sensoren Messwerte vor, so kann ein Aggregat gebildet werden. Die Erstellung weiterverarbeitender Microservices soll aufbauend auf der erweiterbaren Architektur einfach durchführbar sein.

3.2.1.3 Nice-to-have-Anforderungen

Komplexe Aggregate: Eine Abbildung komplexer Aggregate mit mehreren Sensortypen gleicher und unterschiedlicher Art sowie einer Verrechnung in Formeln könnte realisiert werden.

Monitoring: Ein Monitoring der einzelnen Microservices wäre eine sinnvolle Erweiterung. Hier wäre ebenfalls die Benachrichtigung bei Sensorausfall, im Fehlerfall oder bei Messwerten außerhalb des anzunehmenden Spektrums denkbar.

3.2.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen stellen sinnvolle Einschränkungen für das Design dar. Sinn und Zweck ist hier die Einhaltung von Qualitätsstandards.

3.2.2.1 Must-Have-Anforderungen

Flexibel neue Funktionalität anbinden: Neue Funktionalität soll in die bestehende Architektur flexibel integriert werden können. Dazu sind Microservices zu implementieren. Die Entwicklung muss hierbei mit geringem Aufwand realisierbar sein.

Bereitstellung mit Kubernetes: Die Anwendungen sind für das *DSL-Cluster* der Hochschule für Technik und Wirtschaft des Saarlandes bereitzustellen. Dementsprechend sind Kubernetes-Deployments zu erstellen.

Zustandslosigkeit: Die Anwendungen sind zustandslos zu implementieren, damit eine horizontale Skalierung realisierbar ist.

Ausgelagerte Konfiguration: Die Konfiguration muss ausgelagert werden. Sie kann beispielsweise über Umgebungsvariablen erfolgen. Dadurch wird ermöglicht, dass alle Anwendungen über eine zentrale Konfiguration verwaltet werden können. Ändern sich Adressen oder Ports von Anwendungen, so kann der Aufwand einer Anpassung gering gehalten werden.

Dashboard: Dashboards soll nach Möglichkeit mit *Prometheus* und *Grafana* erstellt werden. Dazu müssen Metriken über eine REST-API bereitgestellt werden.

3.2.2.2 Should-Have-Anforderungen

Autoscaling: Es könnten je nach Bedarf mehr Anwendungen gestartet werden, um eine höhere Systemlast zu bewältigen. Dieser Vorgang kann automatisiert erfolgen. Dabei wird die Auslastung beobachtet. Je nach Aktivitätsgrad wird die Anzahl an Programminstanzen erhöht oder reduziert.

3.3 Lösungsansatz

Das ursprüngliche Konzept zur Umsetzung plante die Entwicklung eines *Pluggable Aggregator Service* für das Smart City Testfeld. Dabei sollte es sich um einen Microservice handeln, der auf Basis einer *Plugin-Architektur* erweitert werden kann. Diese Architektur erlaubt die Erweiterung der Funktionalität des Kernmoduls durch Hinzufügen von Plugins [83]. Das Kernmodul stellt Basisfunktionalität bereit, während jedes Plugin auf dieser aufbaut. Ein Plugin kapselt eine bestimmte Aufgabe und implementiert die dafür erforderliche Logik. Der grundlegende Aufbau des Design-Patterns in Bezug auf den Anwendungsfall kann in Abbildung 3.1 nachvollzogen werden.

Durch die Verwendung einer Plugin-Architektur wollte man es ermöglichen, flexibel neue Funktionalität anzubinden. Würde man neue Sensortypen verwenden oder neue Aggregate berechnen wollen, so müsste lediglich ein neues Plugin entworfen und eingebunden werden.

Im Laufe der Analysephase wurde ein alternativer Lösungsansatz angedacht. Statt der Verwendung eines einzelnen Microservices mit Plugins könnten auch mehrere verknüpfte Microservices verwendet werden. Man spricht von einer *Microservices-Architektur*. Bei dieser Architektur wird eine Gesamtfunktionalität durch verschiedene zusammenarbeitende Services abgebildet. Hierbei hat jeder Service eine eigene Aufgabe, welche er unabhängig erfüllen kann. Die Gesamtfunktionalität wird nicht durch einen Monolith realisiert, sondern ergibt sich durch die Interaktion der Services miteinander [84].

Um eine Entscheidung zu treffen, welcher Ansatz zielführender ist, wurde jeweils ein Prototyp erstellt. Dadurch konnte ein direkter Vergleich hinsichtlich der Qualität der Umsetzungen erfolgen.

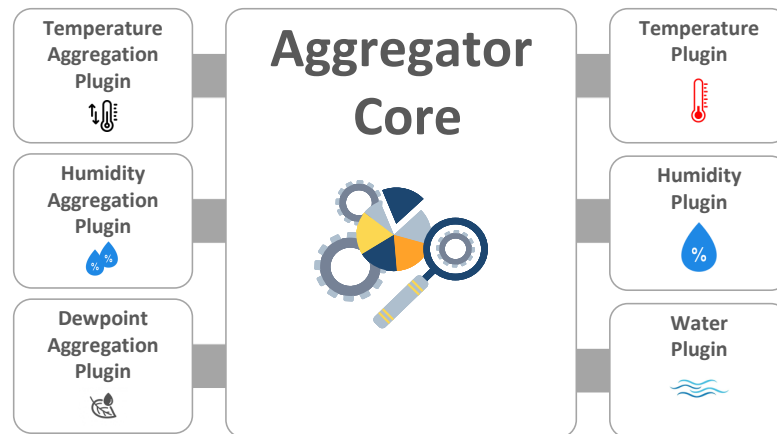


Abbildung 3.1: Plugin-Architektur des Aggregator Service

3.3.1 Prototyp Plugin-Architektur

Eine zentrale Anforderung ist die Erstellung der Komponente als Microservice und das Bereitstellen mit Kubernetes für das DSL-Cluster. *Spring Boot* stellt den Standard für die Implementierung von Microservices mit Java dar. Ein Microservice kann als einfache Spring-Anwendung entwickelt werden. Spring stellt jedoch keine Lösung zur Abbildung einer Plugin-Architektur bereit. Also wurden zuerst mehrere Frameworks verglichen, mit denen die Erstellung einer solchen Architektur realisiert werden kann.

Das populärste Framework zur Erstellung von Plugin-Architekturen ist das *Plugin Framework for Java (PF4J)* [73]. Es stellt Funktionalität bereit, um erweiterbare Anwendungen sowie deren Plugins zu implementieren. Dazu werden sogenannte *ExtensionPoints* definiert. Hierbei handelt es sich um Punkte, an denen Plugins Erweiterungen vornehmen können. Zudem stellt es einen *PluginManager* bereit, der Aufgaben wie das Laden, Starten und Stoppen von Plugins übernimmt. Das PF4J bietet auch ein Konzept zur Integration des Spring Frameworks. Dadurch können das Kernmodul und die Plugins als Spring-Anwendungen implementiert werden.

Eine Erweiterung des Frameworks ist *FlexiCore* [11]. FlexiCore baut auf dem Ansatz der Spring-Integration von PF4J auf und stellt zusätzliche Funktionalität bereit. Es vereinfacht die Integration von Plugins, indem der Implementierungs- und Konfigurationsaufwand reduziert wird. Ebenso stellt es mehr Möglichkeiten bereit, um eine Kommunikation zwischen einzelnen Plugins zu realisieren. Ein großer Nachteil des Frameworks stellt sich jedoch in dessen Schwergewichtigkeit dar. Flexicore erfordert die Installation eines Servers, welcher zahlreiche Features mit sich bringt. Die Zusatzfunktionen, die zur Zielumsetzung nicht benötigt werden, lassen sich jedoch nicht deaktivieren.

Dementsprechend wurde zur Umsetzung des Prototyps das PF4J verwendet. Hier war die Implementierung geringfügig komplexer. Man konnte jedoch an Abhängigkeiten sparen. Der resultierende Prototyp war alleinstehend lauffähig. Er verarbeitet einfache Sensormesswerte. Ein Konzept zur Aggregation wurde noch nicht umgesetzt. Zur Konzeption wurde ein Klassendiagramm erstellt, welches in Abbildung 3.2 zu sehen ist. Hiermit kann der Aufbau der Anwendung nachvollzogen werden.

3.3.2 Prototyp Microservices

Zur Umsetzung eines Prototyps basierend auf einer Microservices-Architektur musste ein multi-modulares Projekt erstellt werden [33]. Es wurde eine Code-Basis entwickelt, welche Schnittstellen für die Microservices beinhaltet. Hierbei handelt es sich nicht um

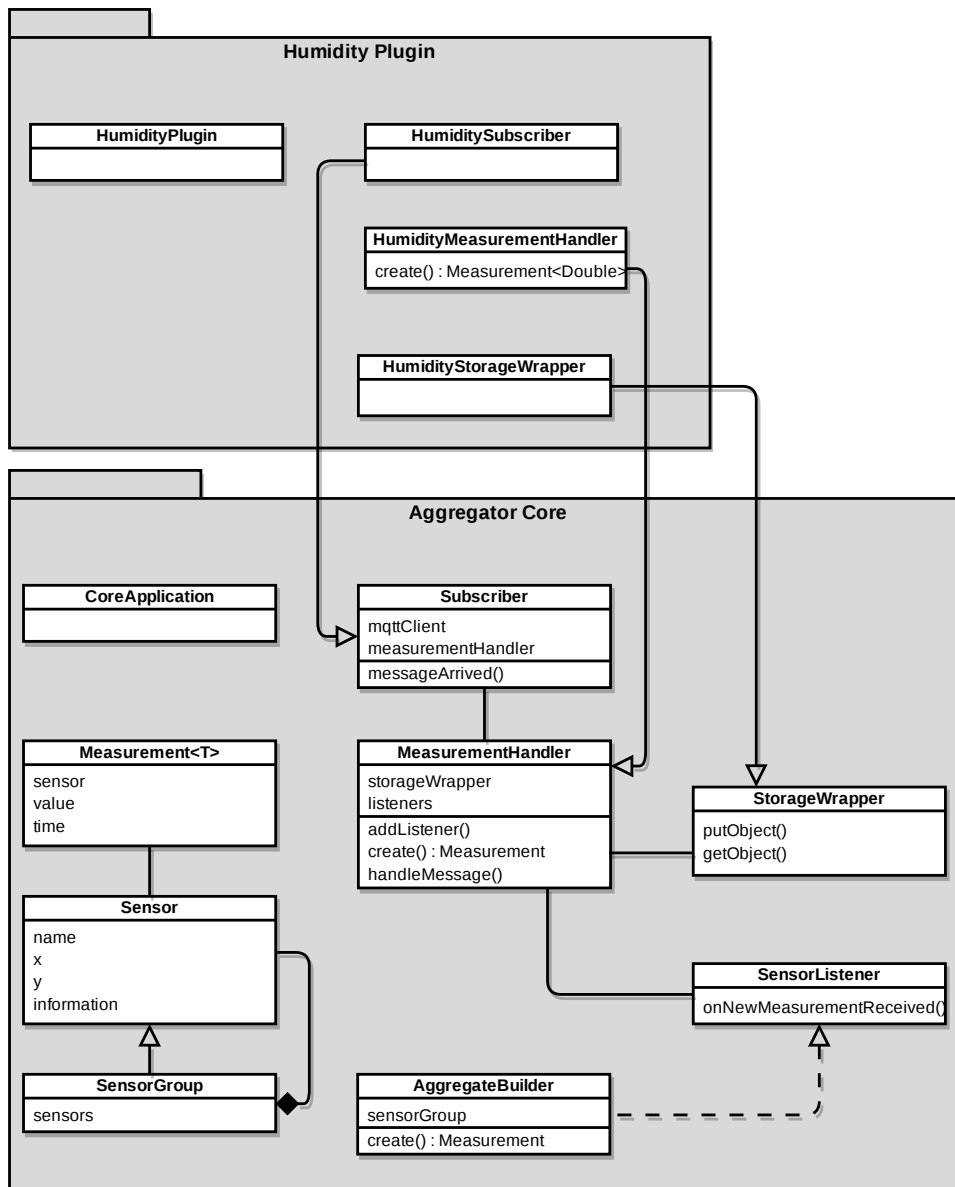


Abbildung 3.2: Klassendiagramm Prototyp Plugin-Architektur

ein ausführbares Programm, sondern eine Bibliothek. Diese wird als Abhängigkeit in den Microservices eingebunden. Die Code-Basis tätigt Vorgaben zum Aufbau der einzelnen Services und stellt Basisfunktionalität bereit. Dadurch muss in jedem Microservice nur die Logik implementiert werden, die diesen abgrenzt.

Der Prototyp besteht aus zwei Microservices, die Rohdaten verarbeiten, einigen Microservices zur Aggregation und einem Hilfsmodul, welches die Stammdatenverwaltung realisiert. Es werden Temperatur- und Luftfeuchtigkeitsdaten verarbeitet. Ein verarbeitender Microservice aggregiert Temperaturgruppen, ein anderer berechnet Tauwerte, ein weiterer berechnet Wärmeflusswerte über eine komplexere Formel. So konnte geprüft werden, inwiefern das Architekturmodell die Umsetzung der verschiedenen Anforderungen ermöglicht.

Zur Weiterverarbeitung von Rohdaten muss eine Kommunikation zwischen den Microservices erfolgen. Es wurde erstmals eine zentrale Komponente konzipiert, die eine Verteilung der Nachrichten basierend auf den Gruppen durchführt. Schnell fiel jedoch

auf, dass eine solche Komponente einen *Single point of failure* darstellt. Zudem muss diese Komponente alle aktiven Services kennen, ähnlich einem *Service Discovery System*. Ein Prinzip von Microservices-Architekturen ist jedoch die Unabhängigkeit der einzelnen Services voneinander. Es soll lediglich eine lose Kopplung bestehen. Dementsprechend wurde die Idee einer zentralen Komponente zur Verteilung verworfen. Der Message Broker kann ebenfalls zur Verteilung der verarbeiteten Messwerte verwendet werden. Ein rohdatenverarbeitender Microservice publiziert eine Referenz zu einem Messwert auf einem entsprechenden Exchange. Über Bindings gelangt der Messwert in die jeweiligen Queues der weiterverarbeitenden Microservices. Aus den Queues können diese den Wert entnehmen und ihre Aggregate bilden. Dadurch ergibt sich ein sehr geringer Verwaltungsaufwand. Ein Microservice agiert als unabhängige Komponente und bildet lediglich einen Verarbeitungsschritt ab. Des Weiteren ergibt sich der Vorteil, dass man die Aggregate erneut in den Prozess zur Weiterverarbeitung einfließen lassen kann.

3.3.3 Vergleich der Lösungsansätze

Um eine Entscheidung für einen Lösungsansatz zu treffen, wurde ein Vergleich beider Prototypen durchgeführt. Es wurden die Vorteile beider Ansätze erfasst und gegeneinander abgewogen. Ebenfalls fand eine Analyse der zugrunde liegenden Design-Patterns statt.

Ein Vorteil der Plugin-Architektur ist die Tatsache, dass man nur eine Anwendung implementieren, konfigurieren, bereitstellen und verwalten muss. Somit hat man während der Umsetzung und auch später während Pflege und Instandhaltung einen geringeren Zeitaufwand. Außerdem wird weniger Netzwerkkommunikation benötigt als bei einer Microservices-Architektur, wo die Services untereinander kommunizieren müssen. Die Einfachheit der Anbindung neuer Funktionalität stellt einen weiteren Vorteil dar. Plugins können bequem zur Laufzeit hinzugefügt werden. Hierbei spricht man von *Hot Deployment* [83].

Die Microservices-Architektur gewährleistet ebenso ein Hot Deployment. Zudem überzeugt sie durch Skalierbarkeit und Load Balancing. Ein einzelner Verarbeitungsschritt kann ein Engpass für den Gesamtablauf darstellen. In einer Microservices-Architektur werden Verarbeitungsschritte durch Microservices abgebildet. Durch die Replikation eines Services kann somit eine gezielte Skalierung durchgeführt werden. Diesen Vorteil bietet eine Plugin-Architektur nicht. Ein weiterer Vorteil ist im Punkt Fehleranfälligkeit zu sehen. Die Verwendung mehrerer Services mit eigenem Kontext führt zu einer Fehlerisolierung. Jede einzelne Anwendung ist widerstandsfähiger und bildet eine logische Einheit. Ebenfalls bietet eine Microservices-Architektur eine größere Flexibilität bezüglich Erweiterung und Bereitstellung. Einzelne Services können beliebig hinzugefügt und entfernt werden. Für diese Architektur sprach auch die einfachere Umsetzung des Microservices-Prototyps. Hier war der Implementierungsaufwand geringer. Außerdem konnte durch die Verwendung des Entwurfsmusters eine höhere Qualität des Quellcodes erreicht werden.

Es wurde entschieden, den Lösungsansatz basierend auf Microservices zu verwenden. Die Vorteile der Microservices-Architektur wurden als überwiegend angesehen. Maßgeblich trug die Modularität des Systems zur Entscheidung bei. Damit einhergehend ist die Architektur flexibler in Erweiterung und Skalierbarkeit. Diese Vorteile sind für ein IoT-System von hohem Stellenwert.

4 Verwandte Arbeiten

Im Folgenden wird auf Technologien eingegangen, die in verwandten Arbeiten zur Lösung vergleichbarer Problemstellungen eingesetzt werden. Dabei wurden Projekte und Forschungen ausgewählt mit möglichst ähnlichen Anforderungen. Es wird auf die Frameworks *Hadoop MapReduce* und *Spark Streaming* eingegangen. Diese ermöglichen eine Verarbeitung von Datenströmen. Vor- und Nachteile der Frameworks werden herausgestellt. Insbesondere wird die Abdeckung der Anwendungsfälle evaluiert. Zuletzt wird eine verwandte Arbeit vorgestellt, die zur Konzeption des *Autoscalings* als Referenz genutzt werden konnte.

4.1 Hadoop MapReduce

Hadoop MapReduce ist ein Framework zur parallelen Verarbeitung von großen Datenmengen [53]. Hierbei werden eingehende Datenströme in einzelne Datenpakete aufgeteilt. Das Framework arbeitet mit Schlüssel-Wert-Paaren auf unterster Ebene. Diese werden dann an die *Mapper-Tasks* weitergegeben. Ein Mapper transformiert ein Schlüssel-Wert-Paar. Als Ausgabe erfolgt ein anderes Paar, mehrere oder keine Paare. Die Ergebnisse werden vom Framework verarbeitet und in der *Shuffle-Operation* entsprechend der Schlüssel sortiert. Anschließend werden die Daten den *Reduce-Tasks* zur Verfügung gestellt. Ein Reducer fasst Schlüssel-Wert-Paare zusammen. Dazu werden diese basierend auf ihrem Schlüssel gruppiert. Im nächsten Schritt werden alle zu einem Schlüssel gehörenden Werte zu einem neuen Wert berechnet. Als Ergebnis entsteht eine kleinere Menge an Schlüssel-Wert-Paaren, wo jeder Schlüssel nur noch einmal vorkommt.

Eine verwandte Arbeit [9] analysiert die Verwendung von Hadoop MapReduce zur Umsetzung eines skalierbaren Systems zum Speichern und Analysieren von Sensordaten. Das System setzt eine *NoSQL-Datenbank* zur Haltung der Daten ein. Sensordaten werden direkt als *JSON-Dateien* gespeichert. Die Daten werden dann aus der Datenbank in Hadoop geladen und dort verarbeitet.

Die dargestellte Datenverarbeitung umfasst lediglich einfache Rohdaten und simple Aggregate. Inwiefern Hadoop MapReduce die komplexeren Aggregationsberechnungen umsetzen kann, musste zusätzlich betrachtet werden. Der Aufbau aller Daten als Schlüssel-Wert-Paare limitiert die Möglichkeiten, komplexe Aggregate zu bilden. Zur Umsetzung aller Anforderungen muss es außerdem möglich sein, Metadaten zu verrechnen. Diese können für Sensoren und Gruppen, in denen sie aggregiert werden, vorliegen. Zusätzliche Metadaten lassen sich in das Datenformat nicht direkt integrieren. Dadurch wird eine Umsetzung der Anforderung zur Gruppierung von Sensoren erschwert.

Ein weiterer Nachteil von MapReduce stellt sich in der Mehrfachverarbeitung von Daten dar. Sensoren sollen in mehreren Gruppen präsent sein können, um verschiedene Aggregate zu berechnen. Dementsprechend müssen die Messwerte dieser Sensoren mehrfach verarbeitet werden. Für eine zyklische Verarbeitung ist Hadoop ungeeignet. In diesem Fall sollte man eher auf Stream Processing oder andere Lösungswege zurückgreifen. Hadoop folgt dem "*Write-once-read-many*" Prinzip. Hierbei wird keine mehrfache Iteration bei der Verarbeitung unterstützt [4].

Außerdem ist anzumerken, dass bei dieser Arbeit keine Verarbeitung in Echtzeit erfolgt.

Stattdessen wird ein *Batch Processing* von Daten durchgeführt, die bereits längere Zeit in der Datenbank gespeichert sind. Ein typischer Anwendungsfall von MapReduce ist die Verarbeitung in festgesetzten Zeitintervallen [99]. Die gegebenen Anforderungen machen jedoch eine Verarbeitung in Echtzeit notwendig. Sobald Rohdaten vorliegen, sollen Aggregate gebildet werden. Diese sollen dann erneut zur Berechnung genutzt werden können. Bei einer zeitversetzten Berechnung bleibt die Weiterverarbeitung von aufbereiteten Daten auf der Strecke. Hier liegen Aggregate erst zu späteren Zeitpunkten vor und können somit nicht zeitgleich mit anderen Rohdaten verrechnet werden.

Hadoop ist nicht einfach einzusetzen und zu implementieren. Für jede Operation muss Programmcode erstellt werden. Dadurch wird der Implementierungsaufwand erhöht und Fehler sind schwerer zu lokalisieren. Ebenso ist die Konfiguration und Verwaltung aufwendiger als die eines maßgeschneiderten Systems [48].

4.2 Stream Processing

Viele verwandte Arbeiten greifen bei der Implementierung von Anwendungen zur Sensorverarbeitung auf *Stream Processing* zurück. Hier werden Daten als kontinuierlicher Fluss von Elementen betrachtet. Diese Datenflüsse werden in Echtzeit abgefragt und verarbeitet [10]. Ein populäres Stream Processing Framework, welches auch häufig im Bereich Internet of Things verwendet wird, ist *Spark Streaming*.

Spark Streaming ist eine Erweiterung der *Apache Spark API* [67]. Daten können hier von verschiedenen Quellen erhoben und mit komplexen Algorithmen verarbeitet werden. Die Daten können nach Verarbeitung gespeichert und in Dashboards visualisiert werden [89]. Bei Spark Streaming wird ein Datenfluss als *Discretized Stream (DStream)* bezeichnet. Diese Streams können sowohl die eingehenden Daten als auch die verarbeiteten Daten nach einer erfolgten Transformation abbilden. Ein DStream besteht aus einer Folge von *Resilient Distributed Datasets (RDDs)*. RDDs kapseln Daten in einer Zeitspanne. Die Operationen auf den DStreams werden von der Spark Engine auf den RDDs ausgeführt. Damit bleibt die Komplexität der Datenverarbeitungsoperationen verborgen [89].

Ein Vorteil, den eine eigene Implementierung bietet, ist eine dynamischere Verarbeitung der Daten. Stream Processing bietet zwar viele Möglichkeiten zur Verarbeitung eines Datenflusses an, jedoch stoßen die meisten Frameworks bei komplexen Anforderungen an ihre Grenzen. Insbesondere lässt sich der Anwendungsfall einer komplexen Aggregatbildung über Gruppen verschiedener und gleicher Sensortypen schwer konzipieren. Hier ist die Unterscheidung von gleichen Sensortypen anhand von Metadaten notwendig, welche extern in Form von Stammdaten gehalten werden. Ein Beispiel wäre die Notwendigkeit einen Außen- von einem Innentemperatursensor zu unterscheiden, um die Berechnung eines Kondensationswerts durchzuführen. Die Verwendung von Operationen, welche einen Zugriff auf externe Speichermedien ausführen, sollten in einem Stream Processing Szenario vermieden werden.

Zudem gelangen bei Stream Processing die Ergebnisse von Operationen stets zur Weiterverarbeitung in neue Streams. Die gegebenen Anforderungen lassen sich jedoch durch ein flexibleres System besser lösen. Ein Messwert nur weiterverarbeitet werden, wenn es eine Gruppe gibt, die zur Aggregatbildung auf diesen angewiesen ist. Bei einer eigenen Implementierung kann je nach Datensatz eine Entscheidung getroffen werden. Es ist möglich, nur relevante Daten für eine Weiterverarbeitung wieder in den Kreislauf einfließen zu lassen. Während Daten, die nicht mehr für eine weitere Aggregatbildung benötigt werden, verworfen werden können. Dadurch kann eine performantere und effizientere Lösung erreicht werden.

Der Anwendungsfall, Aggregate nach Gruppen zu bilden, macht für die Berechnung

ein Warten auf alle Werte der Mitglieder notwendig. Die Anforderungen sehen eine möglichst große Flexibilität aufseiten der Sensoranbindung vor. Aus diesem Grund und der Tatsache, dass Sensoren in beschränkten Umgebungen zum Einsatz kommen, ist nicht zu gewährleisten, dass die Messwerte in gleicher Frequenz ankommen. Ein Warten auf Nachrichteneingang lässt sich mit Stream Processing nur über Umwege abbilden und widerspricht dem Grundkonzept.

Ein weiterer Vorteil einer eigenen Lösung ist die Vermeidung von Abhängigkeiten von Fremdsoftware. Eine maßgeschneiderte Entwicklung stellt einerseits eine exakte Lösung der Problemstellung dar. Außerdem lässt sie sich auch in Zukunft besser steuern. Schnittstellen können selbst entwickelt werden. Updates können eigenständig entworfen werden. Man ist somit viel flexibler in der Weiterentwicklung.

4.3 Autoscaling

Um ein Autoscaling von Spring Boot Anwendungen über Kubernetes zu realisieren, müssen zusätzliche Module installiert und Konfigurationen getätigt werden. Es gibt einige Projekte und Dokumentationen, die die Umsetzung einer automatischen Skalierung beschreiben. Die ausgewählte verwandte Arbeit [120], die im Folgenden kurz beschrieben wird, war auf den eigenen Anwendungsfall am besten übertragbar. Bei der Arbeit handelt es sich um eine Applikation, die Tickets in Form von Nachrichten empfängt. Diese werden dann in eine Queue eingefügt. Worker-Anwendungen entnehmen die Tickets anschließend aus dieser und verarbeiten sie. Auf Basis der Anzahl eingehender Nachrichten erfolgt eine automatische Skalierung. Die Skalierung wird über eine eigene Metrik durch den *Horizontal Pod Autoscaler* durchgeführt. Als Metrik zur Skalierung wird hier die Größe der Queue verwendet. Diese wird über REST auf dem Endpunkt `/metrics` angeboten. Sie befindet sich in einem Format, welches dem Prometheus Standard entspricht. *Prometheus* fragt die Metrik dann in regelmäßigen Intervallen ab. Der *Metrics Server* liest sie aus. Über die *Custom Metrics API* steht sie dann dem HPA bereit. Basierend auf dem aktuellen Wert kann er entscheiden, ob mehr oder weniger Applikationen zur Bewältigung der Last benötigt werden.

Es wird in der Dokumentation auch darauf eingegangen, wie eigene Metriken erstellt werden können und welche Möglichkeiten neben der Verwendung des Metrics Servers noch bestehen. Hier wird der *Prometheus Adapter* als Alternative genannt.

5 Konzeption

In der Analysephase fiel die Entscheidung eine Microservices-Architektur zu verwenden. Im nächsten Schritt musste der Aufbau des Systems konzipiert werden. Ebenfalls mussten Konzepte zur Umsetzung der Anwendungsfälle erstellt werden. Zuerst wurde hierzu das *Top-Level Design* durchgeführt. Hierbei wurde das Gesamtsystem in einzelne Komponenten zerlegt. Dann wurden die Aufgaben dieser grob skizziert. Es wurde die Anbindung der Sensoren sowie die Zusammensetzung des Backends konzipiert. Anschließend wurden die Systembausteine im Detail entworfen. Dazu wurde das *Low-Level Design* erstellt. Es fand die Modellierung von Entitäten statt. Der Aufbau der einzelnen Microservices wurde festgelegt. Ebenso wurden die Anwendungsfälle zur automatischen Skalierung sowie zur historischen Aggregation ausgearbeitet. Zuletzt wurde das Konzept zur Visualisierung in Dashboards aufgestellt.

5.1 Top-Level Design

Die *Top-Level-Architektur* kann in Abbildung 5.1 nachvollzogen werden. Hier sind die verschiedenen Systemkomponenten dargestellt. Aufseiten der Microservices ist zwecks Anschaulichkeit nur ein ausgewählter Teil abgebildet. Zum Verständnis der Gesamtarchitektur werden im Folgenden die vier Hauptkomponenten erläutert.

Model City: Die *Model City* stellt ein Abbild einer realen Stadt dar. Sinn und Zweck ist die Simulation eines Smart City Szenarios. Die Model City ist mit Sensoren und Aktoren ausgestattet. Alle zur Verfügung stehenden Sensoren sind mit einer Anwendung anzubinden. Diese Anwendung muss eine Verbindung zum Backend aufbauen.

Environment Simulation: Die *Environment Simulation* dient der zusätzlichen Simulation von Sensoren, die nicht zur Verfügung stehen, aber für die Abbildung aller Anwendungsfälle benötigt werden. In diesem Sinn kann man von virtuellen Sensoren sprechen, also Anwendungen, die vergleichbar mit physischen Sensoren Messwerte generieren. Die erhobenen Daten fließen analog zu den Sensoren der Model City ins System ein. Hierzu ist eine Anwendung zu implementieren, die Sensoren simuliert beziehungsweise virtuelle Sensoren realisiert.

Backend Kubernetes-Cluster: Das *Backend* umfasst die Komponenten, welche die Anwendungslogik implementieren. Hier wird ein Message Broker zur Verteilung von Nachrichten eingesetzt. Verschiedene Microservices werden zur Verarbeitung und Aggregation der Sensordaten eingesetzt. Ebenso müssen Anwendungen zur Datenthaltung integriert werden. Alle weiteren Anwendungen, die zur Funktionalität dieser Komponenten benötigt werden, werden hier ebenfalls bereitgestellt.

Frontend: Das *Frontend* umfasst Dashboards, welche zur Visualisierung der erhobenen Messwerte verwendet werden. Hier ist der Einsatz von Endanwendungen auf mobilen Geräten, Desktop Computern oder auch im Web vorstellbar.

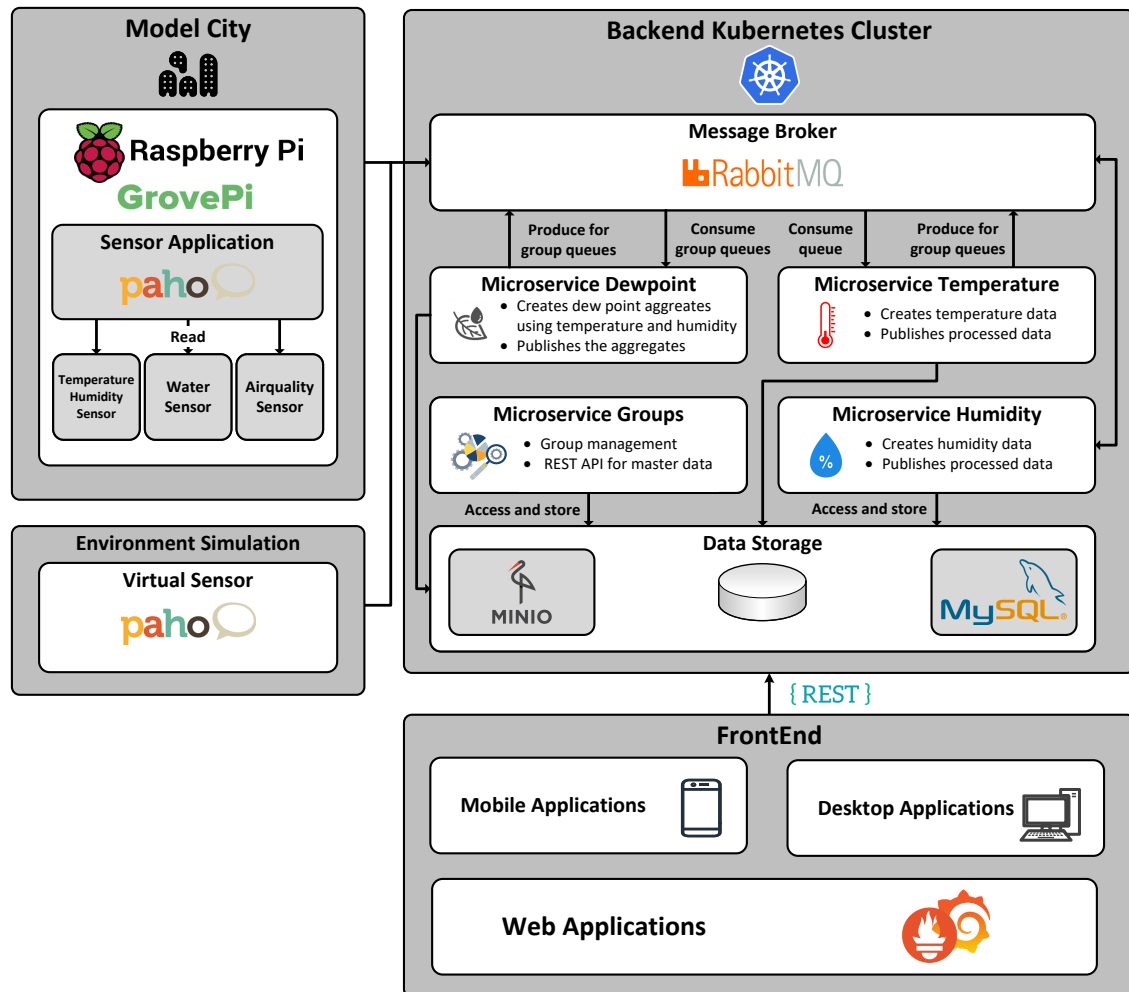


Abbildung 5.1: Architektur Smart City Testfeld

5.1.1 Sensoranbindung

In der Model City sollen *Grove Sensoren* eingesetzt werden. Hier musste die Anbindung der Sensoren analysiert werden. Es stehen die Sensortypen *Temperature&Humidity*, *Airquality* sowie *Water* zur Verfügung. Die Sensoren werden über das *Add-on Board GrovePi+* mit einem *Raspberry Pi* verbunden. Der *GrovePi+* stellt eine Brücke dar, welche die Kommunikation zwischen Sensoren und dem *Raspberry Pi* ermöglicht. Zur Umsetzung lag ein *Raspberry Pi 4 Computer* mit 4GB RAM vor. Eine Ansteuerung der Sensoren kann mittels digitaler und analoger Ports erfolgen. Die Sensoranwendung soll die Daten vom Sensor in regelmäßigen Zeitabständen auslesen und diese dann über MQTT an den Message Broker versenden. Dazu muss eine Verbindung aufgebaut werden und eine Veröffentlichung mittels MQTT-Publish erfolgen.

5.1.2 Backend im Kubernetes-Cluster

Der Aufbau des Backends musste ebenfalls analysiert werden. Hier ist der Großteil der Anwendungslogik zu implementieren. Zur Analyse wird das Backend in drei Hauptkomponenten aufgeteilt. Hierbei handelt es sich um den Message Broker, die Microservices und die Software zur Datenhaltung. Jede Backend-Anwendung ist als Kubernetes Deployment bereitzustellen. Außerdem soll benötigte Fremdsoftware, wie beispielsweise

MySQL und MinIO zur Datenhaltung, nach Möglichkeit über Helm Charts installiert werden. Dadurch kann der Eigenaufwand zur Installation und Konfiguration minimal gehalten werden. Als Testumgebung soll ein lokales Minikube-Cluster verwendet werden. Das finale Deployment findet im DSL-Cluster statt. Im Folgenden werden die Hauptkomponenten im Einzelnen erläutert.

5.1.2.1 Message Broker

Es wird ein *RabbitMQ Message Broker* zum Verteilen der Sensordaten verwendet. RabbitMQ verwendet zur Nachrichtenübertragung Exchanges und Queues. Außerdem unterstützt es Publish-Subscribe Messaging. Damit wird das Senden von Nachrichten auf Topics ermöglicht. Anfangs wurde geplant, *MQTT* zur Kommunikation mit dem Message Broker zu verwenden. Während der Analysephase fiel auf, dass die alleinige Nutzung von MQTT zur Umsetzung der Anforderungen jedoch nicht ausreicht. Bei MQTT werden Nachrichten an alle Subscriber versendet. Dies ist allerdings problematisch, wenn die zu entwickelnden Anwendungen mit Kubernetes skaliert werden sollen. Eine Replikation einer Anwendung, die als Subscriber agiert, führt zu einer mehrfachen Datenverarbeitung. Die ankommenden Nachrichten werden an jedes Replikat verteilt. Messwerte werden mehrfach erhoben. Dieses Verhalten ist nicht erwünscht und verhindert ein *Load Balancing*. Hier sollen Nachrichten zwischen Replikaten aufgeteilt werden, um so einen erhöhten Durchsatz zu ermöglichen.

Zur Lösung dieses Problems sollen zur Bereitstellung der Nachrichten die *AMQP-Queues* verwendet werden. Entnimmt ein Konsument eine Nachricht aus der Queue, so wird diese entfernt. Dementsprechend können die Nachrichten nur von einem Konsumenten verarbeitet werden. Die Skalierung der Anwendungen kann problemlos erfolgen. RabbitMQ ermöglicht über Bindings eine einfache Umleitung der Nachrichten von einem Exchange zu einer Queue. Der Topic Exchange *amq.topic* empfängt alle über MQTT eingehenden Nachrichten. Mittels eines Routing Keys kann entschieden werden, an welche Queue eine Weiterleitung zu erfolgen hat. Bei der Erstellung eines Bindings werden die Queue und die jeweiligen Routing Keys definiert. Damit ist es möglich, Nachrichten verschiedener Topics in eine gemeinsame Queue zusammenlaufen zu lassen.

Eine Sensoranwendung bestimmt einen Messwert und publiziert diesen über MQTT auf dem Topic Exchange. Die Bindings sorgen für die Weiterleitung entsprechend dem Sensortyp in die jeweilige Queue. Aus dieser kann ein rohdatenverarbeitender Microservice die Daten entnehmen und die Verarbeitung durchführen. Wie der beschriebene Ablauf im konkreten Anwendungsfall aussieht, ist in Abbildung 5.2 dargestellt. Durch den Einsatz von MQTT zur Veröffentlichung der Daten auf Sensorseite kann man die Vorteile des Protokolls in Bezug auf Performanz und Einfachheit der Anbindung nutzen. Die Verwendung von AMQP ermöglicht die dynamische Verteilung der Nachrichten über die Exchange und Queue Logik. Durch die parallele Anwendung von MQTT und AMQP kann von den Vorteilen beider Protokolle profitiert werden.

Die Abbildung der Weiterverarbeitung von Messwerten unterscheidet sich von der Erhebung der Rohdaten. Hier können die Referenzen auf die verarbeiteten Messwerte direkt über AMQP publiziert werden. Sie werden zum Topic Exchange *group_exchange* versandt. Um die Informationen über den Produzenten des Messwerts sowie die aggregierende Gruppe zu erhalten, wird ein entsprechender Routing Key gesetzt. Dazu wird die Konvention $\langle group_type \rangle . \langle group_id \rangle . \langle producer_id \rangle$ festgelegt. Der Vorteil den dieser Aufbau des Routing Keys mit sich bringt, wird in Kapitel 5.1.2.2 erläutert. Über Bindings gelangen die Referenzen in die Queues der jeweiligen Gruppentypen, wo sie von einem Microservice zwecks Aggregatbildung konsumiert werden. Auch dieser Ablauf kann in Abbildung 5.2 nachvollzogen werden.

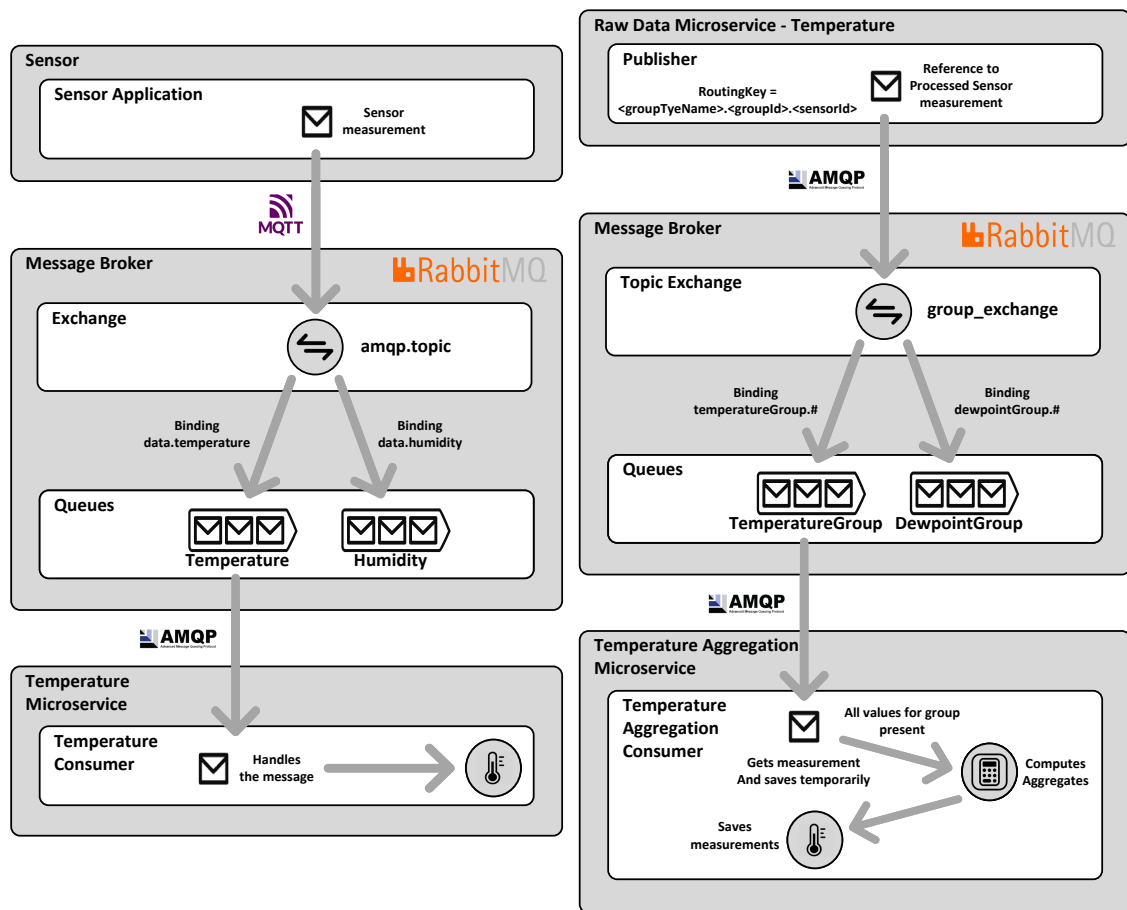


Abbildung 5.2: Message Broker Exchanges, Queues und Bindings

Der Verbindungsaufbau zum Message Broker erfolgt über *Transport Layer Security (TLS)*. Dadurch wird die kryptographische Sicherheit gewährleistet. Gemäß dem *Confidentiality, Integrity and Availability (CIA) Prinzip* sind Bedrohungen festgelegt als Verlust von Vertraulichkeit, Integrität und Verfügbarkeit. TLS reagiert auf diese, indem es ein Handshake-Protokoll zur Authentifizierung der Kommunikationspartner und ein Record-Protokoll zur Sicherung der Datenübertragung implementiert [81]. Zusätzlich zur Authentifizierung über Nutzernamen und Passwort wurden auf dem Server signierte Zertifikate installiert. Hierbei stammt die Signatur von einer Zertifizierungsstelle, auch *Certificate Authority (CA)* genannt. Die CA gewährleistet die Integrität der Zertifikate. Zudem verwaltet sie Bereitstellung sowie Zuordnung. Möchte ein Client eine Verbindung aufbauen, so muss er eine Anfrage mit seinem eigenen Zertifikat stellen. Neben dem Client Zertifikat werden weitere Dateien versandt, welche zu einer Verifizierung durch den Server benötigt werden. Der Client muss ebenfalls einen Schlüssel mitschicken. Dieser wird vom Server zur Entschlüsselung der Kommunikation verwendet. Dadurch liegt ein Schutz gegenüber Man-in-the-middle Angriffen vor. Anschließend prüft der Server die übermittelten Daten. Bei erfolgreicher Prüfung akzeptiert er die Verbindung, andernfalls lehnt er sie ab.

5.1.2.2 Microservices

Zur Abbildung der Anforderungen müssen zwei verschiedene Arten von Microservices unterschieden werden. Die Einteilung kann nach dem Typ der zu verarbeitenden Daten erfolgen. Man differenziert zwischen rohdatenverarbeitenden und aggregierenden Micro-

services. Die erzeugten Messwerte beider Arten müssen gleichbehandelt werden, um sie erneut aggregieren zu können.

Der konzipierte Ablauf einer Aggregation kann in Abbildung 5.3 nachvollzogen werden. Die Weiterverarbeitung von Messwerten wird über den Message Broker ermöglicht. Ein Microservice erzeugt beim Starten Queues und Bindings basierend auf seiner Konfiguration. Empfängt ein Microservice einen Messwert vom Broker, so verarbeitet er diesen. Schließt er die Verarbeitung ab, so erzeugt er einen Datensatz. Dieser erhobene Messwert wird im Object Storage persistiert und im Cache als aktueller Wert hinterlegt. Das Speichern des letzten Messwerts im Cache dient der Darstellung in Dashboards. Der Hintergrund kann in Kapitel 5.2.5 nachvollzogen werden. Nach dem Caching wird geprüft, ob der Messwert noch zu einer Aggregatbildung einer übergeordneten Gruppe benötigt wird. Im nächsten Schritt erzeugt der Microservice eine Referenz auf den Messwert. Bei dieser Referenz handelt es sich um eine *Presigned URL*, welche den Zugriff auf den Messwert ermöglicht. Der Microservice publiziert anschließend die URL auf dem Topic Exchange *group_exchange*. Bei dem Veröffentlichen setzt er den Routing Key entsprechend der festgelegten Konvention, heißt $\langle group_type \rangle . \langle group_id \rangle . \langle producer_id \rangle$. Dadurch kann die Weiterleitung des Messwerts über den Gruppentyp in die jeweilige Queue erfolgen. Aus dieser können die anderen Microservices wiederum die Referenz entnehmen und über diese den Messwert aus dem Object Storage auslesen. Mit dem Routing Key kann zudem der Produzent ermittelt werden. Ebenso ist die Gruppe festgelegt, für die das Aggregat zu erzeugen ist. Der Zugriff auf Metadaten auf Sensor- und Gruppenebene wird somit ermöglicht.

Für aggregierende Microservices kann ein Messwert nicht direkt bei der Verarbeitung bestimmt werden. Zur Bildung eines Aggregats müssen alle Messwerte der jeweiligen Gruppe vorliegen. Um alle aktuell bereits ermittelten Messungen zu speichern, sah das Konzept vor, temporäre Dateien im Object Storage anzulegen. Ein weiterverarbeitender Microservice sollte den gelesenen Messwert in eine solche Datei schreiben. Diese sollte genutzt werden, um alle Daten der Gruppe zu sammeln und bei Ankunft des letzten Datensatzes das Aggregat zu bilden. Anschließend sollte die temporäre Datei gelöscht werden, damit der Vorgang von vorne beginnen kann. Dieses Konzept wurde in der Implementierungsphase auch so umgesetzt. Unvorhergesehene Probleme traten jedoch bei den ersten Testläufen auf, welche eine nachträgliche Änderung erforderten. Statt der Zwischenspeicherung vorliegender Messwerte im Object Storage wurde hierfür auch der Cache verwendet. Wie das vollständige Konzept zur Aggregatbildung umgesetzt wurde, ist in Kapitel 6.2.4.2 dargestellt. Nach der Ankunft des letzten Messwerts kann das Aggregat gebildet werden. Darauf folgend wird dieses gespeichert und die URL zur Weiterverarbeitung wird generiert. Mit der Publikation der URL über AMQP endet auch für die aggregierenden Microservices der Prozess. Da das Aggregat in einer weiteren Gruppe als Mitglied eingetragen sein kann, kann nun derselbe Vorgang von Neuem beginnen. Aggregate können beliebig oft erneut aggregiert werden und mit anderen Aggregaten oder Rohdaten verrechnet werden.

5.1.2.3 Data Storage

Messwerte und Aggregate sollen im *MinIO Object Storage* persistiert werden. Die Verwendung eines Object Storage bietet gegenüber einer herkömmlichen relationalen Datenbank zahlreiche Vorteile. Ein Object Storage ist besser zur Verarbeitung unstrukturierter Daten geeignet. In einem IoT-System können Sensoren beliebige Arten von Messwerten generieren. Dementsprechend müssen auch unstrukturierte Daten, zum Beispiel Bilder oder Videos, verarbeitet werden können. Ein weiterer Vorteil ist die Skalierbarkeit. Durch eine einfache Struktur der Datenhaltung ist das System unbeschränkt skalierbar [7]. Zudem ist

5 Konzeption

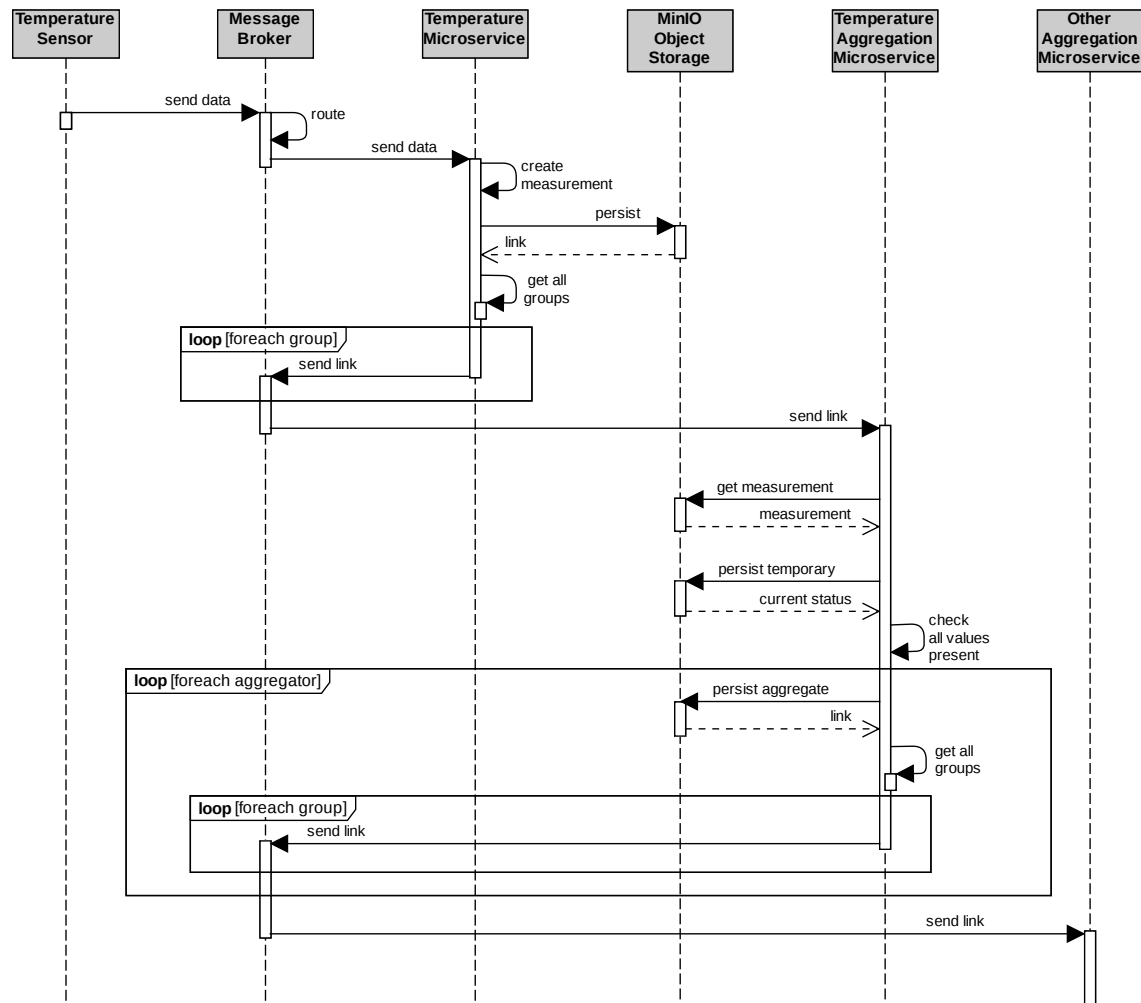


Abbildung 5.3: Kommunikation Microservices

die Verwendung eines Object Storages unkomplizierter und performanter. Durch simple Zugriffe und flache Hierarchien sind Lese- und Schreibvorgänge schnell durchführbar.

Eine Alternative zur Verwendung eines *Amazon S3 Object Storages* wäre die Nutzung von *NoSQL-Lösung*, wie beispielsweise *MongoDB* oder *DynamoDB*. Diese Lösungen ermöglichen die Aktualisierung und Änderung von Objekten beziehungsweise Dateien. Dieser Vorteil kann jedoch für das Projekt ignoriert werden. Messwerte stellen Momentaufnahmen dar. Eine spätere Änderung ist weder sinnvoll noch erlaubt. Ein anderer Vorteil kommt zum Tragen, wenn Operationen auf Gruppen von Objekten durchgeführt werden müssen. Bei den gegebenen Anforderungen ist die historische Aggregation ein solcher Anwendungsfall. Die Anforderung kann jedoch auch bei der Lösung mittels Object Storage durch Definition einer Ordnerstruktur entsprechend der Messzeitpunkte umgesetzt werden. Der Object Storage bietet bei den gegebenen Anforderungen weitere Vorteile, mit denen NoSQL-Lösungen nicht dienen können. Sensoren können beliebige Messwerte erheben, so sind auch Bild- oder Videodaten variabler Größe denkbar. Große Dateien sind hier ohne Umwege speicherbar. Im IoT-Umfeld wird mit umfangreichen Datenmengen gearbeitet. Alte Daten bleiben stets relevant und müssen archiviert werden. Die Durchführung einer Archivierung gestaltet sich für Object Storages sehr einfach. Eine Skalierung ist auch simpler und effizienter durchführbar. Aus diesen Gründen fiel die Wahl zur Haltung der Messwerte auf eine Object Storage-Lösung. Konkret wurde MinIO als Open Source Object Storage Server ausgewählt. Für MinIO spricht die Performanz, Skalierbarkeit und

die weite Verbreitung. Außerdem steht hier eine ausführliche Dokumentation und eine umfangreiche SDK zur Verfügung.

Die Messungen sollen als Objekte mit Datum und Wert im JSON-Format abgelegt werden. Mit MinIO kann man Objekte ebenfalls in Verzeichnissen strukturieren. Dadurch erhält man die Möglichkeit, Messwerte nach Sensoren zu gruppieren. Auf Ebene eines Sensors kann man Messwerte wiederum zeitlich ordnen. Eine hierarchische Organisation entsprechend dem Zeitpunkt der Erhebung ermöglicht eine schnelle Navigation bei der Suche nach Datensätzen. Zudem steigt die Übersichtlichkeit und die Umsetzung einer historischen Aggregation wird erleichtert.

Stammdaten, wie Sensoren, Gruppen und deren Metadaten, sollen in einer relationalen Datenbank abgelegt werden. Für die Verwaltung der Stammdaten in einer relationalen Datenbank spricht die Genauigkeit und Integrität. Durch die Nutzung von *Primary* und *Foreign Keys* kann gewährleistet werden, dass keine Duplikate entstehen. Gleichzeitig können Tabellen verbunden werden. So lassen sich den Sensoren verschiedene Datentypen zuordnen. Des Weiteren können so einer Gruppe Mitglieder und Aggregatoren zugeteilt werden. Der Aufbau eines Entitätsmodells mit Relationen kann realisiert werden. Ein Object Storage ist für die Verwaltung strukturierter Daten weniger geeignet. Durch eine relationale Datenbank lassen sich Strukturen präzise abbilden.

5.2 Low-Level-Design

Mit dem *High-Level-Design* wird der Gesamtaufbau in die jeweiligen Bausteine zerlegt und deren Funktionalität festgelegt. Anschließend müssen diese Bausteine im Detail konzipiert werden. Das *Low-Level-Design* befasst sich mit der Beschreibung des inneren Aufbaus der Komponenten. Im Folgenden wird das Konzept der einzelnen Systemteile erläutert und auf wichtige Designentscheidungen eingegangen.

5.2.1 Entitätsmodellierung

Ein erster Schritt zur Modellierung stellt die semantische Darstellung der Daten dar. Es werden Entitäten gebildet und deren Relationen entworfen. Eine grafische Darstellung des Ergebnisses ist in Abbildung 5.4 zu sehen. Im Folgenden werden die dargestellten Datenmodelle beschrieben.

Zur Umsetzung der Aggregation müssen sowohl Sensoren als auch Aggregatoren in Gruppen eingeteilt werden können. Dazu wurde eine *Producer-Entität* konzipiert, die eine Generalisierung darstellt. Sie umfasst alle Attribute, die Sensoren und Aggregatoren gemeinsam haben. Eine wichtige Eigenschaft eines Producers ist sein *Datentyp*. Der Datentyp legt fest, welche Art von Messwerten erzeugt wird. Dadurch wird eine Einteilung realisierbar. Ein möglicher Datentyp wäre zum Beispiel Temperatur, Luftfeuchtigkeit oder Tauwert. Zur Klassifikation einer Gruppe kann der *Gruppentyp* verwendet werden. Dieser definiert die Art der Gruppe und soll zukünftig genutzt werden, um eine korrekte Anlage von Stammdaten zu gewährleisten. Man kann einer Gruppe Producer als Mitglieder zuordnen. Gleichzeitig kann man einer Gruppe aber auch Aggregatoren zuweisen, welche die Ausgabe der Gruppe bestimmen. Ein Aggregator hat einen hinterlegten *Combinator*. Hierbei handelt es sich um eine namentliche Funktion, welche eine Operation zur Aggregatbildung definiert. Um die komplizierteren Anforderungen bezüglich der Aggregation zu erfüllen, kann man Producer mit *Tags* versehen. Tags stellen Metadaten dar, die zur Einteilung und Kennzeichnung von Producern dienen sollen. Außerdem kann man *FormulaItems* definieren, wobei es sich um Metadaten auf Gruppenebene handelt. Den *FormulaItems* kann man durch Anlage von *FormulaItemValues* einen Wert zuordnen. Diese

können an einer Gruppe hinterlegt werden. Dadurch wird das Speichern von Schlüssel-Wert-Paaren ermöglicht, die für komplexe Berechnungen eingesetzt werden können. Ein mögliches Beispiel wäre die Berechnung eines Wärmedurchgangskoeffizienten. Hierfür benötigt man eine Außentemperatur, einen Tauwert aus dem Rauminneren sowie einen U-Wert. Der U-Wert stellt eine Konstante dar, welche je nach Raum unterschiedlich sein kann. Er gibt die Isolierung beziehungsweise Dämmung an [88]. Bei dem gegebenen Modell kann man dafür ein `FormulaItem` definieren und verschiedene Werte je nach Gruppe festlegen. Diese werden dann bei der Berechnung des Wärmedurchgangskoeffizienten berücksichtigt.

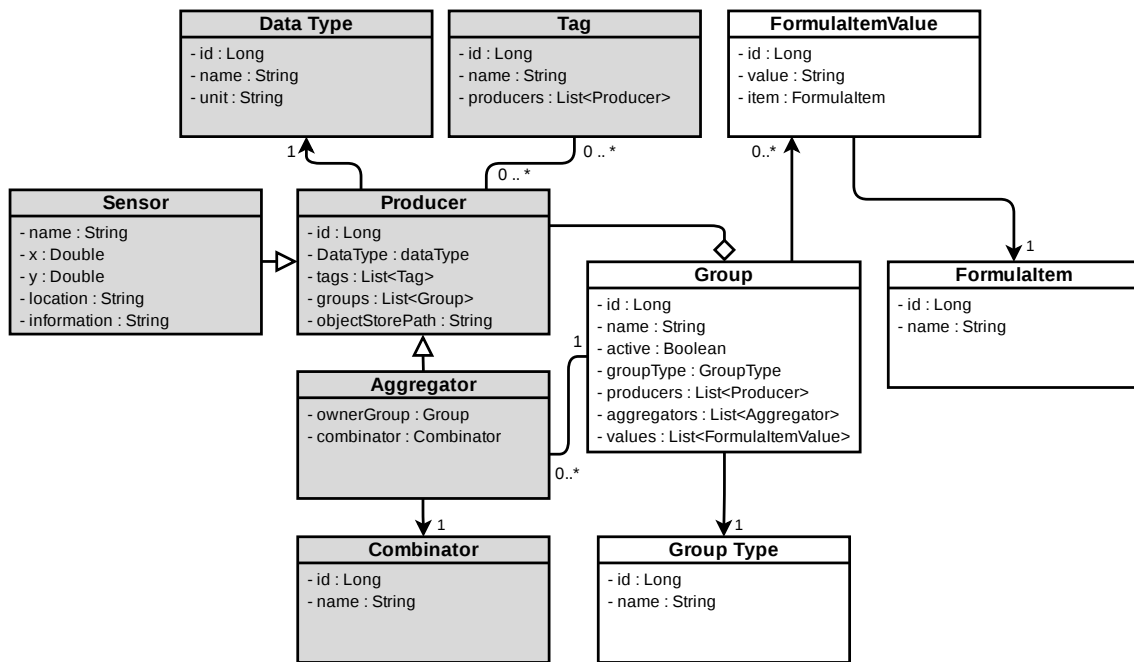


Abbildung 5.4: Entitäten zur Abbildung der Aggregation

Die Entitäten werden als *Plain Old Java Object (POJO)s* abgebildet, das heißt als gewöhnliche Objekte mit Attributen und Methoden. Hibernate wird als *Object Relational Mapping (ORM)-Framework* eingesetzt, um die Entitäten in einer relationalen Datenbank zu speichern und aus dieser zu lesen. Ein Vorteil der Verwendung von Hibernate zur Abbildung der Entitäten ist die Unabhängigkeit vom verwendeten Datenbanksystem. Einen späteren Austausch des Datenbanksystems kann man so ohne großen Aufwand durchführen. Zudem ermöglicht Hibernate die Minimierung des Aufwands bei Änderungen am Datenbankmodell. Tabellen werden automatisch erstellt und aktualisiert auf Basis der vorhandenen Entitäten. Hibernate stellt ebenso die meiste Logik zum Umgang mit den Modellen bereit und vereinfacht die Interaktion mit dem Datenbanksystem. Außerdem kann das Framework durch seine gute Skalierbarkeit und hohe Geschwindigkeit trotz großer Datenmengen überzeugen. Die Nutzung von Features wie *Lazy-Loading* sollen Abfragezeiten minimieren. Durch *Lazy-Loading* wird gewährleistet, dass immer nur die benötigten Daten abgefragt werden.

Es ist geplant, MySQL als RDBMS zu verwenden. MySQL überzeugt durch seine Performanz und Einfachheit. Die Konfiguration und Verwaltung des Systems gestaltet sich als simpel. Zudem lässt es sich sehr gut skalieren.

Die Verwaltung der Stammdaten über eine grafische Anwendung ist nicht im Rahmen des Projekts zu realisieren. Eine zukünftige Thesis soll sich mit der Anlage von Sensoren sowie der Erzeugung von Metadaten befassen. Dementsprechend muss für diese Arbeit

lediglich eine grundlegende Möglichkeit bestehen, die Stammdaten anzulegen, bis beide Projekte miteinander verknüpft werden. Aus diesem Grund ist die Bereitstellung einer simplen REST-Schnittstelle genügend. Um diese möglichst gut bedienbar zu gestalten, soll *OpenAPI Specification (OAS) 3.0* verwendet werden. Dieser Standard ermöglicht den Aufruf der Endpunkte über eine Webschnittstelle. Ebenfalls werden die Endpunkte sowie deren Funktionalität und Aufbau dokumentiert.

5.2.2 Aufbau Microservices

Jeder Microservice hat die Aufgabe Daten zu verarbeiten. Dieser Prozess kann in einzelne Schritte aufgeteilt werden, um die Komplexität zu reduzieren. Der Verarbeitungsprozess wurde auf die folgenden logischen Schritte reduziert:

1. **Datenempfang** - Daten werden vom Message Broker empfangen.
2. **Datenverarbeitung** - Die empfangenen Daten werden verarbeitet.
3. **Messwertgenerierung** - Es werden Messwerte erzeugt oder berechnet.
4. **Speichern** - Die Messwerte werden persistiert.
5. **Bereitstellen** - Die Bereitstellung der Messwerte erfolgt über den Message Broker.

Je nachdem, ob ein Microservice Rohdaten verarbeitet oder Aggregate bildet, werden verschiedene Operationen in diesen Schritten durchgeführt. Im Folgenden wird der konkrete Aufbau der zwei verschiedenen Typen von Microservices beschrieben. Der Aufbau beider Arten ist in Abbildung 5.5 zu sehen. Hier können die Unterschiede in der Verarbeitung nachvollzogen werden. Die dargestellten Komponenten sind mit Nummern versehen. Zusätzliche Erläuterungen zu diesen sind in Tabelle 5.1 zu finden. Hier werden die Aufgaben der Verarbeitungsschichten beschrieben.

Die Einteilung der Anwendungen in Schichten bietet viele Vorteile. Jede Schicht bildet eine Abstraktionsebene und kapselt einen Verarbeitungsschritt. Dadurch wird die Komplexität des Gesamtsystems reduziert. Abhängigkeiten bestehen zwischen den interagierenden Schichten statt auf Systemebene. Ein großer Vorteil stellt die Trennung nach Aufgaben dar. Jede Schicht erfüllt nur eine bestimmte Funktion. Der Gesamtprozess wird durch das Durchlaufen der Schichten abgebildet. Man spricht von *Separation of Concerns (SoC)* [83]. Durch die lose Kopplung und die hohe Kohäsion einzelner Schichten ergibt sich die Möglichkeit, diese separat voneinander zu entwickeln. Darüber hinaus lassen sich Schichten austauschen, ohne die restliche Anwendung zu verändern [12]. Dieser Vorteil machte sich bei der Implementierung mehrmals bemerkbar. Zum Beispiel wurde anfangs für die Receiver-Logik MQTT zur Subscription verwendet. Dies konnte später mit geringem Aufwand zu AMQP umgestellt werden. Dazu musste lediglich die Logik der entsprechenden Schicht angepasst werden.

Die Services und JPARepositories stellen die Datenschicht dar. Sie realisieren die Arbeit mit den im vorigen Kapitel beschriebenen Entitäten. Durch die Aufteilung von Zugriffoperationen in Repository und Service wird eine zusätzliche Abstraktion erzielt. Repositories bilden lediglich Logik ab, die grundlegende Datenbankzugriffe darstellen. Zusätzliche logische Schritte, wie Plausibilitätsprüfungen oder weitere notwendige Operationen, erfolgen in der Service-Schicht. Ein Beispiel ist bei einer Löschung die korrekte Auflösung von Beziehungen zu anderen Entitäten. Falls wichtige Abhängigkeiten bestehen, kann eine Löschung verhindert werden. Anschließend kann ein entsprechender Fehler erzeugt und ausgegeben werden.

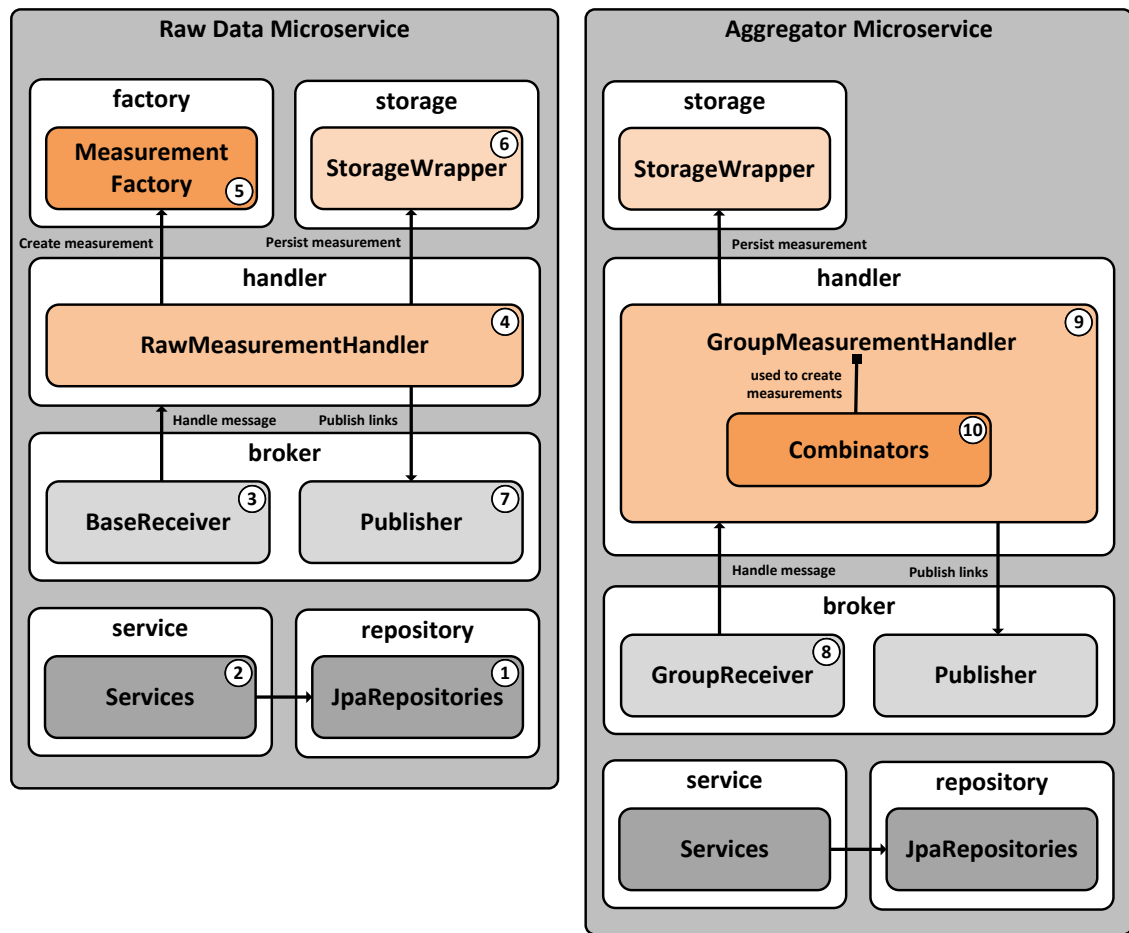


Abbildung 5.5: Bausteinsicht Microservices

5.2.3 Automatische Skalierung

Als Anforderung wurde definiert, dass je nach Auslastung automatisch mehr oder weniger Programminstanzen gestartet werden sollen. Kubernetes als Zielplattform stellt mit seiner Ressource *Horizontal Pod Autoscaler* eine flexible Möglichkeit bereit, ein solches Verhalten abzubilden. In Kapitel 4.3 wird eine verwandte Arbeit beschrieben, die als Referenz genutzt wurde, um eine automatische Skalierung abzubilden. Hiermit wurde jedoch nur das Verständnis einer simplen Anwendung ermöglicht. Der konkrete Anwendungsfall ist jedoch komplexer. Durch die gegebenen Verweise auf weitere Literatur und Dokumentation konnte ein eigenes Konzept aufgestellt werden.

Hierbei war die erste konzeptionelle Hürde das Finden einer Metrik, die eine präzise Skalierung ermöglicht. Der simpelste Weg ist die Skalierung von Microservices anhand ihrer Queues. Jeder Microservice hat eine dedizierte Queue, aus der er Nachrichten entnimmt. Die Betrachtung dieser ermöglicht Rückschlüsse auf die Auslastung der Microservices. Es sind nicht genug Services aktiv, wenn viele unverarbeitete Nachrichten vorliegen. In diesem Fall müssen mehr Programminstanzen gestartet werden. Diese Skalierung wurde prototypisch umgesetzt. Dabei fiel das Problem des Verfahrens auf. Verwendet man die Größe der Queues als Metrik, so lässt sich lediglich eine Aufwärts-Skalierung durchführen. Sind keine Nachrichten mehr vorhanden, kann dies bedeuten, dass alle Microservices gut ausgelastet, aber noch nicht überlastet, sind. Es kann aber auch sein, dass die Last wieder stark gesunken ist und weniger Replikate die Arbeit ebenso bewältigen könnten. Die Anzahl verbleibender Nachrichten in der Queue ist nicht ausreichend aussagekräftig, um

Tabelle 5.1: Beschreibung der Komponenten von Abbildung 5.5

Nr.	Beschreibung
1	Die <i>JPARpositories</i> stellen <i>CRUD-Funktionalität</i> im <i>Hibernate-Kontext</i> bereit.
2	Die <i>Services</i> beinhalten Geschäftslogik. Sie greifen auf die <i>Repositories</i> zur Datenabfrage zu.
3	Der <i>BaseReceiver</i> baut eine Verbindung mit dem Message Broker auf und registriert die notwendigen Bindings. Er übergibt ein <i>Callback</i> an den Message Broker. Hierbei handelt es sich um eine Rückruffunktion zur Verarbeitung der ankommenden Nachrichten.
4	Der <i>RawMeasurementHandler</i> wird vom <i>Callback</i> im Falle eines Nachrichteneingangs aufgerufen. Er realisiert die Verarbeitung der Nachricht beginnend mit der Erzeugung des Messwerts und dem anschließendem Speichern. Im letzten Schritt werden Referenzen auf die Messwerte als URLs veröffentlicht.
5	Die <i>MeasurementFactory</i> verarbeitet den Nachrichteninhalt und erzeugt einen Messwert des jeweiligen Datentyps.
6	Der <i>StorageWrapper</i> stellt Funktionalität zum Speichern von Messwerten und zur Zwischenspeicherung zwecks Aggregatbildung bereit.
7	Der <i>Publisher</i> baut eine Verbindung zum Message Broker auf. Er realisiert die Veröffentlichung der Daten.
8	Der <i>GroupReceiver</i> baut gleich dem <i>BaseReceiver</i> eine Verbindung zum Message Broker auf. Die beiden Receiver-Klassen unterscheiden sich jedoch in ihrer Funktionalität. Der <i>GroupReceiver</i> verwendet unterschiedliche Exchanges und Bindings sowie einen anderen Aufbau der Routing Keys. Die Funktionalität des <i>Callbacks</i> ist ebenso verschieden.
9	Der <i>GroupMeasurementHandler</i> wird analog zum <i>RawMeasurementHandler</i> aufgerufen. Er sorgt für das Caching der Messwerte bis alle Daten zur Aggregatbildung vorliegen. Dann erzeugt er die verschiedenen Aggregatmesswerte mithilfe der jeweiligen Kombinatoren. Diese werden dann jeweils persistiert und gegebenenfalls erneut veröffentlicht.
10	Ein <i>Combinator</i> stellt eine Funktion dar, die aus einer Menge von Messwerten einen neuen Messwert generiert.

ein sinnvolles Herunterskalieren auszuführen.

Eine andere Metrik, die evaluiert wurde, ist die Anzahl an verarbeiteten Werten in einem festen Zeitintervall. Würde sich ein Durchschnittswert gut abschätzen lassen, so könnte dann basierend auf diesem eine Auslastung bestimmt werden. Hier ergibt sich jedoch der Nachteil, dass die Microservices sehr unterschiedliche Daten verarbeiten. Die Daten können von Wahrheitswerten über Zahlen bis hin zu Bildern verschiedenster Größenordnungen variieren. Aggregierende Microservices können simple Durchschnittswerte über kleine Gruppen bilden. Es sollen aber gleichermaßen komplexe und gegebenenfalls zeitintensive Algorithmen zur Berechnung von Aggregaten verwendet werden können. Diese Dynamik macht es notwendig, dass für jeden Microservice erneut evaluiert werden müsste, wie viele Daten verarbeitet werden können. Dieser zusätzliche Aufwand soll nach Möglichkeit erspart bleiben.

Die Entscheidung fiel auf eine Metrik, die die Auslastung als Verhältnis von der Zeit in Ausführung zur inaktiven Zeit darstellt. Diese Aktivitätszahl wird innerhalb der Micro-

services berechnet und von diesen bereitgestellt. Die Berechnung erfolgt durch folgenden Quotienten:

$$microservice_activity_{percent} = \frac{active_time_{recent_minute}}{total_time_{recent_minute}} \cdot 100$$

Mit:

$microservice_activity_{percent}$ = Aktivitätsmetrik in Prozent
 $active_time_{recent_minute}$ = Zeit aktiver Verarbeitung in der letzten Minute in ms
 $total_time_{recent_minute}$ = Gesamtzeit in der letzten Minute in ms

Die Metrik wird so bestimmt, dass sie stets den aktuellen Wert für die letzte Minute darstellt. Anschließend wird sie per Pod über REST an Prometheus bereitgestellt. Der Vorteil dieser Metrik gegenüber den Alternativen ist, dass sie unabhängig von der Datenverarbeitung des jeweiligen Microservice ist. Zudem lässt sich mit ihr auch eine Abwärts-Skalierung realisieren. Der Gesamtaufbau ist in Abbildung 5.6 zu sehen. Das *Deployment* des Microservices definiert und erzeugt ein *ReplicaSet*, welches die verschiedenen Pods verwaltet. Ein *ServiceMonitor* wird erstellt, der dem *Prometheus Operator* mitteilt, dass die Pods abgefragt werden müssen. *Prometheus* sammelt in diesem Fall alle dreißig Sekunden die Aktivitätsmetrik von jedem Pod. Der *Prometheus Adapter* erweitert den Kubernetes *Metrics API Server*, um die von Prometheus gesammelten Metriken. Dadurch werden diese in der API abrufbar und sind somit für den *Horizontal Pod Autoscaler* verfügbar. Dieser bildet einen Mittelwert über alle Aktivitätswerte entsprechend der folgenden Formel [42]:

$$desiredReplicas = ceil[currentReplicas \cdot (currentMetricValue \div desiredMetricValue)]$$

Mit:

$desiredReplicas$ = Anzahl an gewünschten Replikaten nach Aktualisierung
 $currentReplicas$ = Anzahl aktuell laufender Replikate
 $currentMetricValue$ = Aktueller Wert der Metrik
 $desiredMetricValue$ = Zielwert der Metrik

Im Vergleich mit einem gegebenen Zielwert kann der HPA dann Entscheidungen bezüglich der Skalierung des Deployments treffen. Der Zielwert soll so gewählt werden, dass noch ein Puffer besteht. Das heißt, es ist ein Zustand anzustreben, bei dem jede Anwendung stark ausgelastet ist. Ein Zustand der vollen Auslastung oder gar Überlastung ist jedoch zu vermeiden. Dadurch soll eine rechtzeitige Reaktion auf steigende Last ermöglicht werden. Es wird aufwärts skaliert, bevor das System nicht mehr alle Daten verarbeiten kann. So kann weiterhin auf plötzliche Ausnahmefälle reagiert werden, da stets ungenutzte Ressourcen zur Verfügung stehen.

5.2.4 Historische Aggregation

Die historische Aggregation soll ermöglichen, dass Aggregate über längere Zeiträume gebildet werden können. Dazu sollen wiederum Microservices implementiert werden, die Aggregate zu einem Datentyp bilden. Es sind erstmals stündliche, tägliche, monatliche und jährliche Aggregate angedacht. Um deren Bildung zu vereinfachen, sollen die Daten im Object Storage nach einer bestimmten Verzeichnishierarchie abgelegt werden. Ein Messwert eines Sensors soll unter dem eindeutigen Sensornamen, gefolgt von der

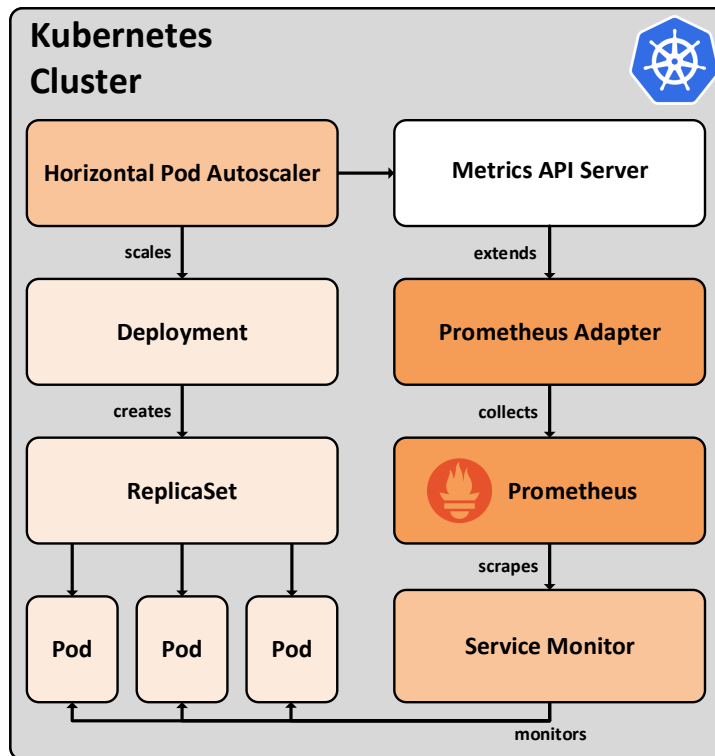


Abbildung 5.6: Konzept automatische Skalierung (Grafik orientiert an [76])

Jahres-, dann der Monats-, Tages- und Stundenkomponente des Datums abgelegt werden. Als Namen des Objekts kann dann die Minuten- und Sekundenkomponente verwendet werden. Da zur historischen Aggregation Messwerte aus dem MinIO Object Storage ausgelesen und verrechnet werden müssen, hilft diese Hierarchie enorm. Das *MinIO-SDK* stellt Funktionalität bereit, mit der alle Objekte selektiert werden können, die einen gemeinsamen Präfix teilen. In diesem Fall können mit dem Präfix alle zur Berechnung des jeweiligen Aggregats benötigten Einzelmesswerte abgerufen werden. Die erzeugten Aggregate können dann in der Verzeichnishierarchie eingebunden werden. Dadurch können diese wiederum zur Berechnung von weiteren historischen Aggregaten oder zur Ansicht verwendet werden.

5.2.5 Frontend - Visualisierung durch Dashboards

Eine wesentliche nichtfunktionale Anforderung ist die Abbildung der Sensormesswerte in einem Grafana Dashboard. Dazu soll Prometheus als Bindeglied verwendet werden. Während der Konzeption fiel auf, dass dies mit der geplanten Architektur nicht direkt abbildbar ist. Prometheus nutzt einen Pull-Mechanismus zur Abfrage von Metriken. Microservices sollen zustandslos sein. Die Rohdaten und Aggregate werden in Echtzeit verarbeitet. Bei einem Pull kann ein Microservice keine Auskunft über vorherige Verarbeitungen geben. Dementsprechend liegen bei einer Abfrage keine letzten Messwerte vor. Das Hinterlegen von Daten innerhalb eines Microservices würde dazu führen, dass dieser einen Zustand hätte. Es musste also ein Weg gefunden werden, um die Messwerte in Prometheus verfügbar zu machen.

Zuerst wurde deshalb der *Prometheus Push Gateway* gesichtet. Hierbei handelt es sich um eine Schnittstelle. Diese ermöglicht das Einpflegen von Daten durch einen Push. Der Push Gateway ist für eine Verarbeitung von Daten aus kurzlebigen Jobs oder Batch-Prozessen

gedacht. Er soll jedoch nicht eingesetzt werden, um Prometheus in ein push-basiertes Monitoring-System umzuwandeln. Dementsprechend eignet er sich nicht als Lösung des Problems.

Als weitere Möglichkeit wurde die Nutzung von Push-Monitoring-Lösungen evaluiert. Systeme wie *Graphite* oder *InfluxDB* stellen Alternativen zu Prometheus dar und können gleichermaßen als Datenquelle für Grafana verwendet werden. Man könnte auch ein push-basiertes Monitoring vorschalten, welches die Daten wiederum an Prometheus exportiert. Software, welche einen Export zu Prometheus ermöglicht, ist für alle gängigen Monitoring-Systeme verfügbar. Diese könnte genutzt werden, um die Daten aus anderen Systemen nach Prometheus zu exportieren. Eine solche Konstellation würde die Architektur und die notwendige Konfiguration komplexer machen. Diese zusätzliche Komplexität ist jedoch ungerechtfertigt, da keine klaren Vorteile entstehen.

Deshalb wurde beschlossen, *Memcached* zu verwenden, um letzte Messwerte für Sensoren und Aggregatoren zu speichern. Wird ein neuer Messwert erhoben, so wird dieser im Cache unter einem eindeutigen Schlüssel persistiert. Ein weiterer Microservice soll erstellt werden, der eine Schnittstelle für Prometheus anbietet, bei welcher aktuelle Messwerte abgefragt werden können. Dazu liest er alle zu exportierenden Sensoren und Aggregatoren aus den Stammdaten aus. Anschließend werden die letzten Messwerte aus dem Cache abgefragt. Diese können dann als Metriken in dem von Prometheus akzeptierten Format über eine REST-API angeboten werden. Die Messwerte können außerdem mit Metadaten angereichert werden. Im ersten Schritt sollen hier bei den Rohdaten der Datentyp und der Sensorname als Prometheus Label gesetzt werden. Bei den Aggregaten soll der Datentyp sowie der Gruppen- und Kombinatorname zur Auswertung genutzt werden können. Während der Implementierung sollen noch weitere Metadaten hinzukommen, welche beispielsweise eine Auswertung nach Standort ermöglichen. Die Labels können in Prometheus zur Spezifikation von Abfragen genutzt werden. Somit kann eine gezielte Anzeige von Daten in den Grafana Dashboards realisiert werden. In Abbildung 5.7 kann dieses Konzept nachvollzogen werden.

Es ist ebenso angedacht, ein Monitoring Dashboard für die Microservices anzubieten. Dieses soll Informationen, wie beispielsweise deren Auslastung, anzeigen. Mit dem Konzept für die Skalierung stehen bereits Aktivitätsmetriken bereit. Weitere Metriken können beispielsweise durch die Verwendung von *Spring Boot Actuator* bezogen werden [91]. Das Modul stellt eine Vielzahl von Metriken bereit, ohne dass ein zusätzlicher Implementierungsaufwand notwendig ist. Alle Metriken werden von Prometheus über die REST-Endpunkte abgefragt und stehen dann in Grafana zur Visualisierung bereit.

Das Caching der Messwerte durch *Memcached* ermöglicht deren Verfügbarkeit. Es wird die Abfrage der letzten Messwerten von Sensoren und Aggregatoren ermöglicht. Als Folge muss man jedoch Abstriche bei der Konsistenz machen. Es ist nicht gewährleistet, dass ein letzter Wert auch wirklich den aktuellsten Datensatz darstellt. Zum Zeitpunkt der Abfrage kann bereits ein neuer Messwert in Verarbeitung sein. Nach dem *CAP-Theorem* gibt es drei grundlegende Anforderungen an ein verteiltes System. Diese sind definiert durch die Abkürzung *Consistency, Availability and Partition Tolerance (CAP)* als Konsistenz, Verfügbarkeit und Ausfalltoleranz. Das Theorem besagt, dass ein System immer nur gleichzeitig zwei dieser Anforderungen erfüllen kann [36]. Für die Thesis sind Verfügbarkeit und Ausfalltoleranz zu gewährleisten. Die Folge sind Einschränkungen der Konsistenz. Im Bereich Cloud Computing wird häufig diskutiert, wie sich die Verfügbarkeit verteilter Systeme erhöhen lässt und man gleichzeitig einen gewissen Grad der Konsistenz beibehalten kann. Wenn keine Garantie der Neuheit von Daten besteht, aber gewährleistet werden kann, dass Aktualisierungen zu späterem Zeitpunkt bereitstehen, spricht man von *Eventual Consistency* [107] [75]. Eine solche Form der Inkonsistenz ist

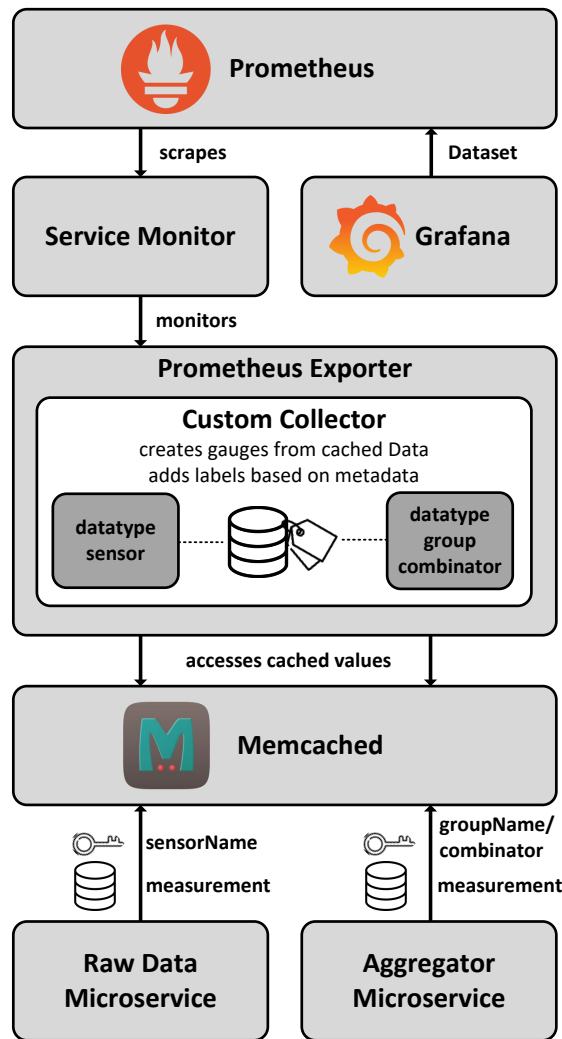


Abbildung 5.7: Konzept Prometheus Exporter

für dieses System tolerierbar. Es hängen keine kritischen Vorgänge an der Gegenwartigkeit der Messungen. Prometheus fragt in kleinen Zeitintervallen die Messwerte ab. Wird ein älterer Messwert bereitgestellt, bedeutet dies meist nur eine kleine Abweichung für ein ebenso kleines Zeitfenster. Die Sensoren im IoT-Umfeld sollen vor allem über längere Zeiträume ausgewertet und analysiert werden. Für die Metriken fallen daher die auftretenden Inkonsistenzen nicht auf und können vernachlässigt werden.

Innerhalb der Microservices muss bei der Implementierung beachtet werden, dass das Schreiben in den Cache möglichst früh nach erfolgter Verarbeitung durchgeführt wird. Dadurch können die zuvor beschriebenen Inkonsistenzen minimiert werden. Nach der Verarbeitung erfolgen im Wesentlichen noch zwei weitere Prozesse. Zuerst werden die erzeugten Messwerte im Object Storage persistiert. Anschließend werden sie über AMQP erneut zur Weiterverarbeitung publiziert. Das Erzeugen der Schlüssel-Wert-Paare im Memcached-Speicher kann vor der Publikation erfolgen.

Da die Gesamtarchitektur skalierbar sein muss, ist auch die Skalierbarkeit des Microservices zur Datenbereitstellung an Prometheus zu konzipieren. Der zeitkritische Aspekt der Anwendung liegt bei der Routine zur Abfrage der Messwerte aus dem Cache. Hierbei muss zur Skalierung die Menge an bereitgestellten Daten partitioniert werden. Jede Programminstanz soll dann einen Teil der Messwerte bereitstellen. Die Einteilung kann auf

5 Konzeption

verschiedenen Wegen durchgeführt werden. Es wäre eine Kategorisierung nach Datentypen denkbar. Gleichmaßen könnte man Producer nach ID-Spannen aufteilen. Nach erfolgter Partitionierung kann man eine *Primary-Secondary-Architektur* einsetzen. Die Secondaries führen parallel eine Abfrage der Daten für den Primary aus. Dieser setzt die Daten aller Secondaries zusammen und stellt sie über REST zur Verfügung. Die Umsetzung soll erstmals nur ein prototypisches Konzept umfassen, welches die Erweiterung bis hin zu einer solchen Architektur ermöglicht. Das heißt, die Umsetzung umfasst ein Secondary, welches durch manuelle Konfiguration skaliert werden kann.

6 Implementierung

Die Implementierung der im fünften Kapitel konzipierten Komponenten wurde mit *Java* durchgeführt. Hier wurde das *Spring Boot Framework* verwendet, welches die Entwicklung von Microservices erleichtert [90]. Java wird häufig zur Implementierung von Microservices-Architekturen eingesetzt. Insbesondere die *Annotation Syntax* und Frameworks wie Spring vereinfachen die Implementierung und die Lesbarkeit des Quellcodes. Das konzipierte System benötigt viele externe Anwendungen zur Abbildung der gewünschten Funktionalität. Aus diesem Grund bietet sich Java an. Durch die weite Verbreitung stehen viele Ressourcen und Bibliotheken zur Verfügung. Gleichmaßen können SDKs zur Anbindung der Fremdsoftware genutzt werden. Zur Verwaltung des Build-Prozesses wird *Apache Maven* verwendet. Hier wird jedes Modul durch eine zentrale *POM* verwaltet, die alle wesentlichen Informationen für den Erstellungsprozess beinhaltet. Die Maven-Lebenszyklen können zur Durchführung der Tests sowie zum Deployment im lokalen und produktiven System verwendet werden. Maven legt für jedes Projekt dessen Struktur, Abhängigkeiten und Versionierung fest. Dadurch wurde die Implementierung und die Bereitstellung der vielen einzelnen Microservices erleichtert.

In diesem Kapitel werden die wesentlichen Phasen beschrieben, die zur Implementierung durchlaufen wurden. Zuerst wurde die Anbindung der Sensoren an die Raspberry Pis realisiert. Anschließend wurde die gemeinsame Code-Basis implementiert. Diese soll von allen datenverarbeitenden Microservices als Grundlage verwendet werden. Exemplarisch wurden dann einige Microservices entwickelt, die die Code-Basis abstrahieren. Die Anwendungen zur historischen Aggregation sowie zur Datenbereitstellung an Prometheus wurden im Anschluss umgesetzt. Im Folgenden wird jeweils die Durchführung der Implementierung der Module dargestellt. Hierbei werden wichtige Implementierungsdetails und notwendige Abweichungen vom ursprünglichen Konzept erläutert.

6.1 Sensoranbindung für die Raspberry Pis

Da der Fokus der Thesis auf der Backend-Architektur liegt, wurde die Logik zur Datengewinnung auf Sensorseite nur prototypisch implementiert. Die Anbindung der *GrovePi-Sensoren* an die Raspberry Pis konnte durch eine Java-Anwendung realisiert werden. GrovePi stellt eine Bibliothek bereit, die bereits Implementierungen von Sensoranbindungen bereitstellt. Hier werden aber nur einige Sensoren exemplarisch angebunden. Mit den gegebenen Beispielen war jedoch das Konzept zur Anbindung der analogen und digitalen Sensortypen nachvollziehbar. Die Sensoren müssen dazu angeschlossen und der jeweilige Port in der Konfiguration festgehalten werden. Für jede Sensorart steht Logik bereit, die in einem anpassbaren Intervall Abfragen tätigt. Messwerte werden über MQTT durch Verwendung der *Eclipse Paho Library* [28] an den *RabbitMQ Broker* gesandt. Hierbei sendet ein Sensor an ein dediziertes Topic. Dieses setzt sich aus dem Präfix des Sensortyps und seinen Sensornamen zusammen. Ein Sonderfall stellte der Sensor vom Typ *Temperature&Humidity* dar, der sowohl Temperatur- als auch Luftfeuchtigkeitswerte bestimmt. Hier wurde bei der Implementierung entschieden, beide Messwerte auf separaten Topics zu publizieren. Dadurch bleibt bei den verarbeitenden Microservices die Funktionalität klar definiert.

6.2 Erweiterbare Code-Basis

Die Implementierung der erweiterbaren Code-Basis ermöglicht die flexible und zeiteffiziente Neuentwicklung von Microservices zur Anbindung und Verarbeitung von Sensordaten. Im Anhang B.6 ist eine Übersicht der Pakete zu finden sowie deren Relationen. Im Folgenden wird auf den Aufbau der einzelnen Pakete gesondert eingegangen und die Implementierung der inbegriffenen Klassen erläutert.

6.2.1 Entitäten und Modelle

Es folgt die Beschreibung der *Entitäten* und *Modelle*. Als Entitäten werden hierbei Objekte bezeichnet, die in der Datenbank persistiert werden. Sie sind also Teil der Stammdaten. Die Entitäten werden durch die *JPA Repositories* verwaltet. Als Modelle werden alle weiteren Datenklassen bezeichnet, die zur Abbildung der Logik benötigt werden. Dazu zählen zum Beispiel Klassen, die zur Aggregatbildung genutzt werden. Ebenfalls sind hier die Klassen zur Abbildung der Messwerte im Object Storage inbegriffen.

6.2.1.1 Entitäten

Die Entitäten wurden entsprechend dem in Kapitel 5.2.1 beschriebenen Konzept implementiert. Es fanden nur geringfügige Erweiterungen hinsichtlich der Attribute der Klassen statt, um eine größere Vielfalt an Stammdaten anzubieten. Ebenso wurden neue Eigenschaften hinzugefügt, die eine gezielte Steuerung der Logik durch Konfiguration ermöglichen. So wurde beispielsweise die Entität *Producer* um ein Attribut erweitert, welches festlegt, ob ein Export an Prometheus zu erfolgen hat. Dadurch kann dynamisch definiert werden, ob Daten im Monitoring angezeigt werden. Bilddaten zum Beispiel können schlichtweg nicht mit Grafana visualisiert werden, da sie nicht dem Format der akzeptierten Metriken entsprechen. Folglich können Sensoren, die Bilddaten generieren, per Konfiguration vom Export ausgeschlossen werden. Außerdem wurde das Konzept der Metadaten weiter ausgebaut. Eine *Location*-Entität wurde gebildet, die Standortinformationen abbildet. Diese kann an einem Producer hinterlegt werden. Dadurch wird es möglich sowohl einem Sensor als auch einem Aggregator einen Standort zuzuordnen. Nach diesem Ort kann dann in den Dashboards gefiltert werden. Das Klassendiagramm zur finalen Implementierung der Entitäten ist im Anhang B.7 zu finden.

CRUD-Operationen ermöglichen das Speichern, Abfragen und Löschen der Entitäten. Diese Logik wird durch die Implementierung von *JPA Repositories* bereitgestellt. Die Repositories sind Bestandteil des *Spring Data JPA* Frameworks. Hier sind grundlegende Operationen bereits implementiert. Diese können ohne Implementierungsaufwand genutzt werden. So steht beispielsweise bereits Logik für das Speichern und Löschen sowie für simple Abfragen bereit. Weitere Funktionen lassen sich über den *Query Derivation Mechanismus* abbilden, indem Methoden entsprechend der gegebenen Konvention benannt werden. So können häufig verwendete Abfragen ohne Entwicklungsaufwand erstellt werden. Ein Beispiel ist die Ausgabe aller Gruppen, die als aktiv gekennzeichnet sind. Komplexe Abfragen werden mittels *JPQL* realisiert. Diese Abfragesprache ermöglicht das Schreiben von Logik auf Anwendungsebene. Dadurch entsteht eine Entkopplung von der Abfragesprache der Datenbank. Zudem sinkt der Implementierungsaufwand. Im Service-Layer werden die Repositories genutzt, um Geschäftslogik abzubilden. Die Services erweitern die Logik zur Interaktion mit der Datenbank. Komplexere Methoden können abgebildet werden, welche beispielsweise mehrere atomare Operationen benötigen. Gleichmaßen können Plausibilitätsprüfungen durchgeführt werden. Hierbei stellen die Services eine zusätzliche Abstraktionsebene dar. Dadurch wird eine höhere Modularität

erreicht. Repositories und Services können mit dem *Spring Boot Test Framework* auf korrekte Funktionalität geprüft werden. Dazu wurden *Unit-Tests* mit *Mockito* erstellt. Hier werden externe Abhängigkeiten durch *Mocks* ersetzt. Dadurch wird nur der gewünschte Code getestet, während Abhängigkeiten eliminiert werden. Ebenfalls wird zum Testen nicht die herkömmliche Datenbank verwendet. Stattdessen wird bei jedem Durchlauf eine *H2-Datenbank* erzeugt und mit Testdaten initialisiert. Da es sich hierbei um eine *In-Memory-Datenbank* handelt, wird diese nach erfolgtem Test einfach gelöscht. Dadurch sind die Tests in sich geschlossen und wiederholbar.

6.2.1.2 Modelle

Es wurden Modell-Klassen implementiert, welche bei den internen Programmabläufen zur Datenkapselung verwendet werden. In Abbildung 6.1 ist das Konzept dargestellt. Aufseiten der weiterverarbeitenden Microservices wird zur Abbildung der Aggregatsfunktionen die *CombinatorModel-Klasse* verwendet. Hierbei handelt es sich um eine generische Klasse, die einen Namen sowie eine *CombinatorFunction* mit gleichem Typparameter beinhaltet. Hiermit wird eine Verknüpfung der *Combinator-Entität* mit einer konkreten, durch die Code-Basis oder den jeweiligen Microservice definierten Funktion ermöglicht. Die *CombinatorFunction* wird durch ein funktionales Interface repräsentiert, welches ein generisches Ergebnis erzeugt. Dazu wird eine Operation auf einem *GroupMeasurementStore* eines generischen Typs durchgeführt. Eine Schnittstelle kann als funktionales Interface deklariert werden, indem eine entsprechende Annotation gesetzt wird. Sie darf nur über genau eine abstrakte Methode verfügen. Ist dies gegeben, so kann die Schnittstelle mit Lambda-Ausdrücken verwendet werden [103]. Dadurch wird die Erstellung von Funktionen zur Aggregation flexibel und kompakt. Für den Fall, dass die Typparameter von Eingabe und Ausgabe gleich sind, steht das *CombinatorUnaryModel* sowie das *CombinatorUnaryOperator-Interface* zur Verfügung. Diese ermöglichen eine vereinfachte Implementierung von Gruppierungen gleicher Sensortypen. Die funktionalen Interfaces werden durch Standard-Implementierungen wie Maximum-, Minimum- und Durchschnittsberechnungen für Datentypen wie Double oder Integer implementiert. Sie können aber auch für komplexe Formeln verwendet und in den Microservices selbst definiert werden.

Der *GroupMeasurementStore* ist ein Modell, welches zur Sammlung von Messwerten eingesetzt wird. Bei einer Gruppierung kann ein Aggregat erst berechnet werden, wenn alle in der Gruppe befindlichen Sensoren einen Messwert gemeldet haben. Der *GroupMeasurementStore* ist so aufgebaut, dass er Informationen zur Gruppe beinhaltet, deren Aggregat gebildet werden soll. Zudem beinhaltet er eine *Map*. Diese nutzt die IDs der Produzenten als Schlüssel und hinterlegt dazu deren Messungen als Werte. Die Sammlung der aktuell vorliegenden Messwerte erfolgt durch Objekte vom Typ *TempGroupMeasurement*. Diese werden serialisiert und im Cache zwischengespeichert. Sie beinhalten einen *GroupMeasurementStore* sowie eine maximale Größe, welche entsprechend der Anzahl an Gruppenmitgliedern gesetzt wird. Dadurch lässt sich beim Ankommen neuer Messwerte schnell verifizieren, ob die Bildung der Aggregate durchgeführt werden kann oder ein erneutes Speichern notwendig ist. Bei einer Aggregatbildung kann man dem *TempGroupMeasurement* ein *CombinatorModel* zuweisen. Dieses generiert dann auf Basis des *GroupMeasurementStores* einen neuen Messwert. Messwerte werden generell durch das Model *Measurement* abgebildet. Da flexible Datentypen als Messwerte verarbeitet werden müssen, bietet sich auch hier die Verwendung einer generischen Klasse an. Nach der Berechnung des Messwerts wird dieser dann im Object Storage persistiert. Anschließend wird er erneut über AMQP publiziert und nach Prometheus exportiert. Das finale Klassendiagramm kann dem Anhang B.8 entnommen werden.

Die Implementierung des *Serializable-Interfaces* ermöglicht hier die flexible Serialisierung und Deserialisierung in den Object Storage. Mit Annotations kann die Umwandlung in das JSON-Format zusätzlich spezifiziert werden. Es ist möglich, Attribute umzubenennen, auszuschließen oder Formate festzulegen. So kann beispielsweise das Datumsformat zur Serialisierung des Zeitpunkts der Messung angegeben werden.

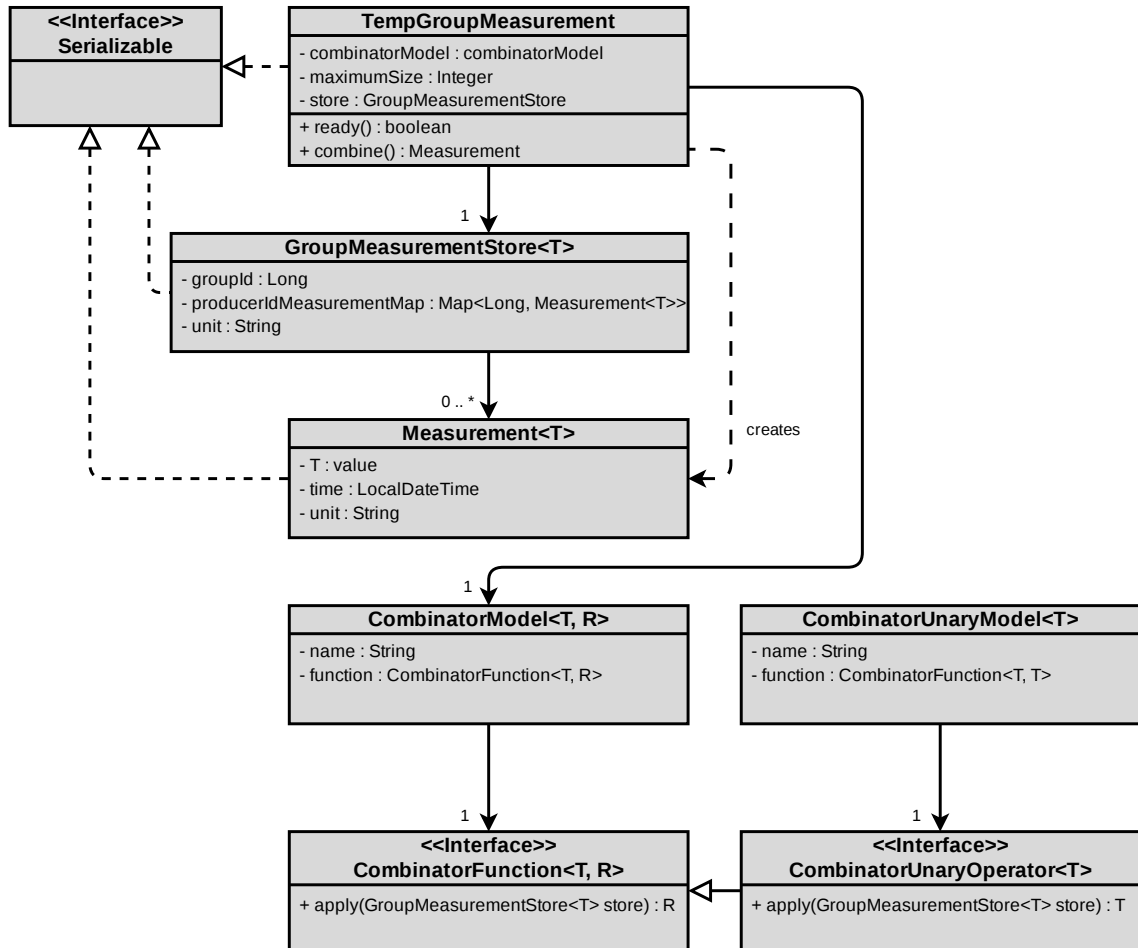


Abbildung 6.1: Models zur Aggregation

6.2.2 Generalisierung grundlegender Abläufe

Bei der Erstellung der Code-Basis wurde ein großer Fokus daraufgelegt, dass aufbauende Microservices schnell und bequem erstellt werden können. Dazu muss die Mehrheit der Logik hier implementiert werden. Die Microservices sollen nur ihre eigenen Besonderheiten abbilden müssen. Für rohdatenverarbeitende Microservices muss nur die Generierung des Messwerts aus einer Nachricht vom Datentyp `String` definiert werden. Die weiterverarbeitenden Microservices müssen die verschiedenen Funktionen zur Aggregatbildung angeben. Grundlegende Abläufe können somit in Oberklassen beziehungsweise abstrakten Klassen abgebildet und durch Vererbung bereitgestellt werden. Es bleibt in Sonderfällen möglich, über die Erweiterungspunkte hinaus durch Überschreiben von Kernfunktionalität auch andere Änderungen durchzuführen. Damit ist ein grundlegendes Schema gegeben, welches eine flexible Implementierung der Mehrheit an Anwendungsszenarien ermöglicht. Gleichmaßen bleibt die Umsetzung komplexer Anwendungsfälle machbar.

6.2.3 Konfiguration

Die Konfiguration der Microservices erfolgt über Spring *PropertySources*. POJOs werden als Konfiguration markiert und von *Spring Beans* abgefragt. Dadurch kann eine Konfiguration beispielsweise durch Dateien mit Schlüssel-Wert-Paaren erfolgen. Mit der *Value-Annotation* wird ermöglicht, einem Attribut einen Schlüssel zuzuordnen. Ist nun eine Konfiguration vorhanden, die mit dem Schlüssel übereinstimmt, so wird das Attribut entsprechend dem Wert gesetzt. Während der Entwicklung konnten so die *application.properties* Dateien verwendet werden, um einen Microservice zu konfigurieren. Eine Besonderheit der Value-Annotation ist, dass im Falle der Existenz einer gleichnamigen Umgebungsvariable diese präferiert wird. Dadurch wird bei einem Deployment der Anwendung über Kubernetes eine Zentralisierung der Konfiguration ermöglicht. Dazu wird eine ConfigMap eingesetzt. Die Einträge der ConfigMap werden den Pods dann als Umgebungsvariablen zugeordnet. Dadurch lässt sich die Konfiguration von außerhalb überschreiben. Die konkrete Umsetzung ist im Deployment der Microservices in Kapitel 6.3.2 beschrieben.

Um die Konfiguration möglichst flexibel zu gestalten, wurden Interfaces implementiert, die Konfigurationsparameter zu den externen Anwendungen kapseln. So befinden sich beispielsweise alle Daten, welche zum Aufbau einer Verbindung zum Broker benötigt werden in einer Schnittstelle. Gleichmaßen wurden Interfaces für die Konfiguration der Memcached- und Minio-Clients entwickelt. Die Verbindung zum MySQL-Server kann durch die Spring *DataSource-Konfiguration* festgelegt werden. Dementsprechend musste hierfür keine eigene Implementierung erfolgen. Außerdem besteht für jede Art von Microservice ein POJO, welches alle notwendigen Konfigurationen umfasst. Dieses POJO implementiert alle benötigten Interfaces und definiert darüber hinaus Namen für anwendungsspezifische Parameter. Über die Konfigurationsdatei oder über Kubernetes können diese Parameter gesetzt werden. Die Implementierung der Klassen zur Abbildung der Konfiguration ist im Anhang B.9 ausgliedert.

Zudem wurden *SetupDataLoader* implementiert. Diese beinhalten Logik, die beim Applikationsstart ausgeführt wird. Hier werden beispielsweise Prüfungen ausgeführt, ob Stammdaten wie der benötigte Datentyp oder Gruppentyp vorhanden sind. Im Falle des Nichtvorhandenseins werden diese angelegt, um eine korrekte Funktionalität des Microservices zu gewährleisten. Dadurch spart man sich die Notwendigkeit Datenbankskripte vor dem ersten Systemstart auszuführen. Durch Hibernate werden die Tabellen erzeugt. Im nächsten Schritt befüllen die *SetupDataLoader* diese mit den benötigten Einträgen.

6.2.4 Verarbeitungsschritte

Die Verarbeitungsschritte wurden entsprechend dem in Kapitel 5.2.2 beschriebenen Konzept realisiert. Hierbei wurden die Phasen Datenempfang, Nachrichtenverarbeitung, Datenspeicherung und Publikation definiert. Im Folgenden wird auf die Implementierung der einzelnen Schritte eingegangen.

6.2.4.1 Datenempfang

Der erste Schritt ist der Empfang einer Nachricht durch Konsumieren der dedizierten AMQP-Queue. Dazu muss zuerst eine Verbindung mit dem Message Broker aufgebaut werden. Für die Produktivumgebung bedeutet dies, dass ebenfalls der *Secure Sockets Layer (SSL)-Kontext* initialisiert werden muss. Bei SSL handelt es sich um den Vorgänger von TLS. Das Setzen des SSL-Kontexts ermöglicht den Schlüsselaustausch und die Authentisierung. Hierbei müssen drei Zertifikate verwendet werden. Das *CA-Zertifikat* dient der Identifikation der *Certificate Authority*. Der Authentifizierung gegenüber dem Server

dienen das *Client-Zertifikat* sowie der *Private-Key*. Die Kommunikation mit dem Server wird über *TLSv1.2* durchgeführt. Der SSL-Kontext muss explizit beim Verbindungsaufbau gesetzt werden. Es ist nicht möglich, den Vorgang mithilfe des Frameworks zu realisieren. In Listing 6.1 kann nachvollzogen werden, wie die Zertifikate zum Verbindungsaufbau zugewiesen werden.

```

1 // load CA certificate
2 X509Certificate caCert = null;
3 InputStream stream = new ByteArrayInputStream(
4     Base64.getDecoder().decode(caFile)
5 );
6 BufferedInputStream bis = new BufferedInputStream(stream);
7 CertificateFactory cf = CertificateFactory.getInstance("X.509");
8 while (bis.available() > 0) {
9     caCert = (X509Certificate) cf.generateCertificate(bis);
10 }
11 // load client certificate
12 stream = new ByteArrayInputStream(
13     Base64.getDecoder().decode(clientCertFile)
14 );
15 bis = new BufferedInputStream(stream);
16 X509Certificate cert = null;
17 while (bis.available() > 0) {
18     cert = (X509Certificate) cf.generateCertificate(bis);
19 }
20 // load client private key
21 byte[] keyArray = Base64.getDecoder().decode(clientKeyFile);
22 PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(keyArray);
23 KeyFactory kf = KeyFactory.getInstance("RSA");
24 PrivateKey privateKey = kf.generatePrivate(spec);
25 // CA certificate is used to authenticate server
26 KeyStore caKs = KeyStore.getInstance(KeyStore.getDefaultType());
27 caKs.load(null, null);
28 caKs.setCertificateEntry("ca-certificate", caCert);
29 TrustManagerFactory tmf = TrustManagerFactory.getInstance(
30     TrustManagerFactory.getDefaultAlgorithm()
31 );
32 tmf.init(caKs);
33 // client key and certificates are sent to server for authentication
34 KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
35 ks.load(null, null);
36 ks.setCertificateEntry("certificate", cert);
37 ks.setKeyEntry(
38     "private-key", privateKey, null, new Certificate[] { cert }
39 );
40 KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
41 kmf.init(ks, null);
42 // finally, create SSL socket factory
43 SSLContext context = SSLContext.getInstance("TLSv1.2");
44 context.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

```

Listing 6.1: Setzen des SSL-Kontexts

Die Logik zum Verbindungsaufbau ist in einer abstrakten Klasse namens *BrokerConnection* zu finden. Diese wird von allen anderen Klassen, die mit dem Broker interagieren, erweitert. Für den Datenempfang gibt es hier die *Receiver-Klasse*. Hierbei handelt es sich um eine weitere abstrakte Klasse, welche auf Basis der Konfiguration des Microservices die jeweilige Queue deklariert. Dadurch wird die Queue bei RabbitMQ erzeugt. Man spart sich die Notwendigkeit, diese manuell anzulegen und gewährleistet zum Applikationsstart deren Existenz. Durch die Anlage der Queues als *durable* wird das Überleben eines Broker Restarts gewährleistet. Zudem wird die *auto-delete* Funktionalität deaktiviert,

sodass im Falle eines Neustarts der Consumer die Queues ebenfalls bestehen bleiben. Außerdem werden wichtige Konfigurationen wie der *Prefetch Count* gesetzt. Hierbei handelt es sich um die Anzahl an Nachrichten, die ein Microservice auf einmal aus der Queue nehmen darf. Die Nachrichten werden im Regelfall einzeln nach Verarbeitung durch Senden eines *Acknowledgement-Frames* bestätigt. Erst mit der Bestätigung aller entnommenen Nachrichten, darf die Verarbeitung fortgesetzt werden. Mit einem Acknowledgement-Frame können aber auch mehrere Nachrichten auf einmal akzeptiert werden. Dazu muss die Mehrfachbestätigung aktiviert werden. Dadurch wird die Netzwerkkommunikation verringert. Die Konfiguration des Prefetch Counts zusammen mit der Mehrfachbestätigung stellt eine Optimierung dar. Nicht mehr mit jeder Verarbeitung muss das Senden eines Acknowledgement-Frames erfolgen. Sowohl Kommunikationsaufwand als auch Verarbeitungszeit werden reduziert. Die Initialisierung einer Queue ist in Listing 6.2 zu sehen. Hier ist ebenfalls die Verarbeitung der Nachrichten durch Aufruf der *Callback-Funktion* dargestellt.

Für jede eingehende Nachricht wird die benötigte Zeit zur Verarbeitung durch den *ActivityManager* bestimmt. Dieser speichert alle Verarbeitungszeiten der letzten sechzig Sekunden. Zudem beinhaltet er eine Routine, die alle Einträge, die älter als sechzig Sekunden sind, löscht. Die Logik des ActivityManagers ist einem Ringspeicher nachempfunden. Eine Aktivitätsmetrik kann zu jedem Zeitpunkt ausgegeben. Dazu dividiert der ActivityManager die Verarbeitungszeit durch die Gesamtzeit entsprechend der in Kapitel 5.2.3 beschriebenen Formel. Die Metrik steht über REST zur Abfrage bereit. Zur Bereitstellung wird ein *Spring Web MVC Controller* implementiert.

```

1 channel.queueDeclare(
2     applicationProperties.getMicroserviceQueue(),
3     true, //durable
4     false, //exclusive
5     false, //auto-delete
6     null
7 );
8 // set prefetch count - limits the amount of messages taken at once
9 channel.basicQos(Constants.PREFETCH_COUNT, false);
10 // will be invoked for every message taken from the queue
11 DeliverCallback deliverCallback = (consumerTag, delivery) -> {
12     long startTime = System.currentTimeMillis();
13     processMessage(
14         delivery.getEnvelope().getRoutingKey(),
15         new String(delivery.getBody(), "UTF-8")
16     );
17     //set multiple to true, so we can ack all fetched messages
18     channel.basicAck(delivery.getEnvelope().getDeliveryTag(), true);
19     long stopTime = System.currentTimeMillis();
20     this.activityManager.addTime(stopTime, stopTime - startTime);
21 };
22 // messages get manually acknowledged so, autoAck is set to false
23 channel.basicConsume(
24     applicationProperties.getMicroserviceQueue(),
25     false,
26     deliverCallback, consumerTag -> {}
27 );

```

Listing 6.2: Queue-Initialisierung und Callback-Funktion

Die Receiver-Klasse wird von der *BaseReceiver-Klasse* erweitert. Diese erzeugt nach Konfiguration Bindings vom Exchange *amq.topic* in die Microservice Queue. Das Routing wird anhand ebenso anpassbarer Topics durchgeführt. Der *GroupReceiver* wiederum legt ein Binding vom Exchange für die Gruppen in die Microservice Queue an. Zur Gruppierung wird hier der Exchange mit dem Namen *group_exchange* verwendet. Zum Routing dient

der Gruppentyp. Es erfolgt eine Weiterleitung basierend auf Wildcards. In Abbildung 5.2 in Kapitel 5.1.2.1 kann das zugrunde liegende Konzept nochmals betrachtet werden. Des Weiteren implementieren beide Klassen die Methode *processMessage* und bilden die Nachrichtenverarbeitung ab. Der Aufbau und die Relationen der Klassen sind im Anhang B.10 zu sehen.

6.2.4.2 Nachrichtenverarbeitung

Der nächste Schritt stellt die Verarbeitung durch den jeweiligen *MeasurementHandler* dar. Zuerst wurden zwei Klassen implementiert. Anschließend wurde eine Generalisierung durchgeführt, um gleichartige Abläufe zusammenzuführen. Es wurde eine Oberklasse entwickelt, die die gemeinsame Logik der beiden Handler-Klassen kapselt. Jede Abstraktion implementiert nur ihre eigene abgrenzende Funktionalität.

Der *RawMeasurementHandler* extrahiert aus dem Routing Key einer ankommenden Nachricht den Sensornamen. Mithilfe des Service-Layers wird ein vorhandener Sensor abgefragt oder ein neuer angelegt. Eine Nachricht liegt stets als String vor. Die *MeasurementFactory* wird eingesetzt, um einen konkreten Messwert zu erzeugen. Mit der Erzeugung des Messwerts endet die Nachrichtenverarbeitung.

Für die aggregierenden Microservices übernimmt der *GroupMeasurementHandler* die Verarbeitung. Es werden durch Service-Operationen die Gruppe und der Producer anhand deren IDs selektiert. Die IDs werden vorangehend aus den Routing-Informationen entnommen. Im ersten Konzept wurde aus dem *StorageWrapper* eine temporäre Datei ausgelesen. Diese konnte durch den Gruppennamen eindeutig identifiziert werden. Für den Fall, dass keine temporäre Datei vorhanden war, wurde ein neues Objekt vom Typ *TempGroupMeasurement* erzeugt. Danach wurde in diesem Objekt ein Eintrag für den aktuellen Messwert eingefügt. Im nächsten Schritt wurde geprüft, ob alle Werte vorhanden sind. War dies der Fall, so wurde für jeden Aggregator ein neuer Messwert gebildet. Hinterher wurde die temporäre Datei gelöscht und der Vorgang konnte von vorne beginnen. Waren noch nicht alle Messwerte vorhanden, so wurde die temporäre Datei, welche der neue Messwert hinzugefügt wurde, erneut persistiert.

Für Gruppen mit vielen Mitgliedern fiel beim Testen jedoch auf, dass das häufige Beschreiben der gleichen Datei Probleme verursacht. Die Schreiboperationen wirkten auf die Verarbeitungsgeschwindigkeit aus. Darüber hinaus traten Fehler auf, welche dazu führten, dass nachfolgende Dateizugriffe nicht ausgeführt werden konnten. Dateien wurden durch schnell aufeinanderfolgende Lese- und Schreiboperationen gesperrt. Als Folge blockiert das System weitere Anfragen. Die Bildung von Aggregaten kann nicht mehr durchgeführt werden. Zur Lösung des Problems wurde zuerst angedacht, die Dateinamen dynamisch zu wählen. Dadurch wäre jedoch eine Skalierung nicht mehr realisierbar. Jeder Service würde eigene temporäre Dateien zur Kumulation von Messwerten anlegen. Die Aggregatbildung über mehrere Replikate wäre nicht mehr möglich. Stattdessen wurde entschieden, ganz auf temporäre Dateien zur Aggregatbildung zu verzichten. Die Objekte vom Typ *TempGroupMeasurementStore* werden stattdessen im Cache zwischengespeichert. Sie werden im JSON-Format als Schlüssel-Wert-Paare abgelegt. Zur Identifikation der Einträge kann die jeweilige Gruppe genutzt werden. Hierbei können die Objekte über diesen Schlüssel abgerufen, eingefügt und gelöscht werden. Statt dem Object Storage wird nun der Cache für die Aggregatbildung verwendet. Der Aufbau der Modelle sowie die grundlegende Logik blieb dabei gleich. Nur das Medium zur Persistenz musste geändert werden. Somit konnte die Anpassung mit geringem Aufwand durchgeführt werden. Als Folge war ebenso eine Verbesserung der Verarbeitungsgeschwindigkeit festzustellen.

Die letzten Messwerte werden unter den Sensornamen beziehungsweise den Gruppennamen, gefolgt vom Namen des Aggregats abgelegt. Es könnte möglich sein, dass es hier

zu Überschneidungen kommt. Zum Beispiel wäre ein korrektes Systemverhalten nicht mehr gewährleistet, wenn eine Gruppe mit dem identischen Namen wie ein Sensor angelegt wird. Um sicherzustellen, dass keine Probleme für das Caching der letzten Messwerte entsteht, musste eine Unterscheidung der Keys durch Präfixe stattfinden. Schlüssel, die auf letzte Messwerte verweisen, werden mit dem Präfix *M* versehen. Die TempGroup-MeasurementStores werden mit *T* gekennzeichnet. Durch die Logik zum Speichern und Abrufen aus dem Cache erfolgt diese Kennzeichnung automatisch. Damit bleibt sie den anderen Schichten verborgen. Der Aufbau der Handler-Klassen ist im Anhang B.11 zu finden.

6.2.4.3 Speichern von Messwerten

Die Messwerte werden zuerst im Object Storage persistiert. Zum Speichern wird eine Wrapper-Klasse, der *StorageWrapper* aufgerufen. Diese beinhaltet wiederum Wrapper für die Clients der *MinIO*- und *Memcached*-SDKs. Hier werden alle benötigten Funktionalitäten der SDKs implementiert. Die *MinioClientWrapper*-Klasse beinhaltet Logik zum Ablegen und Abrufen von einzelnen Objekten über deren Namen. Zudem kann das Löschen von Objekten durchgeführt werden. Weitere Funktionen sind das Selektieren von mehreren Objekten über deren Präfix für die historische Aggregation und das Generieren von *Pre-signed URLs* zur Weiterverarbeitung der Messwerte. In Listing 6.3 kann die generische Implementierung einer Methode zum Auslesen von Objekten nachvollzogen werden. Diese kann dann im *StorageWrapper* zur Abfrage aller serialisierbarer Objekte verwendet werden.

```

1 public <T> T getObject(String name, Class<T> target) {
2     T object = null;
3     try {
4         InputStream is = minioClient.getObject(
5             GetObjectArgs.builder()
6                 .bucket(applicationProperties.getMicroserviceBucket())
7                 .object(name)
8                 .build()
9         );
10        ObjectMapper objectMapper = new ObjectMapper();
11        object = objectMapper.readValue(is.readAllBytes(), target);
12    } catch (ErrorResponseException ere) {
13        log.info("No Object with name found in object store.");
14        return null;
15    } catch (Exception e) {
16        log.error("GetObject from object store failed.");
17        return null;
18    }
19    log.info("GetObject from object store: " + object);
20    return object;
21 }

```

Listing 6.3: Objekt aus Object Storage abrufen

Für die Rohdaten werden Messwerte für jeden Sensor in einer dem Datumsformat entsprechenden Verzeichnisstruktur abgelegt. Bei der Aggregatbildung erfolgt das Speichern anhand der Gruppen und Kombinatoren. Jeder Messwert wird in der entsprechenden Gruppe unter dem Namen der jeweiligen Kombinatorfunktion persistiert. Die Unterordner sind analog zu den rohdatenverarbeitenden Microservices entsprechend des Datums strukturiert. Eine Navigation des Object Storages wird hierdurch erleichtert. Durch eine Gruppierung nach Sensoren beziehungsweise Gruppen kann ein Suchaufwand minimiert werden. Des Weiteren wird die Umsetzung einer historischen Aggregation vereinfacht.

Nach dem Speichern im Object Storage wird, falls der Producer im Monitoring visualisiert werden soll, ein Schlüssel-Wert-Paar im Cache erzeugt. Dies übernimmt die *MemcachedClientWrapper-Klasse*. Diese stellt Funktionalität zum Auslesen und Speichern einzelner und mehrerer Werte bereit. Zudem wird Funktionalität zum Löschen von Einträgen angeboten, welche zum Entfernen der TempGroupMeasurementStores nach Aggregatbildung verwendet wird. Zum tieferen Verständnis kann das Klassendiagramm im Anhang B.12 hinzugezogen werden.

6.2.4.4 Erneutes Publizieren zur Weiterverarbeitung

Im letzten Verarbeitungsschritt wird geprüft, ob ein generierter Messwert auch weiterverarbeitet werden muss. Dazu werden alle aktiven Gruppen, in denen er sich befindet, selektiert. Wurde hierbei keine einzige Gruppe ausgewählt, so wird der nachfolgende Schritt übersprungen. Andernfalls wird die *Presigned URL* generiert. Dazu wird der zurückgegebene Name des Objekts im Object Storage verwendet. Die URL ermöglicht den direkten Zugriff auf den Messwert. Anschließend wird für jede Gruppe ein *Publish* über AMQP ausgeführt. Diesen Vorgang übernimmt die *Publisher-Klasse*. Der Publisher ist genau wie der Receiver eine Erweiterung der *BrokerConnection-Klasse*. Ein Publish erfolgt auf dem Exchange *group_exchange*. Ein Auszug aus der konkreten Implementierung ist in Listing 6.4 zu sehen. Hier ist die Publikation durch einen rohdatenverarbeitenden Microservice dargestellt. Durch den Aufbau des Routing Keys liegen dem weiterverarbeitenden Service alle Informationen vor, die zur Aggregatbildung benötigt werden. Hierbei setzt sich der Routing Key aus Gruppentyp, Gruppen-ID und Producer-ID zusammen. Nach Bedarf kann das Konzept des Routings nochmals in Kapitel 5.1.2.1 nachgeschlagen werden. Der Microservice zur Aggregatbildung hat beim Programmstart bereits ein Binding von dem Exchange in seine jeweilige Queue erzeugt. Dazu hat er den Namen des festgelegten Gruppentyps verwendet. Dieses Binding gewährleistet, dass alle Nachrichten korrekt weitergeleitet werden. Der Microservice kann die Nachrichten dann aus seiner dedizierten Queue entnehmen. Er erhält aus dem Nachrichteninhalt die URL, welche den Zugriff auf den Messwert im Object Storage erlaubt. Aus den Routing-Informationen kann er die ID des Producers, der den Messwert bestimmt hat, sowie ID der Gruppe, deren Aggregat gebildet wird, extrahieren. Diese Daten ermöglichen dann den Zugriff auf die Metadaten, welche bei der Berechnung berücksichtigt werden können.

```
1 List<Group> activeGroups = aggregator.getGroups()
2   .stream()
3   .filter(g -> g.getActive())
4   .collect(Collectors.toList());
5 if (activeGroups.size() > 0) {
6     final String url = storageWrapper.getPresignedObjectUrl(objName);
7     activeGroups.forEach(
8         g -> publisher.publish(
9             String.format("%s.%s.%s",
10                g.getGroupType().getName(),
11                g.getId(),
12                aggregator.getId()
13            ),
14            url
15        )
16    );
17 }
```

Listing 6.4: Erzeugung Presigned URL und Publish

6.3 Microservices zur Rohdatenverarbeitung

Im Folgenden wird beschrieben, wie die Code-Basis zur Implementierung von rohdatenverarbeitenden Microservices verwendet werden kann. Hier wird herausgestellt, welche Klassen implementiert werden müssen. Ebenfalls wird die Bereitstellung der Microservices erläutert.

6.3.1 Nutzung der Code-Basis

Die Microservices zur Rohdatenverarbeitung müssen die abstrakten Klassen der Code-Basis erweitern. *Spring Beans* werden zur Implementierung der Klassen eingesetzt. Eine Definition als Spring Bean führt zur automatischen Initialisierung und Verwaltung von Abhängigkeiten. Dadurch werden die Relationen zwischen den Objekten durch den *IoC-Container* verwaltet statt durch die Objekte selbst. Dies bietet den Vorteil, dass die Objekt-Lebenszyklen nicht selbst verwaltet werden müssen und Abhängigkeiten sehr leichtgewichtig abgebildet werden können. Das Scope von Spring Beans ist standardmäßig *Singleton*. Dadurch wird sichergestellt, dass von jeder Klasse nur eine Instanz existiert. Der Zugriff auf die Singletons soll dem Design-Pattern entsprechend nur an einer zentralen Stelle erfolgen [31]. Dies wird durch das Spring Framework gewährleistet. Für einen Microservice müssen die folgenden Klassen implementiert werden:

RawMicroserviceApplicationProperties: Definiert die Konfiguration der rohdatenverarbeitenden Microservices. Hier müssen die Schlüssel zugewiesen werden.

RawSetupDataLoader: Legt notwendige Einträge in der Datenbank bei Applikationsstart an. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

BaseReceiver: Realisiert den Datenempfang vom Message Broker. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

ActivityMicroserviceController: Stellt die Aktivitätsmetrik über REST bereit. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

RawMeasurementHandler: Bildet die Datenverarbeitung ab. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

MeasurementFactory: Erzeugt den Messwert. Hier muss die Bildung der Messwerte definiert werden.

StorageWrapper: Realisiert die Interaktion mit der Datenhaltung. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

Publisher: Ermöglicht die Weiterverarbeitung. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

6.3.2 Kubernetes-Deployment

Zum Deployment muss der Microservice mit *Maven* als *JAR-Datei* gepackt werden. Anschließend wird die Anwendung containerisiert. Dazu wird ein *Dockerfile* verwendet. Es wird ein Image erzeugt, welches beim Start des Containers die JAR ausführt. Das erzeugte Container-Image wird auf *Docker Hub* bereitgestellt. Dadurch kann es über einen eindeutigen Tag von überall abgerufen werden. Das Kubernetes-Deployment nutzt das Image zum Aufsetzen des Containers innerhalb der Pods. Es wird sich an das „one-container-per-pod“-Modell gehalten. Hierbei wird die Ressourcenverteilung an die Anwendungen

erleichtert. Ebenso ist die Durchführung einer Skalierung besser umsetzbar. Für eine Microservices-Architektur bietet sich das Modell an, da jeder Service in einem Pod ausgeführt wird. Für jeden Pod wird die Konfiguration entsprechend der ConfigMap und Secrets über Umgebungsvariablen gesetzt. Der Container ist ausschließlich von innerhalb des Clusters zu erreichen, da er lediglich von Prometheus abgefragt werden muss. Dadurch muss er nicht zusätzlich gesichert werden. Die Kubernetes-Ressourcen *ServiceMonitor* und *HorizontalPodAutoscaler* ermöglichen die Skalierung entsprechend der im Kapitel 5.2.3 dargestellten Logik. Die Ausgestaltung ist im Anhang B.1 ausgegliedert. Durch das Setzen des Labels "monitoring" auf "true" bei der Definition des Services, wird dieser zur Überwachung gekennzeichnet. Es wird nur ein einzelner ServiceMonitor definiert. Dieser überwacht alle Services, die das Label gesetzt haben. Im Anhang B.2 ist die Definition dieses ServiceMonitors zu sehen.

6.4 Microservices zur Aggregation

Es folgt die Beschreibung der weiterverarbeitenden Microservices. Hier wird auf die Verwendung der Code-Basis zur Implementierung eingegangen. Es wird abgegrenzt, wo ein Eigenaufwand benötigt wird. Anschließend werden die verschiedenen Typen von Aggregaten einzeln betrachtet. Zuletzt wird das Deployment im Kubernetes-Cluster beschrieben.

6.4.1 Nutzung der Code-Basis

Die Microservices zur Aggregation müssen ebenfalls die abstrakten Klassen der Code-Basis erweitern. Hier ist die Implementierung folgender Klassen notwendig:

GroupMicroserviceApplicationProperties: Definiert die Konfiguration der weiterverarbeitenden Microservices. Hier müssen die Schlüssel zugewiesen werden.

GroupSetupDataLoader: Legt notwendige Einträge in der Datenbank bei Applikationsstart an. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

GroupReceiver: Realisiert den Datenempfang vom Message Broker. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

ActivityMicroserviceController: Stellt die Aktivitätsmetrik über REST bereit. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

GroupMeasurementHandler: Bildet die Datenverarbeitung ab. Hier muss die Methode *addCombinators* implementiert werden. Dies ermöglicht das Hinzufügen von Klassen des Typs *CombinatorModel*, die zur Aggregatbildung verwendet werden können.

StorageWrapper: Realisiert die Interaktion mit der Datenhaltung. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

Publisher: Ermöglicht die Weiterverarbeitung. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

6.4.2 Aggregation gleicher Datentypen

Zur Aggregation gleicher Datentypen müssen lediglich Daten gleichen Typs verrechnet werden. Hier können vordefinierte *CombinatorModels* verwendet werden, welche bereits die Bestimmung von Durchschnittswerten, Minima und Maxima für gängige Datentypen ermöglichen. Andernfalls können hier auch eigene *CombinatorModels* definiert werden.

6.4.3 Aggregation verschiedener Datentypen

Die notwendigen *CombinatorModels* zur Aggregation verschiedener Datentypen müssen immer selbst definiert werden. An der Kalkulation von Tauwerten kann die Logik nachvollzogen werden. Ein Tauwert kann mit einer Formel bestimmt werden, die Temperatur- und Luftfeuchtwerte verrechnet. Die Methode *addCombinators* eines *Microservices* zur Berechnung von Tauwerten kann im Anhang B.3 betrachtet werden.

6.4.4 Aggregation komplexe Formeln

Es wurde auch ein *Microservice* implementiert, der eine komplexere Formel umsetzt. Dieser *Microservice* berechnet einen Wärmedurchgangskoeffizienten und einen Kondensationswert. Dazu sind drei Werte notwendig, und zwar eine Außen-, eine Innentemperatur und ein Taupunkt. Um eine Unterscheidung von Außen- und Innentemperaturwerten zu ermöglichen, werden die *Tags* verwendet. Ein Sensor muss dazu vorher als Außensensor markiert werden. Die Formel zur Berechnung von Wärmefluss und Kondensation bezieht die Isolierung mit ein. Isolierungswerte sind jedoch abhängig von den Räumlichkeiten. Es ist möglich, diese auf Ebene der Gruppe über die *FormulaItemValues* zu definieren. Diese werden dann bei der Bildung der Aggregate berücksichtigt. Die Implementierung der Berechnungen ist im Anhang B.4 zu finden.

6.4.5 Kubernetes-Deployment

Die Bereitstellung der *Microservices* unterscheidet sich im Aufbau nicht von dem im Anhang B.1 dargestellten Deployment. Im Wesentlichen müssen nur andere Einträge innerhalb der zentralen *ConfigMap* definiert werden.

6.5 Microservices zur historischen Aggregation

Die *Microservices* zur historischen Aggregation erweitern ebenfalls die Code-Basis. Es wird beschrieben, welche Klassen implementiert werden müssen. Danach erfolgt eine kurze Erklärung zur Bereitstellung über Kubernetes.

6.5.1 Nutzung der Code-Basis

Auch zur Implementierung der *Microservices* zur historischen Aggregation werden die Klassen der Code-Basis erweitert. Es müssen folgende Oberklassen abstrahiert werden:

HistoricAggregatorApplicationProperties: Die Konfiguration der *Microservices* zur historischen Aggregation wird festgelegt. Hier müssen die Schlüssel zugewiesen werden.

HistoricAggregatorScheduler: Führt die Aggregation aus. Hier muss die Methode *addHistoricCombinators* implementiert werden. Dies ermöglicht das Hinzufügen von Klassen des Typs *HistoricCombinatorModel*, die zur Aggregatbildung verwendet werden können.

HistoricStorageWrapper: Realisiert die Interaktion mit der Datenhaltung. Für die Historisierung ist ein kleinerer Funktionsumfang ausreichend. Es ist kein zusätzlicher Implementierungsaufwand notwendig.

Die *HistoricCombinatorModel*-Klasse beinhaltet eine *HistoricCombinatorFunction*. Hierbei handelt es sich um ein funktionales Interface. Aus einer Liste von Messungen eines generischen Typs wird ein neuer Messwert erzeugt. Dieser hat den gleichen Datentyp. Hierdurch kann eine Aggregation von Messwerten über Zeiträume realisiert werden.

6.5.2 Kubernetes-Deployment

Das Deployment der Microservices zur historischen Aggregation benötigt keine Definition von Services, ServiceMonitors oder eines Horizontal Pod Autoscalers. Eine Skalierung ist nicht notwendig, da für eine historische Aggregation keine Zeitdringlichkeit vorliegt. Dementsprechend ist ein Deployment mit einem einzelnen Pod ausreichend.

6.6 Maven Archetypes zur Erstellung neuer Microservices

Die Implementierung neuer Microservices zur Rohdatenverarbeitung, zur Aggregation in Echtzeit sowie zur historischen Aggregatbildung soll möglichst schnell und bequem durchführbar sein. Aus diesem Grund wurden eigene *Maven Archetypes* entwickelt. Ein Archetype ist ein Template für ein Projekt, welches zur Erzeugung ähnlicher Module genutzt werden kann [55]. Um einen eigenen Archetype zu erstellen, muss ein neues Maven Projekt angelegt werden. Dieses deklariert in seiner POM, dass keine ausführbare Datei erzeugt wird, sondern ein Archetype. Durch das Hinzufügen von Erweiterungen des Build-Managements kann das Projekt im lokalen Maven Repository installiert werden [54]. Außerdem muss eine Datei namens *archetype-metadata.xml* erzeugt werden. Diese definiert alle Verzeichnisse und Dateien, die bei der Erzeugung des Archetypes anzulegen sind. Hier werden dann die Ordner referenziert, in denen sich der Quellcode befindet. Ebenso werden Konfigurationsdateien, zum Beispiel die *application.properties* Datei, hinzugefügt. Darüber hinaus können auch Dockerfiles oder YAML-Dateien gelistet werden, welche für jedes Projekt des Archetypes präsent sein sollen. Die aufgeführten Verzeichnisse und Dateien müssen dann in einem Ordner mit dem Namen *archetype-resources* vorhanden sein. Dieser bildet den Aufbau des Prototyps ab, der bei der Generierung des Archetyps verwendet wird. Hierbei werden alle Klassen, bei denen kein zusätzlicher Implementierungsaufwand notwendig ist, vollständig implementiert. Gleichmaßen werden Konfigurationen standardmäßig vorgenommen. Es werden entsprechende Kommentare eingefügt, die auf Stellen verweisen, wo zusätzliche Arbeit benötigt wird. Ebenfalls können Platzhalter genutzt werden. Diese werden bei der Generierung durch Parameter ersetzt. Die POM des Prototyps muss ebenfalls beschrieben werden. Auch hier können Platzhalter für parametrisierbare Informationen wie den Artefaktnamen genutzt werden. Um den Archetype verwenden zu können, muss er im kompiliert und installiert werden.

Durch die Verwendung von Archetypes kann der Grundaufbau eines Microservices automatisch durch den Aufruf eines einzelnen Befehls generiert werden. Als Beispiel kann das in Listing 6.5 dargestellte Kommando benutzt werden, um einen neuen rohdatenverarbeitenden Microservice zu erzeugen. Hier können zusätzliche Daten, wie der Name des Artefakts, des Pakets oder die Version, angegeben werden. Diese werden dann bei der Erstellung der POM beziehungsweise der Klassen eingesetzt. Nach dem Absetzen des Befehls liegt dem Entwickler ein Projekt vor, welches alle benötigten Klassen, Konfigurationsdateien und Tests beinhaltet. Zudem wird das Projekt entsprechend der allgemeinen Paketstruktur erstellt. Abhängigkeiten sowie Konfigurationen sind bereits definiert. Es müssen nur an den markierten Stellen Ergänzungen erfolgen. Falls eine komplexere Umsetzung angedacht ist, können weitere Anpassungen getätigt werden. Der Archetype

ermöglicht also einen Schnelleinstieg in die Entwicklung, indem er die wiederholende Tätigkeit des Aufbaus der Projektstruktur automatisiert.

```

1 mvn archetype:generate
2   -DarchetypeGroupId=de.htw.saar.smartcity
3   -DarchetypeArtifactId=raw-ms-archetype
4   -DarchetypeVersion=0.0.1-SNAPSHOT
5   -DgroupId=de.htw.saar.smartcity
6   -DartifactId=new-raw-ms
7   -Dversion=0.0.1-SNAPSHOT
8   -Dpackage=de.htw.saar.smartcity.newpackage

```

Listing 6.5: Befehl zur Erzeugung eines rohdatenverarbeitenden Microservices

6.7 Microservice zur Bereitstellung an Prometheus

Das *Prometheus SDK* ermöglicht die Erstellung eines Clients. Zur direkten Instrumentalisierung sind vier Arten von Metriken verfügbar, namentlich *Counter*, *Gauges*, *Summaries* und *Histograms*. Diese werden als Klassen angeboten, welche Logik zur Aktualisierung bei Wertänderung beinhalten. Die Bereitstellung an Prometheus wird automatisch durch den Client durchgeführt. In Listing 6.6 ist die Erstellung einer Gauge dargestellt.

```

1 class YourClass {
2     static final Gauge inprogressRequests = Gauge.build()
3         .name("inprogress_requests").help("Inprogress requests.").
4         register();
5
6     void processRequest() {
7         inprogressRequests.inc();
8         // Your code here.
9         inprogressRequests.dec();
10    }
11 }

```

Listing 6.6: Beispiel direkte Instrumentalisierung mit Gauge [124]

Die Daten, die an Prometheus exportiert werden sollen, liegen im Cache und müssen von dort ausgelesen werden. Der Einsatz der direkten Instrumentalisierung würde die periodische Abfrage des Caches erfordern. So würde die in Kapitel 5.2.5 beschriebene Inkonsistenz ein großes Ausmaß annehmen. Ein weiteres Problem liegt in der Tatsache, dass für jeden Producer ein Objekt erstellt werden müsste. Diese Objekte stellen eine einzelne Metrik dar und müssten in einer Datenstruktur gehalten werden. Es können neue Producer hinzukommen oder wegfallen. In beiden Fällen müsste dies dem Microservice mitgeteilt werden. Daraufgehend müsste dieser die Datenstruktur aktualisieren. Dieser Ablauf ist umständlich und fehleranfällig. Eine direkte Instrumentalisierung bietet sich für Metriken an, die direkt aus einer Anwendung entnommen werden können. Der Microservice zur Bereitstellung an Prometheus muss diese jedoch dynamisch aufbereiten.

Eine Alternative stellt die Implementierung eines *CustomCollectors* dar. Dieser muss bei der *CollectorRegistry* registriert werden. Die Bereitstellung der Metriken erfolgt in der *collect-Methode*. Durch die Implementierung eines *CustomCollectors* kann diese definiert werden. Die Methode wird bei einem Prometheus-Pull aufgerufen. Sie legt alle zu exportierenden Metriken fest. Durch die Implementierung der *collect-Methode* wird eine gezielte Erzeugung von Metriken möglich. Die Rückgabe erfolgt dann in Form einer Liste vom Typ *MetricFamilySamples*.

Die Implementierung eines *CustomCollectors* bietet sich für die Bereitstellung an. Es kann eine flexiblere Zusammensetzung der Metriken erfolgen. Außerdem stellt die Ent-

wicklung eines CustomCollectors eine schlüssigere Lösung dar. Eine direkte Instrumentalisierung kann nur über Umwege realisiert werden. Zudem können die entstehenden Inkonsistenzen minimiert werden.

Zuerst werden alle zum Export markierten Producer aus der Datenbank ausgelesen. Anschließend werden für diese die letzten Messwerte vom Memcached-Server abgefragt. Mit den Messwerten werden dann Metriken gebildet. Diese werden mit Metadaten angereichert. Dazu werden Labels gesetzt, die Auskunft über den Datentyp und deren Herkunft geben. Der Aufbau der Klassen für die Bereitstellung an Prometheus ist im Anhang B.13 zu sehen. Im Anhang B.14 ist ein Auszug aus den generierten Metriken der Anwendung zu finden.

6.7.1 Weiterführung der Implementierung

Es können je nach Konfiguration nur bestimmte Datentypen oder nur ausgewählte ID-Ranges exportiert werden. Dies ermöglicht die Durchführung einer manuellen Skalierung durch Partitionierung der Producer. Mehrere Microservices können bereitgestellt werden, die jeweils nur einen Teil der Daten exportieren. Aufbauend auf der aktuellen Implementierung kann wie in Kapitel 5.2.5 beschrieben eine *Primary-Secondary-Architektur* implementiert werden. Durch diese Erweiterung würde eine Skalierung automatisch erfolgen.

6.7.2 Kubernetes Deployment

Um alle vom Microservice bereitgestellten Metriken abfragen zu können, muss folgende Konfiguration gesetzt sein:

```
management.endpoints.web.exposure.include = prometheus, metrics
```

Dadurch stellt die *CollectorRegistry* die Endpunkte über REST bereit. Das vollständige Deployment des Microservices kann im Anhang B.5 betrachtet werden. Auch hier greift der zentrale ServiceMonitor auf die jeweiligen Endpunkte zu und überwacht diese.

6.8 Microservice Monitoring

Zum Monitoring der Microservices wird die eigene Aktivitätsmetrik verwendet. Zusätzlich werden Informationen zu den jeweiligen Deployments von Kubernetes bereitgestellt. Weitere Metriken werden durch *Spring Boot Actuator* verfügbar [91]. Dazu muss das Modul über Maven hinzugefügt werden. Dies geschieht durch Einfügen der jeweiligen Abhängigkeit in der POM. Anschließend steht eine Vielzahl von Metriken zur Abfrage bereit. Die Daten kommen beispielsweise von der JVM, dem ausführenden System, der Logging-Schnittstelle, der DataSource und dem HTTP-Client. Dadurch lassen sich Metriken bestimmen, wie zum Beispiel die CPU-Zeit und der Speicherverbrauch, die Anwendungslaufzeit, die Anzahl kürzlich aufgetretener Fehler und die Summe an eingehenden REST-Calls.

6.9 Hilfsmodule

Im Folgenden wird auf die zwei Hilfsmodule eingegangen. Diese sind nicht für die Abbildung der Kernfunktionalität benötigt. Das Simulationsmodul dient dem Testen des Systems. Es wird zur Erzeugung von Sensormesswerten eingesetzt. Das andere Modul ermöglicht eine Verwaltung von Stamm- und Metadaten. Hier können Sensoren, Aggregatoren, Gruppen und ihre Metadaten angelegt werden.

6.9.1 Simulationsmodul

Das Simulationsmodul ermöglicht den Publish von Sensormesswerten über MQTT. Dazu werden zufällige Werte eines Datentyps generiert und in separaten Threads publiziert. Je nach Datentyp werden die Messwerte auf einem entsprechenden Topic publiziert. Zur Abbildung eines virtuellen Sensors wird die *Agent-Klasse* verwendet. Diese abstrakte Klasse beinhaltet Logik, die in festlegbaren Zeitintervallen eine Veröffentlichung entsprechend dem ebenfalls konfigurierbaren Sensornamen ausführt. Konkrete Implementierungen, wie beispielsweise ein *TemperatureAgent*, können dann gestartet werden. Sie erheben analog zu einem physischen Sensor Messwerte.

Zufällig generierte Daten ermöglichen das Testen der Funktionalität. Zum Finden von sinnvollen Visualisierungsmöglichkeiten im Grafana Dashboard sind solche Daten jedoch nicht geeignet. Aus diesem Grund wurde das Simulationsmodul erweitert. Das Ziel der Erweiterung war das Erzeugen realitätsnaher Messwerte. Statt realistische Werte zu erzeugen, können auch reale Daten verwendet werden. Deshalb wurden diverse APIs gesichtet, die Sensordaten bereitstellen. Die Entscheidung fiel dann auf die API von *OpenWeather* [110]. Diese kann kostenfrei bis zu sechzigmal die Minute abgefragt werden. Bei einem standardmäßigen Publikationsintervall von dreißig Sekunden können somit bis zu dreißig Sensoren angebunden werden. Diese Anzahl ist zum Testen mehr als ausreichend. Mithilfe der ausführlichen Dokumentation [18] wurde ein Client zur Abfrage der API über REST erstellt. Hierbei ist die Abfrage der aktuellen Wetterlage nach Stadt möglich. Die Städte werden durch eindeutige Nummern repräsentiert, welche aus einer Liste entnommen werden können. Aus den Daten werden dann Informationen wie Temperatur, Luftfeuchtigkeit und die Witterung extrahiert. Anschließend werden diese auf virtuelle Sensoren projiziert. Zuletzt erfolgt die Publikation auf den jeweiligen Topics. Die bei der API abgefragten Städte können über die Konfiguration festgelegt werden. Somit können diese je nach Bedarf geändert werden.

Das Simulationsmodul konnte dann beispielsweise eingesetzt werden, um fehlende Hardware zu ersetzen. Hier wurde es zum Senden von Bilddaten verwendet. Gleichmaßen wurde es eingesetzt, um das gleichzeitige Senden vieler Sensoren zu simulieren. Dies war hilfreich beim Testen der Systemfunktionalität und bei der Ausführung eines Lasttests. Auch bei der Erstellung der Dashboards war es nützlich, da die realen Wetterdaten die Gestaltung der Graphen und Tabellen erleichterten.

6.9.2 Modul zur Verwaltung von Stamm- und Metadaten

Ein Modul zur Verwaltung der Stamm- und Metadaten wurde ebenfalls implementiert. Hierbei handelt es sich um eine *REST-API*. Bei der Erstellung der Schnittstelle wurde sich möglichst an den REST-Paradigmen orientiert. Hier wurde insbesondere auf die Zustandslosigkeit, das *Uniform Interface* und die Mehrschichtigkeit geachtet. Für alle benötigten Ressourcen wurden *Spring Web MVC Controller* erstellt. Das Spring-Framework ermöglicht hierbei ein bequemes Mapping von Pfaden und Endpunkten. Dazu werden Annotations gesetzt. Diese erzeugen automatisch eine Assoziation zwischen den *HTTP-Commands* und den Endpunkten. Ebenfalls ermöglichen die Annotations eine leichtgewichtige Serialisierung von *Pfadvariablen* und *HTTP-Request-Inhalten*. Auch *HTTP-Responses* lassen sich flexibel zusammenbauen. Dazu kann die *ResponseEntity-Klasse* verwendet werden. Diese ermöglicht das Setzen des Inhalts und Statuscodes der HTTP-Response. Dabei können dem Body je nach Operation verschiedene Inhalte zugeordnet werden. Beispielsweise bleibt er bei DELETE-Operationen leer. Bei GET-Operationen hingegen kann er ein einzelnes Element oder auch eine Liste von Objekten beinhalten. Mithilfe von *ControllerAdvice*s können auftretende Fehler direkt in passende ResponseEntities umgewandelt werden.

Hierbei können Status Codes flexibel zugeordnet und entsprechende Meldungen gesetzt werden. Dies ist insbesondere für eine eigene Fehlerbehandlung nützlich. Die Schnittstelle wurde mit dem OpenAPI 3.0 Standard implementiert. Dadurch steht ein Webinterface bereit, womit die Endpunkte bedient und getestet werden können. Eine andere Projektarbeit soll später ein Modul zur Registrierung und Verwaltung von Sensoren und Robotern realisieren. Aus diesem Grund war eine konzeptionelle Umsetzung ausreichend. Für dieses zukünftige Projekt können die bereitgestellten Endpunkte gegebenenfalls verwendet oder erweitert werden. Die Implementierung eines Clients für die Schnittstelle lässt sich mit *Swagger Codegen* realisieren. Hiermit kann automatisch ein Client-SDK für die API erzeugt werden.

6.10 Deployment

Die Bereitstellung der Anwendungen erfolgte während der Entwicklung in die lokale Testumgebung. Hier wurden alle Microservices installiert. Es wurde ebenfalls ein Integrationstest durchgeführt. Anschließend wurden die Anwendungen im Produktivsystem bereitgestellt. Dieses unterscheidet sich teilweise im Aufbau und durch die verwendete Software. Hier musste erneut die Funktionalität aller Microservices geprüft werden.

6.10.1 Minikube Testumgebung

Die Software, die von den Microservices benötigt wird, muss bereitgestellt werden. Hier wurde versucht, möglichst auf *Helm Charts* zurückzugreifen. Diese vereinfachen die Bereitstellung und Verwaltung von Kubernetes-Ressourcen. Für die Testumgebung wurden die Helm Charts für die Server von *MySQL*, *MinIO* und *Memcached* verwendet. Ebenso wurde die Installation des *Kubernetes Prometheus Stack* über Helm durchgeführt. Hier sind neben dem *Prometheus Operator* noch weitere zum Cluster Monitoring benötigte Komponenten beinhaltet. *Grafana* ist ein Bestandteil des Stacks und muss nicht gesondert installiert werden. Der *Prometheus Adapter* hingegen muss separat bereitgestellt werden. *RabbitMQ* wurde in der Testumgebung durch ein eigenes angepasstes Deployment aufgesetzt, wo spezielle Konfigurationen getätigt werden. Diese Anpassungen aktivieren die RabbitMQ Plugins zum Management und zum Monitoring durch Prometheus sowie zur Nutzung von MQTT. Außerdem wird das Exportieren von Metriken für jedes Kubernetes-Objekt konfiguriert. Ein *ServiceMonitor* für den RabbitMQ-Service ermöglicht das Auslesen von Metriken durch Prometheus. Damit kann das Monitoring des RabbitMQ-Servers sowie der Queues realisiert werden. Die zugehörige YAML-Datei zur Anlage der Kubernetes-Ressourcen kann im Anhang B.16 betrachtet werden.

Die Microservices wurden nach und nach in die lokale Testumgebung integriert. Zum Testen wurde auf dem lokalen System ein Kubernetes-Cluster mithilfe von *Minikube* aufgesetzt. Mit Minikube kann ein *Single Node Cluster* auf dem eigenen Rechner ausgeführt werden. Das Cluster kann innerhalb eines *Docker Containers* oder einer *Virtual Machine (VM)* ausgeführt werden. Die Nutzung einer VM bietet hierbei Vorteile für die Konfiguration. Es ist beispielsweise möglich, dieser eine feste IP-Adresse zuzordnen. Zur Erstellung der VM wurde *Hyper-V* verwendet. Hierbei handelt es sich um die Hypervisor-basierte Virtualisierungssoftware von Microsoft [116]. Mit Minikube kann ein Cluster betrieben werden, ohne dass viel Hardware benötigt wird. Ebenfalls ist die Installation und Konfiguration verhältnismäßig simpel und schnell durchführbar. Dementsprechend eignet es sich für die lokale Anwendungsentwicklung und als Testumgebung. Die Microservices können hier bereitgestellt werden. Hierbei werden den Services statische NodePorts zugeordnet. Die feste IP-Adresse der VM und die Konfiguration der NodePorts gewährleisten

eine gleichbleibende Konfiguration. Neustarts der Testumgebung wirken sich nicht auf die Funktionalität aus. Dies ist wichtig, da die VM zusammen mit dem Betriebssystem heruntergefahren wird. In der Testumgebung konnte die korrekte Funktionalität der Microservices verifiziert werden. Ein Deployment in der Produktivumgebung konnte im Anschluss erfolgen.

6.10.2 Produktivsystem

Durch das vollständige Deployment der Microservices in die Testumgebung konnte die Integration verifiziert werden. Im DSL-Cluster musste zuerst auch die Installation der benötigten Software erfolgen. Hier wurde gleichermaßen auf die Helm Charts zugegriffen. Diese wurden im eigenen Namespace bereitgestellt. Im Default-Namespace stand bereits ein RabbitMQ-Deployment zur Verfügung. Es wurden separate YAML-Dateien für die Bereitstellung geschrieben. Die Deployments der einzelnen Microservices mussten hier nicht angepasst werden. Es war jedoch notwendig eine separate ConfigMap zu erstellen. Gleichmaßen mussten gesonderte Secrets angelegt werden. Hier wurden dann die entsprechenden Einträge zu den Verbindungsinformationen der Anwendungen getätigt. Beispielsweise mussten hier die RabbitMQ-Zertifikate angegeben werden, welche zur Verbindung mit dem Server benötigt werden. Diese Konfiguration war für die Testumgebung nicht notwendig. Hier ist der Server ohnehin nur aus dem lokalen Netz erreichbar und muss dementsprechend nicht gesichert werden.

6.11 Grafana Dashboards

Die Dashboards wurden über Grafana erstellt. Sie wurden in der Testumgebung konfiguriert. Von dort wurden sie exportiert und schließlich in das Produktivsystem importiert. Grafana ermöglicht es einfache Anzeigen direkt über das Webinterface zu erstellen. Daten können direkt visualisiert oder nach ihren Labels selektiert werden. Es ist auch eine komplexe Abfrage der Metriken möglich. Hierfür wird *PromQL* verwendet. Dabei handelt es sich um eine Abfragesprache von Prometheus, welche von Grafana unterstützt wird. PromQL ermöglicht zudem eine Weiterverarbeitung der Daten. So können zum Beispiel Summen über Labels oder über Zeiträume gebildet werden. Beispielsweise werden mit folgendem Ausdruck alle publizierten Nachrichten der letzten sechzig Sekunden summiert:

```
sum(rate(rabbitmq_channel_messages_published_total[60s]))
```

Nach Definition von passenden Abfragen kann die Visualisierung weiter angepasst werden. Es stehen verschiedene Anzeigeformate bereit. Dazu zählen unter anderem Tabellen, Balkendiagramme, Counter und Histogramme. Grafana ermöglicht je nach Anzeige eine Interpretation der Daten. Metriken können je nach Wert farblich gekennzeichnet werden. Ein Mapping von Werten ist auch möglich, so kann zum Beispiel eine Boolean-Variable interpretiert werden. Für den Wassersensor ist eine Anzeige von nass oder trocken aussagekräftiger als null oder eins. Die Interpretation von Luftqualitätswerten kann so ebenfalls erfolgen. Im Anhang B.15 ist die finale Ausgestaltung der Dashboards zu sehen. Bei dem dargestellten *RabbitMQ* Dashboard handelt es sich um eine vordefinierte Anzeige, welche eingebunden wurde. Sonst sind hier Auszüge aus den Dashboards zum Monitoring der Microservices und zur Anzeige der Sensordaten dargestellt.

7 Evaluation

Im Folgenden werden die gestellten Anforderungen der Umsetzung gegenübergestellt. Es wird ein Fertigstellungsgrad bestimmt. Anschließend wird eine Auswertung der Rohdatenverarbeitung durchgeführt. Anhand eines Benchmarking-Microservices wird die Verarbeitungsgeschwindigkeit in Bezug auf Sensormessungen verschiedener Größe festgestellt. Es werden Schlussfolgerungen für die Microservices zur Aggregation in Echtzeit gezogen. Darauffolgend wird die Auswertung des Microservices zum Export an Prometheus beschrieben. Zum Schluss erfolgt ein Ausblick auf die Skalierung der Anwendungen, welche zur Performancesteigerung verwendet werden kann.

7.1 Anforderungserfüllung

Der Fertigstellungsgrad der Umsetzung wird basierend auf den Arbeitspaketen beurteilt. Von den definierten Arbeitspaketen im Projektstrukturplan wurden alle umgesetzt. Folgt man der *0/100-Methode* zur Ermittlung eines Fortschrittsgrads für das Gesamtprojekt, erreicht man einen kumulierten Fertigstellungsgrad von hundert Prozent [109]. Im Folgenden wird die Umsetzung der festgelegten Anforderungen beurteilt. Hierzu ist eine Bewertung der Implementierung auf Basis der funktionalen Anforderungen in Tabelle 7.1 und der nichtfunktionalen Anforderungen in Tabelle 7.2 zu sehen.

7.2 Auswertung Microservices

Zur Auswertung der Microservices wurden Tests erstellt, die ein Benchmarking durchführen. Als Framework für das Benchmarking wird *Java Microbenchmark Harness* verwendet. Die Tests simulieren das Laufen der Microservices. Sie messen die Zeit beginnend mit dem Eingang des Messwerts beim *MeasurementHandler* bis zur Beendigung der Verarbeitung. Ziel des Benchmarks ist es, die maximale Anzahl an ausgeführten Verarbeitungen in einem Zeitintervall zu bestimmen. Dadurch kann die benötigte Zeit pro Verarbeitung abgeleitet werden. Die Funktionalität jedes Microservices wurde mit einem solchen Test verifiziert.

Das Benchmarking mit dem JMH-Framework ermöglicht auch die Ausführung in mehreren Iterationen festgelegter Dauer. So können verschiedene Messungen der Geschwindigkeit durchgeführt werden. Dadurch fallen statistische Ausreißer, also extreme Messwerte, weniger ins Gewicht. Zudem können Warmup-Iterationen durchgeführt werden. Diese werden vor dem eigentlichen Benchmarking ausgeführt und gehen nicht in die Auswertung mit ein. Die Ausführung von Warmup-Iterationen ermöglicht der Anwendung einen realitätsnahen Zustand zu erreichen. In diesem sind alle Ressourcen bereits allokiert. Initialisierungen wurden durchgeführt und Caches gefüllt. Bei vielen Implementierungen dauern Erstdurchläufe länger. Durch ein vorangehendes Aufwärmen wird erst bei einer Normalisierung der Ausführungszeit mit der Messung begonnen.

Tabelle 7.1: Erfüllung funktionaler Anforderungen

Anforderung	Umsetzung
Must-Have	
Flexible Anbindung neuer Sensortypen	Klassische Sensoren, die einen Messwert bestimmen, können mit minimalem Aufwand angebunden werden. Hierbei ist lediglich eine Sensoranwendung sowie ein Microservice zur Rohdatenverarbeitung zu erstellen. Durch die implementierte Code-Basis und die Maven Archetypes kann dies mit geringem Entwicklungsaufwand erfolgen.
Stamm- und Metadaten verwalten	Die Verwaltung wurde konzeptionell umgesetzt. Die Weiterentwicklung erfolgt in einer zukünftigen Arbeit.
Simple Echtzeitdatenverarbeitung	Microservices, die simple Aggregate bestimmen, können in kurzer Zeit entwickelt werden.
Historische Daten verarbeiten	Die Verarbeitung historischer Daten ist auch durch die Implementierung von Microservices realisierbar. Auch hier muss die bestehende Code-Basis erweitert werden. Ausstehend ist die Umsetzung eines Konzepts zur Anzeige der historischen Daten. Momentan umfassen die Dashboards nur aktuelle Daten.
Sensoranbindung	Die zur Verfügung stehenden Sensoren wurden angebunden. Anwendungen, die zu diesen sinnvolle Aggregate bilden, wurden implementiert.
Datenbereitstellung	Die Daten stehen in Grafana Dashboards bereit. Es wurden Dashboards erstellt, die die Messwerte der angebundenen Sensoren abbilden. Diese können zukünftig als Referenz dienen.
Should-Have	
Fortgeschrittene Echtzeitdatenverarbeitung	Eine komplexere Aggregatbildung ist ebenso möglich. Exemplarisch wurden hier ein Taupunkt-Microservice implementiert.
Nice-To-Have	
Komplexe Aggregatbildung	Auch zur Berechnung komplexer Aggregate basierend auf Formeln, wurde ein Proof-Of-Concept erstellt. Eine exemplarische Umsetzung wurde durch den Wärmefluss- und Kondensations-Microservice durchgeführt.
Monitoring	Ein simples Monitoring wurde realisiert. In den Grafana Dashboards können die aktiven Microservices überwacht werden. Hier stehen diverse Informationen, wie beispielsweise deren Aktivität, Ressourcennutzung und Laufzeit bereit. Als Erweiterung kann hier die Nutzung des Prometheus Alertmanager angedacht werden. Dieser ermöglicht die gezielte Benachrichtigung, wenn Metriken definierte Grenzen über- oder unterschreiten.

Tabelle 7.2: Erfüllung nichtfunktionaler Anforderungen

Anforderung	Umsetzung
Must-Have	
Flexibel neue Microservices anbinden	Der Implementierungsaufwand bei der Anbindung neuer Microservices zur Rohdatenverarbeitung oder Aggregation ist gering.
Bereitstellung mit Kubernetes	Alle Anwendungen wurden im Kubernetes-Cluster des Distributed Systems Lab bereitgestellt.
Zustandslosigkeit	Alle Microservices wurden zustandslos implementiert, wodurch eine Skalierung ermöglicht wird.
Ausgelagerte Konfiguration	Die Konfiguration kann zentral über eine Kubernetes Config-Map erfolgen.
Should-Have	
Automatische Skalierung	Die rohdatenverarbeitenden und aggregierenden Microservices skalieren automatisch nach der Frequenz eingehender Sensormessungen. Der Microservice zum Export kann händisch skaliert werden. Auch hier kann eine automatische Skalierung realisiert werden. Dazu wurde ein Konzept erstellt. Die Umsetzung hat noch zu erfolgen.

7.2.1 Rohdatenverarbeitung

Zur Evaluation der Microservices ist die Verarbeitungsgeschwindigkeit eine aussagekräftige Metrik. Zusätzlich zu dem Benchmarking der einzelnen Microservices wurde ein spezieller Benchmarking-Microservice erstellt. Dieser kann losgelöst von den regulären Microservices bereitgestellt werden. Er nutzt eigene Topics und persistiert die Daten in einem separaten Bucket. Gleichmaßen werden die erhobenen Messwerte nicht beim Export zu Prometheus berücksichtigt. Es werden zwei Warmup-Iterationen, gefolgt von zehn Mess-Iterationen durchgeführt. Eine einzelne Iteration läuft hierbei über einen Zeitraum von zehn Sekunden. Dadurch wird die Methode zur Datenverarbeitung, welche nur eine Dauer von einigen Millisekunden hat, sehr oft aufgerufen. Dies trägt dazu bei, dass die Genauigkeit des berechneten Mittelwerts steigt. Der Benchmarking-Microservice implementiert Tests, welche Messwerte verschiedener Größe verarbeiten. Die Nachrichtengröße wird von 1KB bis 1MB gestaffelt. Hierdurch kann analysiert werden, wie sich eine steigende Größe auf die Verarbeitungsgeschwindigkeit auswirkt. Durch das JMH-Framework kann eine Methode, flexibel mit verschiedenen Parametern gemessen werden. Hierzu müssen lediglich passende Annotations gesetzt werden. In Listing 7.1 ist ein Auszug aus der Benchmark-Klasse dargestellt. Hier kann der Aufbau des Tests nachvollzogen werden.

```

1 private int i = 0;
2 @Param({"1", "10", "20", "30", "40", "50", "60", "70", "80", "90", "100", "200",
        "300", "400", "500", "750", "1000"})
3 public int sizeInKB;
4
5 @Benchmark
6 public void executeBenchmark() {
7     benchmarkWithMessage(generateMessageOfSize(sizeInKB * 1024));
8 }

```

```

9 public void benchmarkWithMessage(String message) {
10     // check if benchmarkingRawMeasurementHandler is present
11     assert(benchmarkingRawMeasurementHandler != null);
12     SensorMeasurement sensorMeasurement = new SensorMeasurement();
13     sensorMeasurement.setMeasurement(message);
14     sensorMeasurement.setSensorName("data/aggregator/benchmark/sensor" +
        ((i++) % 100 + 1));
15     benchmarkingRawMeasurementHandler.handleMessage(sensorMeasurement);
16 }

```

Listing 7.1: Auszug Testklasse des Benchmarking-Microservice

Die meiste Aussagekraft hat das Benchmarking, wenn es in der Produktivumgebung durchgeführt wird. Deshalb sollen die Tests im DSL-Cluster ablaufen. Dazu werden sie innerhalb eines Containers in einem Pod ausgeführt. Hierbei ist auch die feste Zuteilung von Systemressourcen wichtig. Zur Durchführung wurde die Zuteilung von 512Mi Speicher und einer CPU-Zeit von 1000m festgelegt. Dadurch wird gewährleistet, dass stets die gleichen Ressourcen bereitstehen und keine Beeinträchtigung durch Engpässe entstehen können. Erfolgte Messungen der Ausführungszeit werden von der Anwendung in eine Datei geschrieben. Diese kann nach Beendigung des Pods über ein *Persistent Volume* weiterhin ausgelesen werden. Auf dem Cluster werden PVs dynamisch durch *glusterfs* provisioniert. Dementsprechend muss nur ein *Persistent Volume Claim* erzeugt werden, der einen Speicher der benötigten Größe anfordert. Die vollständige YAML für den Benchmarking-Microservice ist im Anhang C.1 ausgelagert. Im Anhang C.2 ist ein Auszug aus den unverarbeiteten Ergebnissen zu sehen.

Die Abbildung 7.1 dient der visuellen Darstellung der gemessenen Verarbeitungszeiten. Hier wird ein Durchschnittswert über die ausgeführten Iterationen gebildet. Schwarz markiert sind die Abweichungen der Messwerte vom dargestellten Mittel. Es ergibt sich für die Verarbeitung von Messwerten der Größe 1KB ein zeitlicher Aufwand von 38 Millisekunden. Man kann hierbei von einer Grundlast sprechen. Je größer die Nachrichten werden, desto länger dauert auch die Verarbeitung. Eine Steigerung von etwa 30KB führt etwa zu einer Verlängerung der Verarbeitungszeit um eine Millisekunde. Des Weiteren ist zu sehen, dass die Fehlerspannen relativ gering ausfallen. Die meisten Messwerte liegen dicht beieinander. Es liegen keine extremen Ausreißer vor.

Wie im Diagramm 7.2 zu erkennen ist, steigt die Verarbeitungszeit linear mit der Größe der erhobenen Messwerte. Dementsprechend können bei der Verarbeitung von Sensormessungen fester Größe ebenfalls konstante Zeiten bestimmt werden.

7.2.2 Aggregation

Die Microservices, welche eine Aggregatbildung durchführen, lassen sich nicht sinnvoll evaluieren. Je nach Aggregat kann die Berechnung verschiedenster Dauer sein. Bei simplen Aggregaten kann nahezu kein Zeitaufwand benötigt werden, während komplexe Formeln auch längere Berechnungen erfordern können. Aus diesem Grund wurde lediglich ein kurzer Vergleich zwischen Rohdatenverarbeitung und Aggregation gezogen. Auf eine Auswertung konkreter Messwerte wurde verzichtet. Für die Aggregatbildung ist anzumerken, dass die meisten Verarbeitungen sogar schneller ablaufen. Hier findet nur ein Zugriff auf den Cache statt. Neue Messwerte werden der bestehenden Datenstruktur hinzugefügt. Falls alle benötigten Werte vorliegen, muss die Bildung des Aggregats durchgeführt werden. In diesem Fall ist der Zeitaufwand größer. Hier ergibt sich eine längere Verarbeitungszeit durch das Auslesen des Messwerts über die *Presigned-URL*. Ebenfalls wirkt sich die Abfrage und das Löschen der gesammelten Messwerte aus dem Cache auf die Verarbeitungszeit aus. Hinzu kommt noch der variable Aufwand für die Berechnung der Aggregate.

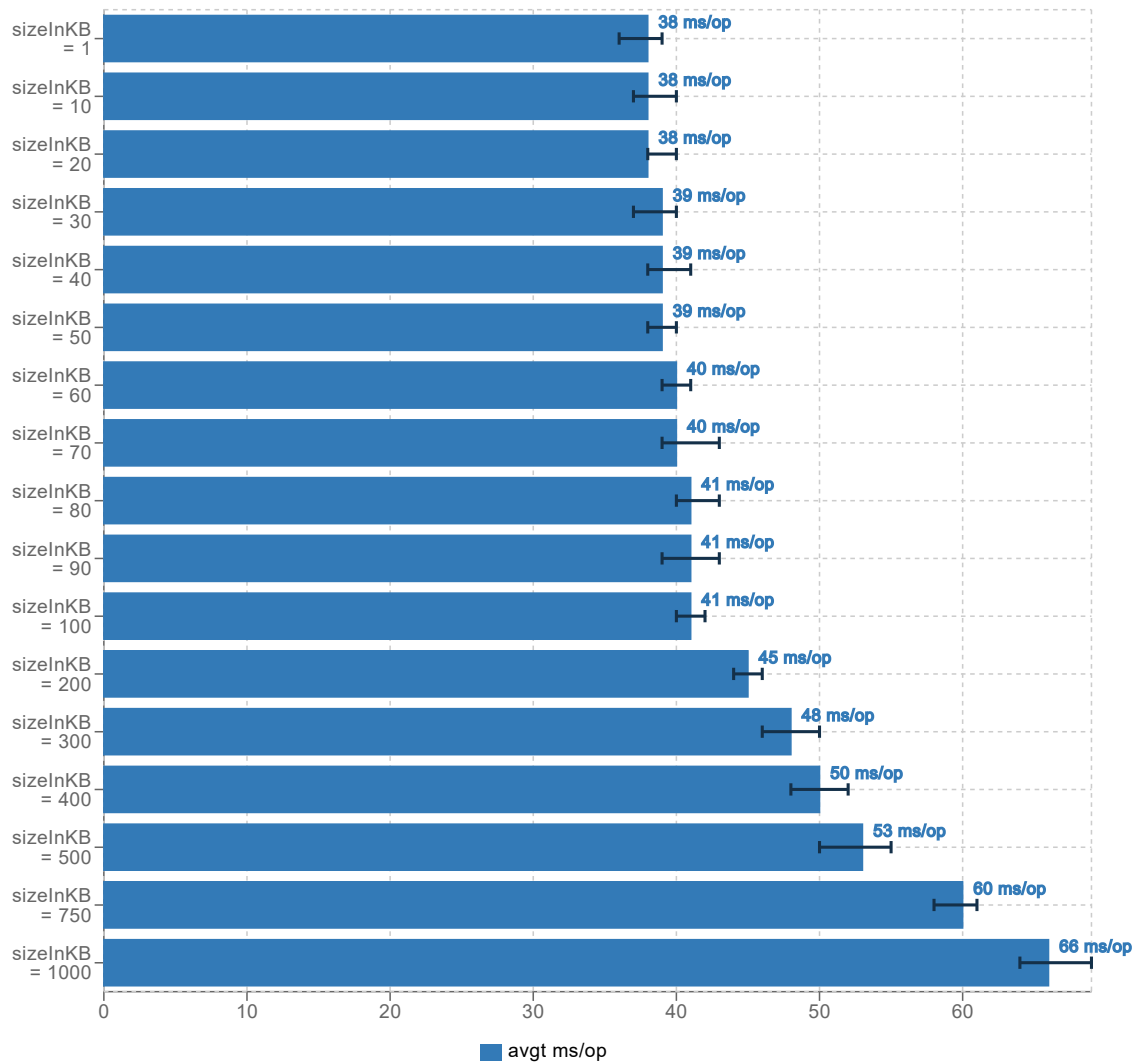


Abbildung 7.1: Auswertung Benchmarks Rohdatenverarbeitung

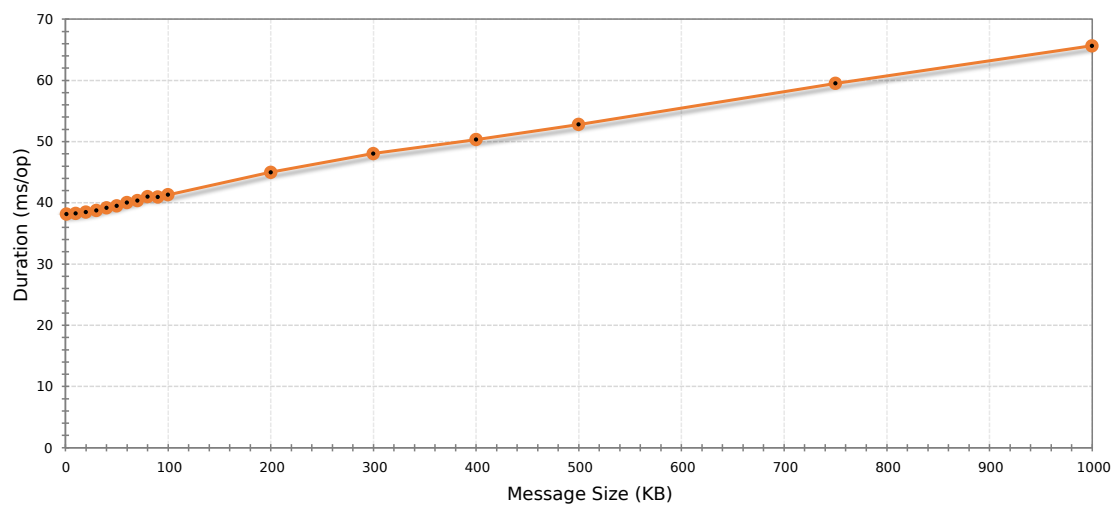


Abbildung 7.2: Verhältnis Nachrichtengröße zu Verarbeitungsdauer

7.2.3 Prometheus Exporter

Der Prometheus Exporter kann analog der Rohdatenverarbeitung evaluiert werden. Es wird die benötigte Zeit zum Sammeln aller Sensormesswerte bestimmt. Dazu wird ein Benchmarking der *CustomCollector-Implementierung* durchgeführt. Es wird eine Methode zur Generierung der Metriken ausgewertet. Diese wird mit verschiedenen Parametern ausgeführt. Dadurch kann die Verarbeitungszeit in Abhängigkeit von der Anzahl an vorliegenden Daten bestimmt werden. Hierbei werden ebenfalls mehrere Messungen in separaten Iterationen ausgeführt. Da die generelle Dauer der Ausführung hier jedoch erheblich größer ist als bei der Rohdatenverarbeitung, wird die Anzahl der Iterationen erhöht. Statt zehn werden fünfzig Messungen vorgenommen. Die Durchführung von mehr Messungen wirkt sich positiv auf die Genauigkeit von berechneten Durchschnittswerten aus. Systemressourcen werden wie bei dem Benchmarking der verarbeitenden Microservices zugewiesen. Da zum Benchmarking bis zu einer halben Million Datensätze exportiert werden sollen, muss hier mehr Speicher zugeteilt werden. Statt 512Mi Speicher muss eine Speichergröße zwischen 1024Mi und 2048Mi verwendet werden. Neben der geänderten Definition in der YAML-Datei ist hier eine weitere Anpassung des *Container-Images* notwendig. In Listing 7.2 ist das entsprechende *Dockerfile* zu sehen. Durch die Parameter wird sichergestellt, dass die JVM einen ausreichend großen Speicherbereich für den *Heap* allokiert. *-Xms1024M* weist die initiale Größe zu. Mit *-Xmx2048M* wird eine Obergrenze für den Speicher festgelegt.

```
1 FROM openjdk:14-alpine
2 ARG JAR_FILE=target/*.jar
3 COPY ${JAR_FILE} app.jar
4 ENTRYPOINT ["java", "-Xms1024M", "-Xmx2048M", "-jar", "/app.jar", "de.
    htw.saar.smartcity.aggregator.exporter.benchmarking.
    ExporterMicroserviceBenchmark"]
```

Listing 7.2: Dockerfile Prometheus Exporter Benchmarking

Wie im Dockerfile zu sehen ist, wird die Testklasse aus der JAR-Datei gestartet. Der Test nutzt eine *In-Memory-Datenbank* statt der sonst verwendeten relationalen Datenbank. Diese wird mit 500.000 Sensoren initialisiert, welche alle exportiert werden sollen. Vor der Ausführung der Messung wird der Cache mit Einträgen zu den entsprechenden Sensoren befüllt. Anschließend wird ein Teil dieser Daten dann abgefragt, aufbereitet und bereitgestellt. Das Benchmarking misst dann, wie lange die Ausführungszeit dieser Routine ist. Im Anhang C.3 ist das Ergebnis ausgegliedert. Die ermittelten Daten sind in Abbildung 7.3 nochmals grafisch dargestellt.

Der Anstieg der Verarbeitungszeit erfolgt nicht immer gleichmäßig. Wird eine Grenze von 75.000 hinzukommenden Sensordaten überschritten, so steigt die Verarbeitungszeit stärker an als sonst. Zusammenfassend kann man jedoch von einem schwachen quadratischen Wachstum sprechen. Da Prometheus alle dreißig Sekunden die Daten über REST ausliest, soll die Routine dieses Zeitfenster nicht überschreiten. Rechnet man die Verarbeitungszeit auf diese Obergrenze hoch, so ergibt sich, dass etwa 475.000 Sensormesswerte innerhalb dieser Zeit gesammelt werden können. Die Abfrage einer großen Menge von Daten über die Memcached-API führt bei längerer Dauer jedoch öfters zu einem Timeout. Aus diesem Grund ist es sinnvoll, große Abfragen in Teilabfragen aufzuteilen. Es wurde eine Obergrenze von hunderttausend Schlüsseln für einen einzelnen Aufruf festgelegt. Zudem wurden weitere Optimierungen der Abfragelogik vorgenommen, welche sich positiv auf die Ausführungszeit auswirkten. Für die geänderte Logik wurde ebenso ein Benchmarking durchgeführt. Dieses kann mit der ursprünglichen Umsetzung verglichen werden. Im Anhang C.4 ist das Ergebnis zu finden. Die durchschnittlichen Messwerte sind in Abbildung 7.4 grafisch zusammengefasst.

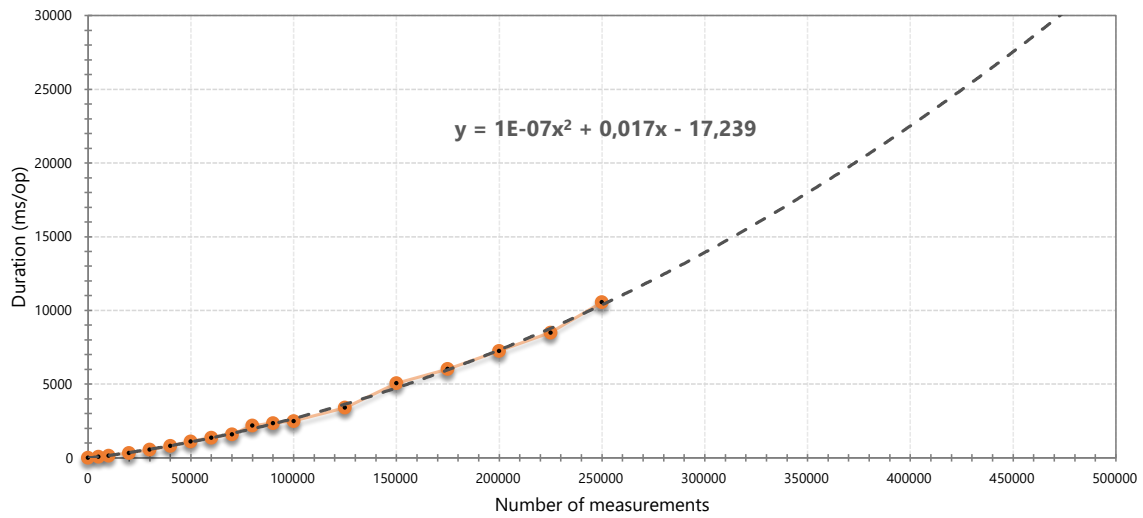


Abbildung 7.3: Messwerte Benchmarking Prometheus Exporter

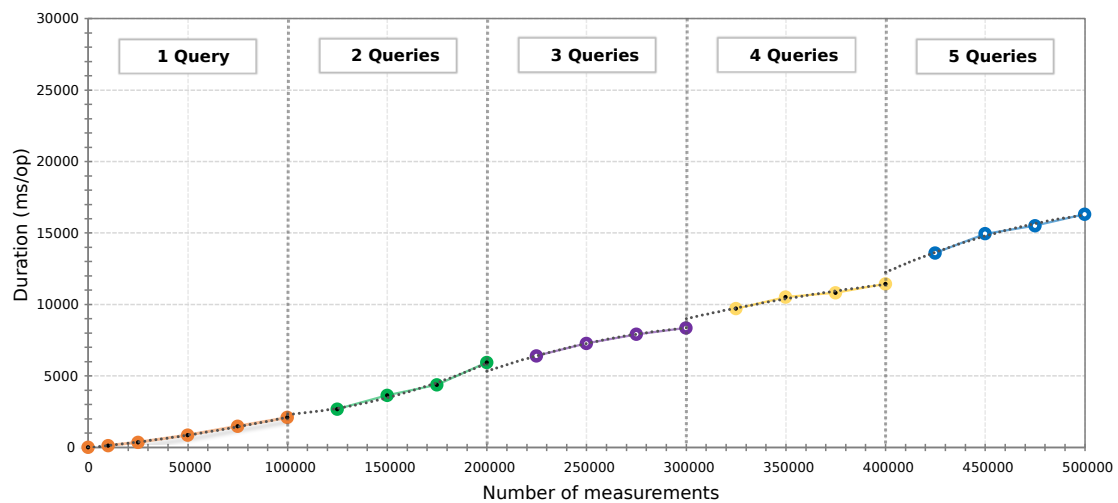


Abbildung 7.4: Messwerte Benchmarking Prometheus Exporter nach Optimierung

Hier ist zu sehen, dass die Änderung eine Optimierung der Verarbeitungszeiten für große Mengen an Daten zur Folge hatte. Für den zeitlichen Rahmen von dreißig Sekunden ist hier eine maximale Anzahl von über 750.000 exportierten Messwerten abschätzbar. Wird diese Grenze erreicht oder angenähert, so sollte eine manuelle Skalierung durchgeführt werden.

7.2.4 Ausblick Skalierung

Anhand der berechneten Verarbeitungszeiten für Messwerte variabler Größen ist eine Maximallast berechenbar. Diese Maximallast gilt jedoch nur für einen einzelnen Pod mit festgelegten Systemressourcen. Sie lässt sich dementsprechend einfach erhöhen, indem die entsprechende Anwendung skaliert wird. Durch das implementierte Autoscaling erfolgt dies ohne eigenes Zutun. Die Skalierung wird lediglich durch die zur Verfügung stehende Hardware begrenzt. Durch das Hinzufügen weiterer Rechenleistung und Speicher kann die Architektur beliebig großes Ausmaß annehmen.

8 Zusammenfassung und Ausblick

Es folgt die Zusammenfassung der Thesis. Die wichtigsten Punkte werden nochmals aufgeführt. Ebenfalls werden Möglichkeiten der Erweiterung aufgezeigt.

8.1 Zusammenfassung

Die Thesis befasst sich mit einem häufigen IoT-Anwendungsfall, den Smart Cities. Es wurde ein Konzept erstellt, das eine möglichst flexible Anbindung von Sensoren ermöglicht. Hierbei stand die Erhebung und Aggregation der Messwerte im Vordergrund. Die Implementierung umfasst Microservices zur Verarbeitung von Rohdaten für Temperatur-, Luftfeuchtigkeits-, Luftqualitäts-, Wasser- und Bildsensoren. Ebenso wurden Microservices zur Aggregation in Echtzeit erstellt. Hier erfolgt die Gruppierung von Temperatur-, Luftfeuchtigkeits- und Luftqualitätsmesswerten zur Bildung von Minima, Maxima und Mittelwerten. Wassersensoren können in Gruppen zusammengefasst werden, die kumulieren, wie viele Sensoren Feuchtigkeit messen. Außerdem wurde ein Microservice zur Berechnung von Taupunkten und ein weiterer zur Bildung von Wärmefluss- und Kondensationswerten erstellt. Dadurch wurden die diversen Anforderungen zu den verschiedenen Arten der Verarbeitung exemplarisch abgebildet. Zudem sind so alle aktuell verwendeten Sensoren des Testfelds angebunden. Die Microservices können über Kubernetes konfiguriert und bereitgestellt werden. Zusammen mit der Anwendung zur Verwaltung von Stammdaten und dem Microservice zum Export an Prometheus wird der Gesamtprozess realisiert. Dieser beginnt mit der Anlage von Sensoren und Gruppen, dem Setzen von Metadaten, der Verarbeitung von Messwerten und endet schließlich mit der Datenansicht und Analyse in Grafana Dashboards. Darüber hinaus ermöglichen die Microservices zur historischen Datenaggregation die Bildung von Aggregaten über längere Zeiträume.

Durch die Erstellung einer erweiterbaren Code-Basis lassen sich neue Microservices mit geringem Zeitaufwand implementieren und bereitstellen. Alle Microservices nutzen die Code-Basis und können als Referenzen für zukünftige Entwicklungen dienen. Die Anbindung verschiedener Sensoren sowie die Bildung neuer Aggregate wird ermöglicht. Dazu muss lediglich ein neuer Microservice erstellt werden, der die gewünschte Funktionalität abbildet. Durch die Code-Basis wird der Aufwand dieser Entwicklung auf die wesentlichen Teile beschränkt. Die Grundfunktionalität ist bereits vorhanden. Die Maven Archetypes verkürzen den zeitlichen Aufwand der Implementierung, indem Templates für Projekte automatisch generiert werden können.

Das System wurde so umgesetzt, dass es sich flexibel skalieren lässt. Wie die Evaluation gezeigt hat, kann ein einzelner Microservice schon mit einer Vielzahl an Sensoren umgehen. Möchte man jedoch tausende Messwerte pro Sekunde verarbeiten, so ist dies ebenfalls möglich. Durch das Deployment in einem Kubernetes-Cluster mit erweiterbaren Ressourcen kann das System beliebig skaliert werden. Es ist somit zum Einsatz in einem kleinen Sensornetzwerk geeignet, kann aber auch für Anwendungsszenarien mit einer Vielzahl an Sensoren verwendet werden.

8.2 Ausblick

Die Komponente zur Pflege von Stammdaten wurde als Prototyp entwickelt. Hier ist eine weiterführende Arbeit geplant, die sich mit der Konzeption einer komplexen Anwendung zur Verwaltung von Metadaten beschäftigt. Diese soll basierend auf einem *Plug-And-Play* Mechanismus Sensoren dem System bekannt machen. Die bestehende Implementierung gibt ein grobes Konzept für Metadaten vor, kann jedoch hier beliebig verändert oder erweitert werden.

Die in Kapitel 6.7 beschriebene Implementierung des Microservices zur Bereitstellung der Daten an Prometheus ermöglicht eine Skalierung, jedoch muss diese manuell durchgeführt werden. Hier ist die Implementierung einer Primary-Secondary-Architektur aufbauend auf dem aktuellen Prototyp angedacht. Damit wäre eine Skalierung flexibel und ohne Aufwand durchführbar. Es wurden die Grundbausteine zur Entwicklung einer Primary-Secondary-Architektur gelegt. Die aktuelle Umsetzung wurde im Hinblick auf eine zukünftige Erweiterung konzipiert. Im Wesentlichen ist hier die Implementierung der Primary-Komponente ausstehend. Der bestehende Microservice würde die Secondary-Komponente bilden. Darüber hinaus müsste noch die Kommunikation zwischen Primary und Secondary realisiert werden.

Das Monitoring der Microservices erfolgt über die Grafana Dashboards. Hier stehen viele Metriken bereit, welche die Anwendungen und das ausführende System beschreiben. Eine mögliche Erweiterung wäre der Einsatz des *Prometheus Alertmanagers*. Dieser kann genutzt werden, um gezielte Benachrichtigungen zu konfigurieren. Für den Fall, dass Metriken kritische Werte annehmen, kann eine Alarmierung über Mail, DevOps oder sonstige Wege erfolgen [6].

Eine weitere Erweiterungsmöglichkeit stellt die Entwicklung neuer Microservices dar. Mit dem Ausbau des Smart City Testfelds könnten die Einsatzmöglichkeiten verschiedenster Sensoren evaluiert werden. Hier müssten dann weitere Microservices zur Verarbeitung der Messungen sowie zur Aggregatbildung erstellt werden. Anwendungen zur Ansteuerung der Sensoren müssten in diesem Fall ebenso entwickelt werden. Auch für die Sensoranbindung wäre es denkbar das bestehende Konzept zu erweitern und eine gleichermaßen flexible Architektur zu schaffen. Mit dieser könnte die Implementierung neuer Sensorlogik noch weiter vereinfacht werden.

Literatur

- [1] *API Code & Client Generator | Swagger Codegen*. <https://swagger.io/tools/swagger-codegen/>. (accessed on 2021-07-05).
- [2] *About Swagger Specification | Documentation | Swagger*. <https://swagger.io/docs/specification/about/>. (accessed on 2021-07-05).
- [3] *About storage drivers | Docker Documentation*. <https://docs.docker.com/storage/storagedriver/>. (accessed on 2021-06-04).
- [4] Shiva Achari. *Hadoop Essentials*. Packt Publishing, 30. Apr. 2015. 194 S. ISBN: 1784396680. URL: https://www.ebook.de/de/product/24132866/shiva_achari_hadoop_essentials.html.
- [5] Tanweer Alam. „A Reliable Communication Framework and Its Use in Internet of Things (IoT)“. In: (Aug. 2020). ISSN: 2456-3307. DOI: 10.36227/techrxiv.12657158.v1.
- [6] *Alertmanager | Prometheus*. <https://prometheus.io/docs/alerting/latest/alertmanager/>. (accessed on 2021-09-01).
- [7] Avi Alkalay. *Object storage benefits, myths and options - Cloud computing news*. <https://www.ibm.com/blogs/cloud-computing/2017/02/01/object-storage-benefits-myths-and-options/>. (accessed on 2021-07-05). Feb. 2017.
- [8] Leonidas G. Anthopoulos. „The Rise of the Smart City“. In: *Public Administration and Information Technology*. Springer International Publishing, 2017, S. 5–45. DOI: 10.1007/978-3-319-57015-0_2.
- [9] Galip Aydin, Ibrahim Riza Hallac und Betül Karakus. „Architecture and Implementation of a Scalable Sensor Data Storage and Analysis System Using Cloud Computing and Big Data Technologies“. In: *Journal of Sensors* 2015 (2015), S. 1–11. DOI: 10.1155/2015/834217.
- [10] Luca Bixio, Giorgio Delzanno, Stefano Rebora und Matteo Rulli. „A Flexible IoT Stream Processing Architecture Based on Microservices“. In: *Information* 11.12 (Dez. 2020), S. 565. DOI: 10.3390/info11120565.
- [11] *Build Modular Spring Applications with FlexiCore | Wizzdi*. <https://wizzdi.com/>. (accessed on 2021-06-04).
- [12] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal. *Pattern-Oriented Software Architecture, a System of Patterns*. WILEY, 1. Aug. 1996. 476 S. ISBN: 0471958697. URL: https://www.ebook.de/de/product/3239671/frank_buschmann_regine_meunier_hans_rohnert_peter_sommerlad_michael_stal_pattern_oriented_software_architecture_a_system_of_patterns.html.
- [13] *Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3)*. <https://aws.amazon.com/s3/>. (accessed on 2021-06-04).
- [14] *Cluster Networking | Kubernetes*. <https://kubernetes.io/docs/concepts/cluster-administration/networking/>. (accessed on 2021-06-04).

- [15] *ConfigMaps* | *Kubernetes*. <https://kubernetes.io/docs/concepts/configuration/configmap/>. (accessed on 2021-06-04).
- [16] *Containers* | *Kubernetes*. <https://kubernetes.io/docs/concepts/containers/>. (accessed on 2021-06-04).
- [17] *Core Technologies*. <https://docs.spring.io/spring-framework/docs/current/reference/html/core.html>. (accessed on 2021-07-04).
- [18] *Current weather data - OpenWeatherMap*. <https://openweathermap.org/current>. (accessed on 2021-08-02).
- [19] *Custom Metrics API* · github.com/kubernetes/community. <https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/custom-metrics-api.md>. (accessed on 2021-07-01).
- [20] *DAO support*. <https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/dao.html>. (accessed on 2021-07-04).
- [21] Shahir Daya. *Microservices from theory to practice : creating applications in IBM Bluemix using the microservices approach*. Poughkeepsie, NY: IBM Corporation, International Technical Support Organization, 2015. ISBN: 0738440817.
- [22] *Deployments* | *Kubernetes*. <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>. (accessed on 2021-06-04).
- [23] Philippe Dobbelaere und Kyumars Sheykh Esmaili. „Kafka versus RabbitMQ“. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM, Juni 2017. DOI: 10.1145/3093742.3093908.
- [24] *Docker Hub - Container Image Library* | *Docker*. <https://www.docker.com/products/docker-hub>. (accessed on 2021-06-04).
- [25] *Docker overview* | *Docker Documentation*. <https://docs.docker.com/get-started/overview/>. (accessed on 2021-06-04).
- [26] *Dockerfile reference* | *Docker Documentation*. <https://docs.docker.com/engine/reference/builder/>. (accessed on 2021-06-04).
- [27] David Dossot. *RabbitMQ essentials : hop straight into developing your own messaging applications by learning how to utilize RabbitMQ*. Birmingham, England: Packt Publishing, 2014. ISBN: 9781783983209.
- [28] *Eclipse Paho* | *The Eclipse Foundation*. <https://www.eclipse.org/paho/>. (accessed on 2021-07-06).
- [29] Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare und Peyman Oreizy. „Reflections on the REST architectural style and "principled design of the modern web architecture"(impact paper award)“. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Aug. 2017. DOI: 10.1145/3106237.3121282.
- [30] Roy Thomas Fielding und Richard N. Taylor. „Architectural Styles and the Design of Network-Based Software Architectures“. AAI9980887. Diss. 2000. ISBN: 0599871180.
- [31] Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides. *Design Patterns*. Prentice Hall, 1. Dez. 1995. ISBN: 0201633612. URL: https://www.ebook.de/de/product/3236753/erich_gamma_richard_helm_ralph_e_johnson_john_vlissides_design_patterns.html.
- [32] *GanttProject - Free Project Management Application*. <https://www.ganttproject.biz/>. (accessed on 2021-06-04).

- [33] *Getting Started | Creating a Multi Module Project*. <https://spring.io/guides/gs/multi-module/>. (accessed on 2021-06-04).
- [34] *Getting started | Grafana Labs*. <https://grafana.com/docs/grafana/latest/getting-started/>. (accessed on 2021-07-01).
- [35] J. Ramon Gil-Garcia, Theresa A. Pardo und Taewoo Nam. „What makes a city smart? Identifying core components and proposing an integrative and comprehensive conceptualization“. In: *Information Polity* 20.1 (Juli 2015), S. 61–87. ISSN: 15701255, 18758754. DOI: 10.3233/IP-150354.
- [36] Seth Gilbert und Nancy Lynch. „Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services“. In: *ACM SIGACT News* 33.2 (Juni 2002), S. 51–59. DOI: 10.1145/564585.564601.
- [37] *Gluster Docs*. <https://docs.gluster.org/en/latest/>. (accessed on 2021-06-04).
- [38] *HTTP API | Prometheus*. <https://prometheus.io/docs/prometheus/latest/querying/api/>. (accessed on 2021-07-01).
- [39] Wilhelm Hasselbring und Guido Steinacker. „Microservice Architectures for Scalability, Agility and Reliability in E-Commerce“. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, Apr. 2017. DOI: 10.1109/icsaw.2017.11.
- [40] *Helm*. <https://helm.sh/>. (accessed on 2021-07-05).
- [41] Michael Hofmann. *Microservices best practices for Java*. Poughkeepsie, NY: IBM Corporation, International Technical Support Organization, 2016. ISBN: 0738442275.
- [42] *Horizontal Pod Autoscaler | Kubernetes*. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. (accessed on 2021-07-01).
- [43] Urs Hunkeler, Hong Linh Truong und Andy Stanford-Clark. „MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks“. In: *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*. IEEE, Jan. 2008. DOI: 10.1109/comswa.2008.4554519.
- [44] Joseph Ingeno. *Software Architect's Handbook : Become a Successful Software Architect by Implementing Effective Architecture Concepts*. Birmingham: Packt Publishing Ltd, 2018. ISBN: 9781788624060.
- [45] *JSON*. <https://www.json.org/json-en.html>. (accessed on 2021-07-05).
- [46] Alvaro Videla Jason J. W. Williams. *Rabbitmq in Action: Distributed Messaging for Everyone*. MANNING PUBN, 1. Mai 2012. 312 S. ISBN: 1935182978. URL: https://www.ebook.de/de/product/16295018/jason_j_w_williams_alvaro_videla_rabbitmq_in_action_distributed_messaging_for_everyone.html.
- [47] Jakob Jenkov. *JMH - Java Microbenchmark Harness*. <http://tutorials.jenkov.com/java-performance/jmh.html>. (accessed on 2021-07-05). Sep. 2015.
- [48] Vasiliki Kalavri und Vladimir Vlassov. „MapReduce: Limitations, Optimizations and Open Issues“. In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, Juli 2013. DOI: 10.1109/trustcom.2013.126.
- [49] *Kubernetes: Production-Grade Container Scheduling and Management*. <https://github.com/kubernetes/kubernetes>. (accessed on 2021-06-04).
- [50] *Labels and Selectors | Kubernetes*. <https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>. (accessed on 2021-06-04).

- [51] Valerie Lampkin. *Building Smarter Planet solutions with MQTT and IBM WebSphere MQ Telemetry*. Place of publication not identified: IBM, 2012. ISBN: 0738437085.
- [52] Jorge E. Luzuriaga, Miguel Perez, Pablo Boronat, Juan Carlos Cano, Carlos Calafate und Pietro Manzoni. „A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks“. In: *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*. IEEE, Jan. 2015. DOI: 10.1109/ccnc.2015.7158101.
- [53] *MapReduce Tutorial*. https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html. (accessed on 2021-06-08).
- [54] *Maven – Guide to Creating Archetypes*. <https://maven.apache.org/guides/mini/guide-creating-archetypes.html>. (accessed on 2021-09-05).
- [55] *Maven – Introduction to Archetypes*. <https://maven.apache.org/guides/introduction/introduction-to-archetypes.html>. (accessed on 2021-09-05).
- [56] *Maven – Introduction to the Build Lifecycle*. <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>. (accessed on 2021-08-11).
- [57] *Maven – Introduction*. <https://maven.apache.org/what-is-maven.html>. (accessed on 2021-07-04).
- [58] *Messaging that just works — RabbitMQ*. <https://www.rabbitmq.com/>. (accessed on 2021-06-04).
- [59] *MinIO | High Performance, Kubernetes Native Object Storage*. <https://min.io/>. (accessed on 2021-06-04).
- [60] *MinIO | The MinIO Quickstart Guide*. <https://docs.min.io/docs/>. (accessed on 2021-06-04).
- [61] *MinIO Object Storage for Hybrid Cloud — MinIO Hybrid Cloud Documentation*. <https://docs.min.io/minio/k8s/>. (accessed on 2021-06-04).
- [62] *Multi-Module Project With Spring Boot | Java Development Journal*. <https://www.javadevjournal.com/spring-boot/multi-module-project-with-spring-boot/>. (accessed on 2021-07-04).
- [63] Morrisson Mutuku und Stephen M.A Muathe. „Nexus Analysis: Internet of Things and Business Performance“. In: *International Journal of Research in Business and Social Science (2147- 4478)* 9.4 (Juli 2020), S. 175–181. DOI: 10.20525/ijrbs.v9i4.726.
- [64] *MySQL Database Service | Oracle*. <https://www.oracle.com/mysql/>. (accessed on 2021-06-04).
- [65] *Namespaces | Kubernetes*. <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>. (accessed on 2021-06-04).
- [66] *Nodes | Kubernetes*. <https://kubernetes.io/docs/concepts/architecture/nodes/>. (accessed on 2021-06-04).
- [67] *Overview - Spark 3.1.2 Documentation*. <https://spark.apache.org/docs/latest/index.html>. (accessed on 2021-06-07).
- [68] *Overview | Prometheus*. <https://prometheus.io/docs/introduction/overview/>. (accessed on 2021-07-01).
- [69] *Overview · memcached/memcached Wiki*. <https://github.com/memcached/memcached/wiki/Overview>. (accessed on 2021-07-01).

- [70] Sebastiano Panichella, Mohammad Rahman und Davide Taibi. „Structural Coupling for Microservices“. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science und Technology Publications, 2021. DOI: 10.5220/0010481902800287.
- [71] Charith Perera, Chi Harold Liu, Srimal Jayawardena und Min Chen. „A Survey on Internet of Things From Industrial Market Perspective“. In: *IEEE Access* 2 (2014), S. 1660–1679. DOI: 10.1109/access.2015.2389854.
- [72] *Persistent Volumes | Kubernetes*. <https://kubernetes.io/docs/concepts/storage/persistent-volumes/>. (accessed on 2021-06-04).
- [73] *Plugin Framework for Java (PF4J)*. <https://github.com/pf4j/pf4j>. (accessed on 2021-06-04).
- [74] *Pods | Kubernetes*. <https://kubernetes.io/docs/concepts/workloads/pods/>. (accessed on 2021-06-04).
- [75] Dan Pritchett. „BASE: An Acid Alternative“. In: *Queue* 6.3 (Mai 2008), S. 48–55. DOI: 10.1145/1394127.1394128.
- [76] Stefan Prodan. *Kubernetes Horizontal Pod Autoscaler with Prometheus custom metrics*. <https://stefanprodan.com/2018/kubernetes-horizontal-pod-autoscaler-prometheus-metrics/>. (accessed on 2021-07-01). März 2018.
- [77] *Querying basics | Prometheus*. <https://prometheus.io/docs/prometheus/latest/querying/basics/>. (accessed on 2021-07-01).
- [78] *REST API Documentation Tool | Swagger UI*. <https://swagger.io/tools/swagger-ui/>. (accessed on 2021-07-05).
- [79] David Rensin. *Kubernetes*. City: O'Reilly Media, Inc, 2015. ISBN: 9781491931875.
- [80] *ReplicaSet | Kubernetes*. <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>. (accessed on 2021-07-01).
- [81] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Techn. Ber. Aug. 2018. DOI: 10.17487/rfc8446.
- [82] *Resource metrics pipeline | Kubernetes*. <https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/>. (accessed on 2021-07-01).
- [83] Mark Richards. *Software architecture patterns : understanding common architecture patterns and when to use them*. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 9781491924242.
- [84] Chris Richardson. *Microservice Patterns*. Manning, 1. Juni 2019. ISBN: 1617294543. URL: https://www.ebook.de/de/product/29971389/chris_richardson_microservice_patterns.html.
- [85] *Secrets | Kubernetes*. <https://kubernetes.io/docs/concepts/configuration/secret/>. (accessed on 2021-06-04).
- [86] *Service | Kubernetes*. <https://kubernetes.io/docs/concepts/services-networking/service/>. (accessed on 2021-06-04).
- [87] *Sharing an object with a presigned URL - Amazon Simple Storage Service*. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html>. (accessed on 2021-07-01).
- [88] *Shutter Condensation Calculator*. <https://www.builditsolar.com/References/Calculators/Window/condensation.html>. (accessed on 2021-06-08).

- [89] *Spark Streaming - Spark 3.1.2 Documentation*. <https://spark.apache.org/docs/latest/streaming-programming-guide.html>. (accessed on 2021-06-07).
- [90] *Spring | Microservices*. <https://spring.io/microservices>. (accessed on 2021-06-04).
- [91] *Spring Boot Actuator: Production-ready Features*. <https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html>. (accessed on 2021-09-01).
- [92] *Spring Boot Reference Documentation*. <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>. (accessed on 2021-07-04).
- [93] *Spring Boot*. <https://spring.io/projects/spring-boot>. (accessed on 2021-07-05).
- [94] *Spring Data JPA - Reference Documentation*. <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>. (accessed on 2021-07-05).
- [95] *Spring Data JPA*. <https://spring.io/projects/spring-data-jpa>. (accessed on 2021-07-05).
- [96] *Spring Data*. <https://spring.io/projects/spring-data>. (accessed on 2021-07-04).
- [97] *Spring Framework*. <https://spring.io/projects/spring-framework>. (accessed on 2021-07-04).
- [98] *Storage Classes | Kubernetes*. <https://kubernetes.io/docs/concepts/storage/storage-classes/>. (accessed on 2021-06-04).
- [99] Sachin Arun Thanekar, K. Subrahmanyam und A. B. Bagwan. „A Study on MapReduce: Challenges and Trends“. In: *Indonesian Journal of Electrical Engineering and Computer Science* 4.1 (Okt. 2016), S. 176. DOI: 10.11591/ijeecs.v4.i1.pp176-183.
- [100] *The Java Persistence Query Language (Release 7)*. <https://docs.oracle.com/javaee/7/tutorial/persistence-querylanguage.htm>. (accessed on 2021-07-05).
- [101] *The Official YAML Web Site*. <https://yaml.org/>. (accessed on 2021-06-04).
- [102] Suraj Tripathi. *Stateless Microservices Accelerate Change*. <https://www.linkedin.com/pulse/stateless-microservices-accelerate-change-suraj-tripathi>. (accessed on 2021-07-05). Juni 2020.
- [103] Christian Ullenboom. *Java ist auch eine Insel*. Rheinwerk Verlag GmbH, 1. Juli 2020. 1246 S. ISBN: 3836277379. URL: https://www.ebook.de/de/product/38777293/christian_ullenboom_java_ist_auch_eine_insel.html.
- [104] *Understanding Kubernetes Objects | Kubernetes*. <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>. (accessed on 2021-06-04).
- [105] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune und John Wilkes. „Large-scale cluster management at Google with Borg“. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM, Apr. 2015. DOI: 10.1145/2741948.2741964.
- [106] Steve Vinoski. „Advanced Message Queuing Protocol“. In: *IEEE Internet Computing* 10.6 (Nov. 2006), S. 87–89. DOI: 10.1109/mic.2006.116.
- [107] Werner Vogels. „Eventually consistent“. In: *Communications of the ACM* 52.1 (Jan. 2009), S. 40–44. DOI: 10.1145/1435417.1435432.

- [108] *Volumes | Kubernetes*. <https://kubernetes.io/docs/concepts/storage/volumes/>. (accessed on 2021-06-04).
- [109] *Was ist die 0/100-Methode? - Wissen kompakt - t2informatik*. <https://t2informatik.de/wissen-kompakt/0-100-methode/>. (accessed on 2021-08-18).
- [110] *Weather API - OpenWeatherMap*. <https://openweathermap.org/api>. (accessed on 2021-08-02).
- [111] *Web on Servlet Stack*. <https://docs.spring.io/spring-framework/docs/current/reference/html/web.html>. (accessed on 2021-07-05).
- [112] *Welcome! | minikube*. <https://minikube.sigs.k8s.io/docs/>. (accessed on 2021-07-05).
- [113] *What is Object/Relational Mapping? - Hibernate ORM*. <https://hibernate.org/orm/what-is-an-orm/>. (accessed on 2021-06-09).
- [114] *What is a Container? | App Containerization | Docker*. <https://www.docker.com/resources/what-container>. (accessed on 2021-06-04).
- [115] *Your relational data. Objectively. - Hibernate ORM*. <https://hibernate.org/orm/>. (accessed on 2021-06-09).
- [116] *Übersicht über die Hyper-V-Technologie | Microsoft Docs*. <https://docs.microsoft.com/de-de/windows-server/virtualization/hyper-v/hyper-v-technology-overview>. (accessed on 2021-07-12).
- [117] *grafana/grafana: The open and composable observability and data visualization platform. Visualize metrics, logs, and traces from multiple sources like Prometheus, Loki, Elasticsearch, InfluxDB, Postgres and many more*. <https://github.com/grafana/grafana>. (accessed on 2021-07-01).
- [118] *helm/helm: The Kubernetes Package Manager*. <https://github.com/helm/helm>. (accessed on 2021-07-05).
- [119] *kubernetes-sigs/prometheus-adapter: An implementation of the custom.metrics.k8s.io API using Prometheus*. <https://github.com/kubernetes-sigs/prometheus-adapter>. (accessed on 2021-07-01).
- [120] *learnk8s/spring-boot-k8s-hpa: Autoscaling Spring Boot with the Horizontal Pod Autoscaler and custom metrics on Kubernetes*. <https://github.com/learnk8s/spring-boot-k8s-hpa>. (accessed on 2021-07-12).
- [121] *memcached - a distributed memory object caching system*. <https://memcached.org/>. (accessed on 2021-07-01).
- [122] *openjdk/jmh: Java Microbenchmark Harness (JMH)*. <https://github.com/openjdk/jmh>. (accessed on 2021-07-05).
- [123] *prometheus-operator/prometheus-operator: Prometheus Operator creates/configures/manages Prometheus clusters atop Kubernetes*. <https://github.com/prometheus-operator/prometheus-operator>. (accessed on 2021-07-01).
- [124] *prometheus/client_java: Prometheus instrumentation library for JVM applications*. https://github.com/prometheus/client_java. (accessed on 2021-07-09).
- [125] *prometheus/prometheus: The Prometheus monitoring system and time series database*. <https://github.com/prometheus/prometheus>. (accessed on 2021-07-01).
- [126] *prometheus/pushgateway: Push acceptor for ephemeral and batch jobs*. <https://github.com/prometheus/pushgateway>. (accessed on 2021-07-01).

Abbildungsverzeichnis

3.1	Plugin-Architektur des Aggregator Service	24
3.2	Klassendiagramm Prototyp Plugin-Architektur	25
5.1	Architektur Smart City Testfeld	32
5.2	Message Broker Exchanges, Queues und Bindings	34
5.3	Kommunikation Microservices	36
5.4	Entitäten zur Abbildung der Aggregation	38
5.5	Bausteinsicht Microservices	40
5.6	Konzept automatische Skalierung (Grafik orientiert an [76])	43
5.7	Konzept Prometheus Exporter	45
6.1	Models zur Aggregation	50
7.1	Auswertung Benchmarks Rohdatenverarbeitung	71
7.2	Verhältnis Nachrichtengröße zu Verarbeitungsdauer	71
7.3	Messwerte Benchmarking Prometheus Exporter	73
7.4	Messwerte Benchmarking Prometheus Exporter nach Optimierung	73
B.1	Übersicht Paketstruktur	100
B.2	Klassendiagramm Entitäten	101
B.3	Klassendiagramm Modelle Teil 1	102
B.4	Klassendiagramm Modelle Teil 2	103
B.5	Klassendiagramm Konfiguration	104
B.6	Klassendiagramm Broker	105
B.7	Klassendiagramm Handler	106
B.8	Klassendiagramm Storage	107
B.9	Klassendiagramm Exporter	108
B.10	Dashboard Microservices Monitoring Teil 1	111
B.11	Dashboard Microservices Monitoring Teil 2	112
B.12	Dashboard RabbitMQ Monitoring	113
B.13	Dashboard Sensor Metriken Rohdaten	114
B.14	Dashboard Sensor Metriken Aggregate	115
B.15	Dashboard Sensor Metriken Wassermessungen und Aggregate	116

Tabellenverzeichnis

2.1	Docker-Befehle	7
2.2	HTTP-Commands	15
5.1	Beschreibung der Komponenten von Abbildung 5.5	41
7.1	Erfüllung funktionaler Anforderungen	68
7.2	Erfüllung nichtfunktionaler Anforderungen	69
C.1	Durchschnittliche Verarbeitungszeit in Relation Größe der Messwerte . . .	123
C.2	Durchschnittliche Verarbeitungszeit in Relation zur Anzahl an Datensätzen	124
C.3	Durchschnittliche Verarbeitungszeit in Relation zur Anzahl an Datensätzen	125

Listings

2.1	Beispiel Dockerfile	7
2.2	Beispiel Deployment	9
2.3	Beispiel PersistentVolume	10
2.4	Beispiel PersistentVolumeClaim	10
2.5	Beispiel Service	11
2.6	Beispiel Horizontal Pod Autoscaler	12
6.1	Setzen des SSL-Kontexts	52
6.2	Queue-Initialisierung und Callback-Funktion	53
6.3	Objekt aus Object Storage abrufen	55
6.4	Erzeugung Presigned URL und Publish	56
6.5	Befehl zur Erzeugung eines rohdatenverarbeitenden Microservices	61
6.6	Beispiel direkte Instrumentalisierung mit Gauge [124]	61
7.1	Auszug Testklasse des Benchmarking-Microservice	69
7.2	Dockerfile Prometheus Exporter Benchmarking	72
B.1	Temperature Microservice Deployment	93
B.2	Microservices ServiceMonitor	94
B.3	Code der Taupunktberechnung	95
B.4	Code der Wärmefluss- und Kondensationsberechnung	96
B.5	Deployment Prometheus Exporter Microservice	99
B.6	Prometheus Metriken	109
B.7	Deployment RabbitMQ Testumgebung	117
C.1	Benchmarking Microservice Deployment	119
C.2	Auszug JMH-Ergebnisdatei	120

Abkürzungsverzeichnis

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
Amazon S3	Amazon Simple Storage Service
CAP	Consistency, Availability and Partition Tolerance
CA	Certificate Authority
CIA	Confidentiality, Integrity and Availability
CRUD	Create, read, update and delete
DAO	Data Access Object
DNS	Domain Name System
DSL	Distributed Systems Lab
DStream	Discretized Stream
HPA	Horizontal Pod Autoscaler
htw saar	Hochschule für Technik und Wirtschaft des Saarlandes
IoC	Inversion of Control
IoT	Internet of Things
JAR	Java Archive
JMH	Java Microbenchmark Harness
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
k8s	Kubernetes
MOM	Message Oriented Middleware
MQTT	Message Queuing Telemetry Transport
MVC	Model View Controller
OAS	OpenAPI Specification
ORM	Object Relational Mapping
PF4J	Plugin Framework for Java
POJO	Plain Old Java Object
POM	Project Object Model
PVC	Persistent Volume Claim
PV	Persistent Volume

Abkürzungsverzeichnis

PromQL	Prometheus Query Language
RDBMS	Relationales Datenbankmanagementsystem
RDDs	Resilient Distributed Datasets
REST	Representational State Transfer
SDK	Software Development Kit
SQL	Structured Query Language
SSL	Secure Sockets Layer
SoC	Separation of Concerns
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Anhang

A Weiterführende Informationen zur Analyse

A.1 Use Cases

Aktor – Sensor:

- Digitale und analoge Ports auslesen
- Daten über MQTT bereitstellen

Aktor – Admin:

- Stammdaten verwalten
- Anwendungen zentral konfigurieren
- Microservices überwachen
- Systemauslastung überwachen
- Logging einsehen
- Automatische Skalierung nachvollziehen

Aktor – Entwickler:

- Neue Sensoren simpel anbinden
- Neue Microservices mit Minimalaufwand erstellen
- Anbindung neuer Microservices ohne Änderungen
- Komplexe Formeln abbilden können
- Deployment und Skalierung über Kubernetes

Aktor – Microservices-Architektur:

- Sensordaten in Echtzeit verarbeiten
- Aggregate in Echtzeit bilden
- Historische Aggregate am Tages-, Wochen- oder Monatsende erzeugen
- Daten bereitstellen

B Weiterführende Informationen zur Implementierung

B.1 Temperature Microservice Deployment

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: temperature-deployment
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: temperature-ms
10   template:
11     metadata:
12       labels:
13         app: temperature-ms
14     spec:
15       containers:
16         - name: temperature-ms
17           image: user/temperature-ms
18           envFrom:
19             - configMapRef:
20               name: general-config
21             - secretRef:
22               name: general-secrets
23       ports:
24         - containerPort: 8080
25           name: http
26 ---
27 apiVersion: v1
28 kind: Service
29 metadata:
30   name: temperature-service
31   labels:
32     app: temperature-ms
33     monitoring: "true"
34 spec:
35   type: ClusterIP
36   selector:
37     app: temperature-ms
38   ports:
39     - port: 8080
40       name: http
41 ---
42 apiVersion: autoscaling/v2beta2
43 kind: HorizontalPodAutoscaler
44 metadata:
45   name: temperature-hpa
46 spec:
47   scaleTargetRef:
48     apiVersion: apps/v1
49     kind: Deployment
```

```
50     name: temperature-deployment
51     minReplicas: 1
52     maxReplicas: 10
53     metrics:
54     - type: Pods
55       pods:
56         metric:
57           name: microservice_activity
58         target:
59           type: AverageValue
60           averageValue: 75
```

Listing B.1: Temperature Microservice Deployment

B.2 Microservices ServiceMonitor

```
1  apiVersion: monitoring.coreos.com/v1
2  kind: ServiceMonitor
3  metadata:
4    name: microservice-monitor
5    labels:
6      release: my-prom
7  spec:
8    namespaceSelector:
9      any: true
10   selector:
11     matchLabels:
12       monitoring: "true"
13   endpoints:
14     - port: http
15       path: "/metrics"
16       interval: 30s
17       scrapeTimeout: 30s
18     - port: http
19       path: "/actuator/prometheus"
20       interval: 30s
21       scrapeTimeout: 30s
```

Listing B.2: Microservices ServiceMonitor

B.3 Code der Taupunktberechnung

```

1  @Override
2  protected void addCombinators() {
3      CombinatorFunction<Double> dewpointFunction = (gms) -> {
4          Map<Producer, Measurement<Double>> newMap =
5              gms.getProducerIdMeasurementMap()
6                  .entrySet()
7                  .stream()
8                  .collect(Collectors.toMap(
9                      e -> producerService.findProducerById(e.getKey()).get(),
10                     e -> e.getValue())
11              );
12          Producer temperatureProducer = newMap.keySet()
13              .stream()
14              .filter(p -> p.getDataType().getName()
15                  .equals(applicationProperties.getTemperatureDataTypeName()))
16              .findFirst()
17              .orElseThrow(
18                  () -> new MeasurementException("No temperature present")
19              );
20          Producer humidityProducer = newMap.keySet()
21              .stream()
22              .filter(p -> p.getDataType().getName()
23                  .equals(applicationProperties.getHumidityDataTypeName()))
24              .findFirst()
25              .orElseThrow(
26                  () -> new MeasurementException("No humidity present")
27              );
28
29          Double temperatureValue =
30              newMap.get(temperatureProducer).getValue();
31          Double humidityValue =
32              newMap.get(humidityProducer).getValue();
33          Double alpha =
34              Math.log(humidityValue / 100) +
35              A * temperatureValue / (B + temperatureValue);
36          Double dewpointValue = B * alpha / (A - alpha);
37          return Math.round(dewpointValue * 100) / 100.0;
38      };
39      String dewpointFunctionName = "dewpoint-combinator";
40      CombinatorModel<Double> combinatorModel =
41          new CombinatorModel<>(dewpointFunctionName, dewpointFunction);
42      combinatorModels.add(combinatorModel);
43  }

```

Listing B.3: Code der Taupunktberechnung

B.4 Code der Wärmefluss- und Kondensationsberechnung

```
1  @Override
2  protected void addCombinators() {
3      Function<Map<Long, Measurement<Double>>, Map<Producer, Measurement<
4          Double>>> idToProducerMapper =
5          map -> map
6              .entrySet()
7              .stream()
8              .collect(Collectors.toMap(e -> producerService.findProducerById(
9                  e.getKey()).get(),
10                  e -> e.getValue())
11              );
12      Function<Map<Producer, Measurement<Double>>, Optional<Producer>>
13          dewPointProducerGetter =
14          map -> map
15              .keySet()
16              .stream()
17              .filter(
18                  p -> p.getDataType().getName()
19                      .equals(applicationProperties.getDewpointDataTypeName())
20              )
21              .findFirst();
22      Function<Map<Producer, Measurement<Double>>, Optional<Producer>>
23          insideTemperatureProducerGetter =
24          map -> map
25              .keySet()
26              .stream()
27              .filter(p -> p.getDataType().getName()
28                  .equals(applicationProperties.getTemperatureDataTypeName()) &&
29                  p.getTags()
30                  .stream()
31                  .anyMatch(
32                      t -> t.getName()
33                          .equals(applicationProperties.getTagInsideTemperature())
34                  )
35              )
36              .findFirst();
37      Function<Map<Producer, Measurement<Double>>, Optional<Producer>>
38          outsideTemperatureProducerGetter =
39          map -> map
40              .keySet()
41              .stream()
42              .filter(p -> p.getDataType().getName()
43                  .equals(applicationProperties.getTemperatureDataTypeName()) &&
44                  p.getTags()
45                  .stream()
46                  .anyMatch(
47                      t -> t.getName()
48                          .equals(applicationProperties.getTagOutsideTemperature())
49                  )
50              )
51              .findFirst();
52      CombinatorFunction<Double> heatFluxFunction = (gms) -> {
53          log.info("Start heatflux combinator function");
54          Map<Producer, Measurement<Double>> producerMap =
55              idToProducerMapper.apply(gms.getProducerIdMeasurementMap());
56          Optional<Producer> dewPointProducer =
57              dewPointProducerGetter.apply(producerMap);
58          Optional<Producer> outsideTemperatureProducer =
```

B.4 Code der Wärme- und Kondensationsberechnung

```
57     outsideTemperatureProducerGetter.apply(producerMap);
58
59     Double dewPointValue = producerMap.get(
60         dewPointProducer
61         .orElseThrow(
62             () -> new MeasurementException("Missing dewpoint value.")
63         )
64     )
65     .getValue();
66     Double outsideTemperatureValue = producerMap.get(
67         outsideTemperatureProducer
68         .orElseThrow(
69             () -> new MeasurementException("Missing outside temperature.")
70         )
71     )
72     .getValue();
73
74     Double u = 3.0;
75     Group group = groupService.findGroupById(gms.getGroupId())
76     .orElseThrow(() -> new MeasurementException("Group invalid"));
77     Optional<FormulaItemValue> formulaItemValue = group.getValues()
78     .stream()
79     .filter(g -> g.getFormulaItem().getName()
80         .equals(applicationProperties.getFormulaItemNameUValue()))
81     .findFirst();
82     try {
83         if(formulaItemValue.isPresent())
84             u = Double.valueOf(formulaItemValue.get().getValue());
85         else
86             u = Double.valueOf(
87                 applicationProperties.getFormulaItemNameUValueDefault()
88             );
89     } catch (NumberFormatException numberFormatException) {
90         log.error("Formula item value not be convertible to number");
91     }
92
93     Double heatflux = (dewPointValue - outsideTemperatureValue) * u;
94     log.info("End heatflux combinator function");
95     return Math.round(heatflux * 100) / 100.0;
96 };
97 String heatFluxFunctionName = "heatflux-combinator";
98 CombinatorModel<Double> combinatorModel =
99     new CombinatorModel<>(heatFluxFunctionName, heatFluxFunction);
100 combinatorModels.add(combinatorModel);
101
102 CombinatorFunction<Double> shutterFunction = (gms) -> {
103     log.info("Start shutter combinator function");
104     Map<Producer, Measurement<Double>> producerMap =
105         idToProducerMapper.apply(gms.getProducerIdMeasurementMap());
106
107     Optional<Producer> dewPointProducer =
108         dewPointProducerGetter.apply(producerMap);
109     Optional<Producer> insideTemperatureProducer =
110         insideTemperatureProducerGetter.apply(producerMap);
111
112     Double dewPointValue = producerMap.get(
113         dewPointProducer
114         .orElseThrow(
115             () -> new MeasurementException("Missing dewpoint value.")
116         )
117     )
118     .getValue();
```

B Weiterführende Informationen zur Implementierung

```
119 Double insideTemperatureValue = producerMap.get(  
120     insideTemperatureProducer  
121     .orElseThrow(  
122         () -> new MeasurementException("Missing inside temperature.")  
123     )  
124 )  
125     .getValue();  
126  
127 Double heatflux = heatFluxFunction.apply(gms);  
128 Double shutter =  
129     heatflux / (insideTemperatureValue - dewPointValue);  
130 log.info("End shutter combinator function");  
131 return Math.round(shutter * 100) / 100.0;  
132 };  
133 String shutterCombinatorName = "shutter-combinator";  
134 CombinatorModel<Double> combinatorModelShutter =  
135     new CombinatorModel<>(shutterCombinatorName, shutterFunction);  
136 combinatorModels.add(combinatorModelShutter);  
137 }
```

Listing B.4: Code der Wärmefluss- und Kondensationsberechnung

B.5 Deployment Prometheus Exporter Microservice

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: prom-exporter-deployment
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: prom-exporter-ms
10   template:
11     metadata:
12       labels:
13         app: prom-exporter-ms
14     spec:
15       containers:
16         - name: prom-exporter-ms
17           image: user/prom-exporter-ms
18           envFrom:
19             - configMapRef:
20               name: general-config
21             - secretRef:
22               name: general-secrets
23           ports:
24             - containerPort: 8080
25               name: http
26 ---
27 apiVersion: v1
28 kind: Service
29 metadata:
30   name: prom-exporter-service
31   labels:
32     app: prom-exporter-ms
33     monitoring: "true"
34 spec:
35   type: ClusterIP
36   selector:
37     app: prom-exporter-ms
38   ports:
39     - port: 8080
40       name: http
```

Listing B.5: Deployment Prometheus Exporter Microservice

B.6 Übersicht Paketstruktur

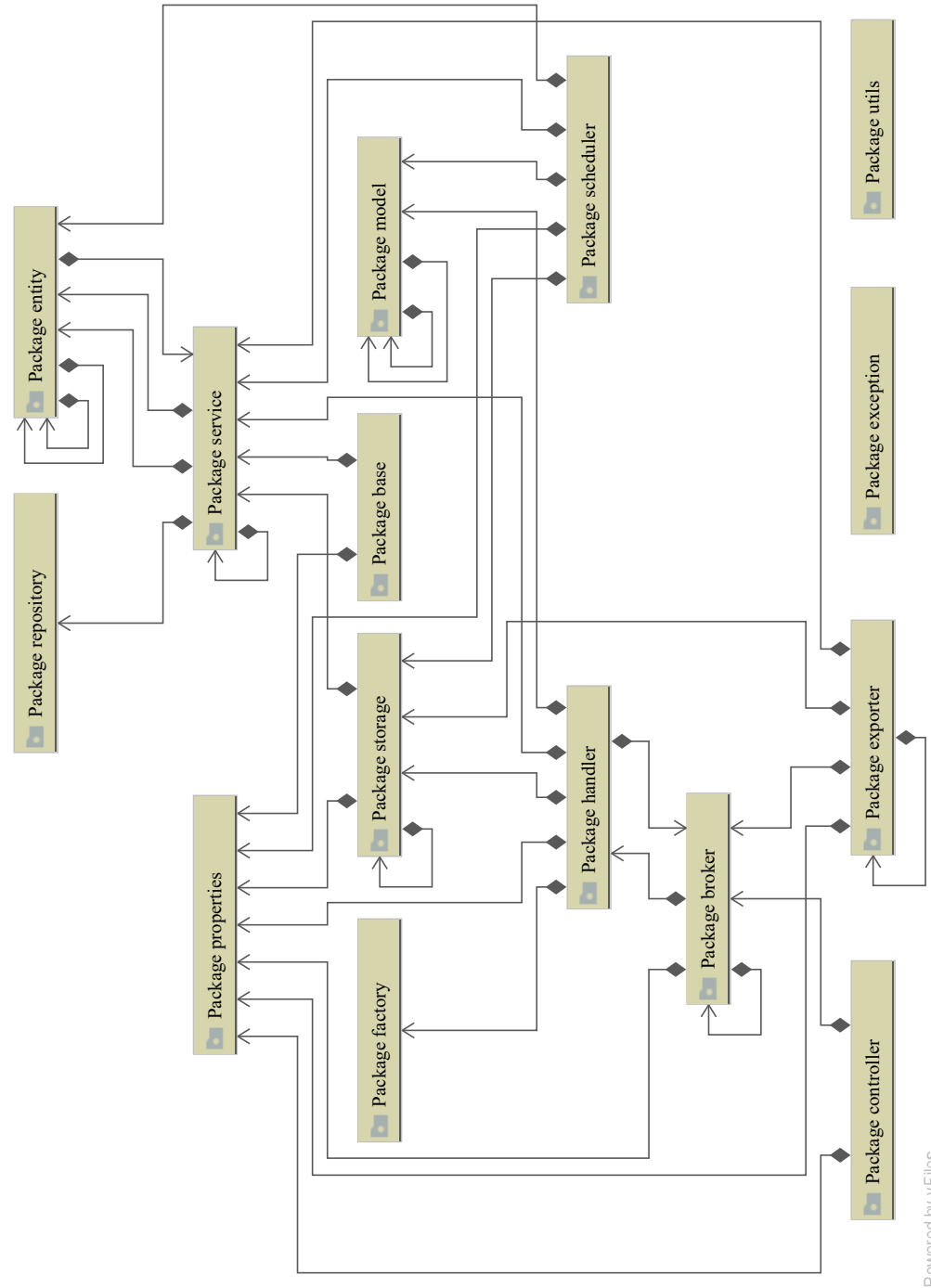


Abbildung B.1: Übersicht Paketstruktur

B.8 Klassendiagramm Modelle

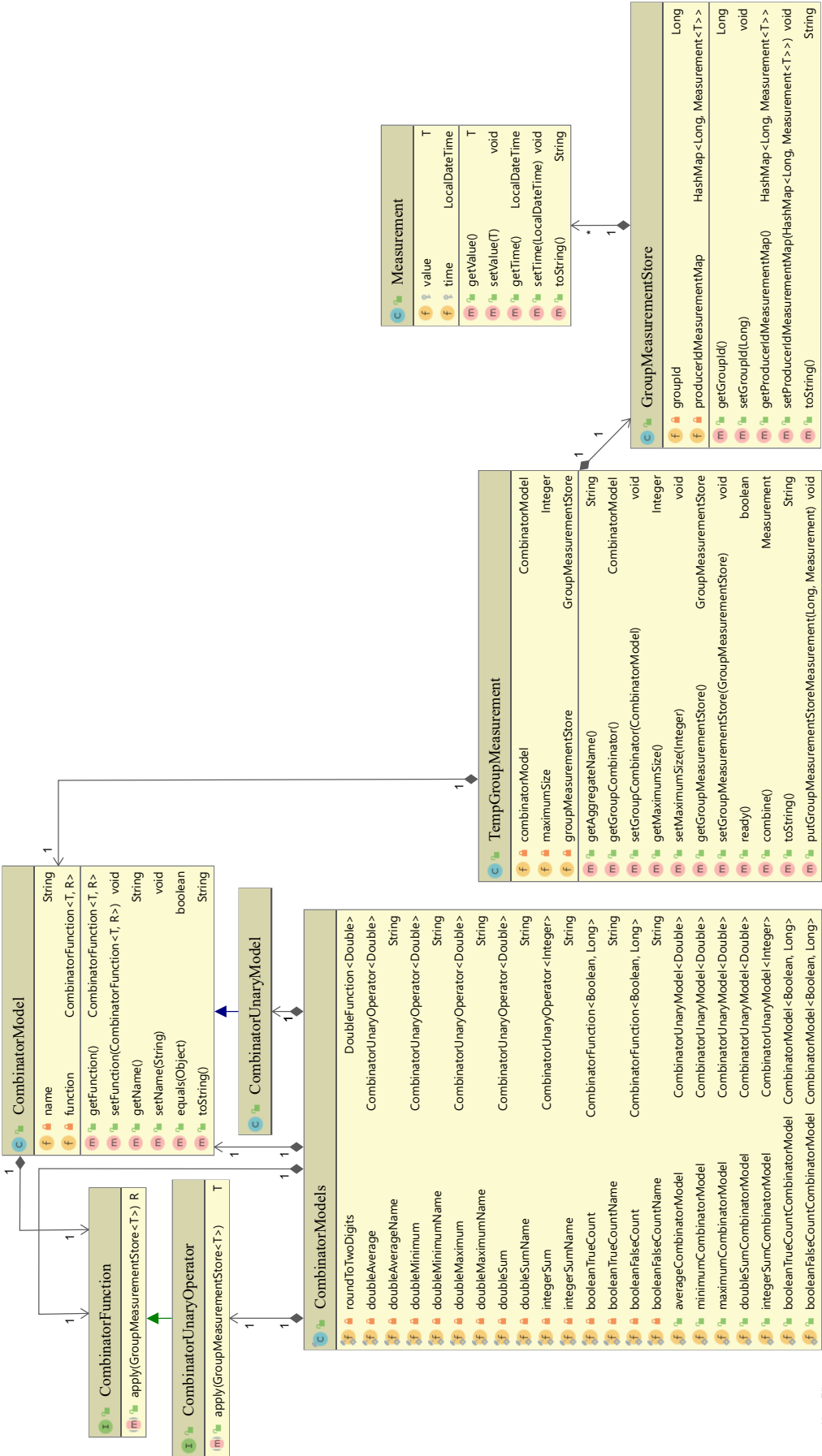
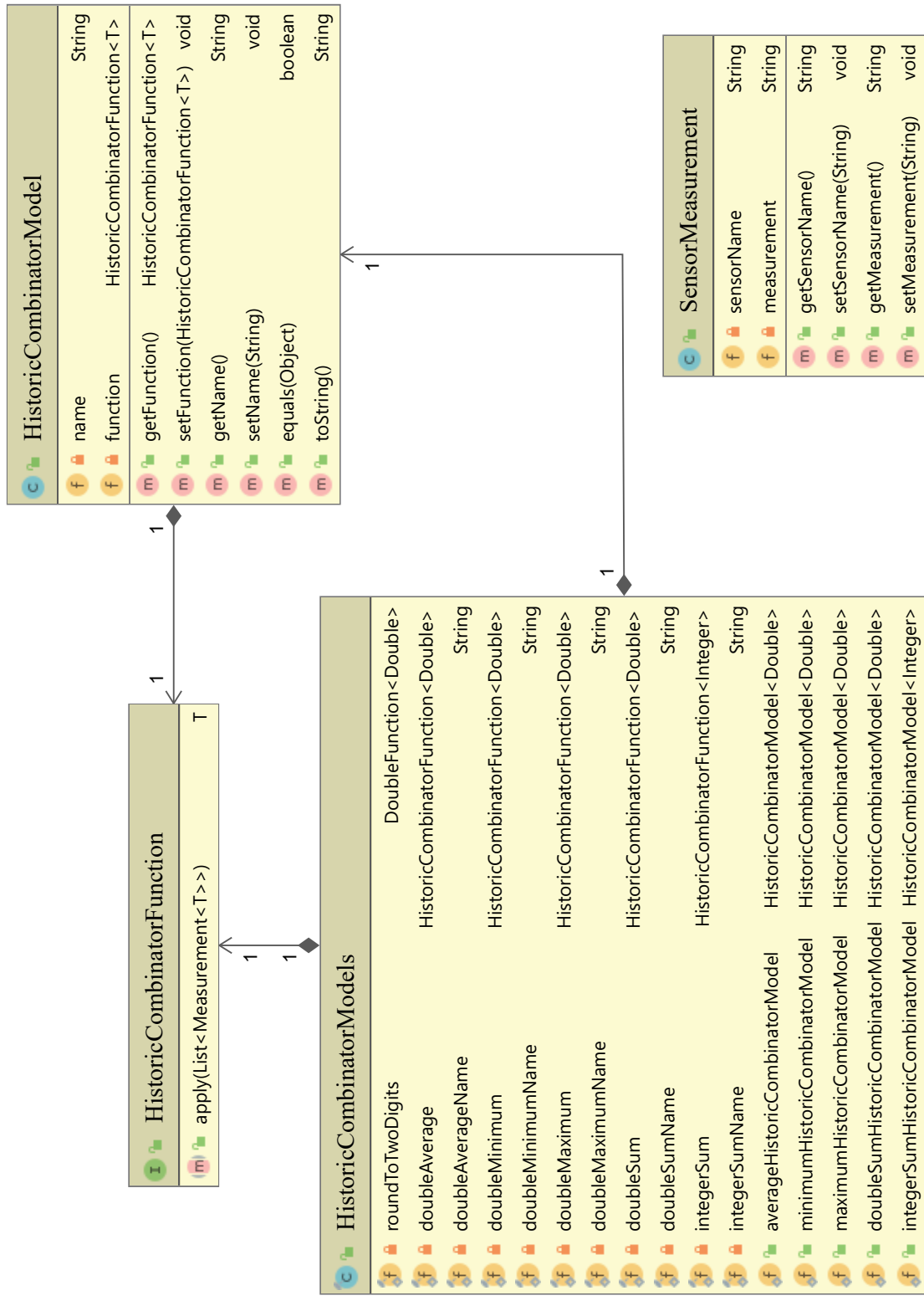


Abbildung B.3: Klassendiagramm Modelle Teil 1



Powered by yFiles

Abbildung B.4: Klassendiagramm Modelle Teil 2

B.9 Klassendiagramm Konfiguration

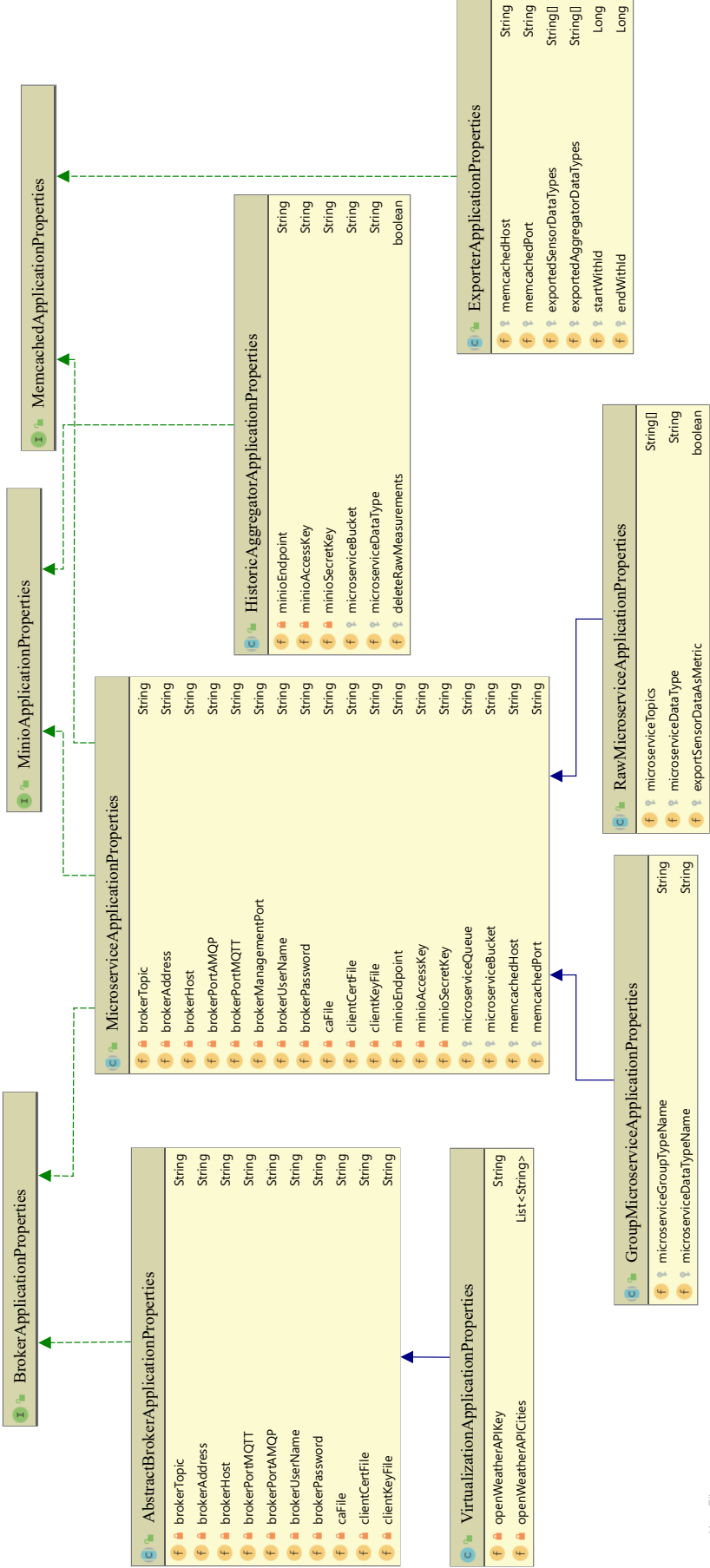


Abbildung B.5: Klassendiagramm Konfiguration

B.10 Klassendiagramm Broker

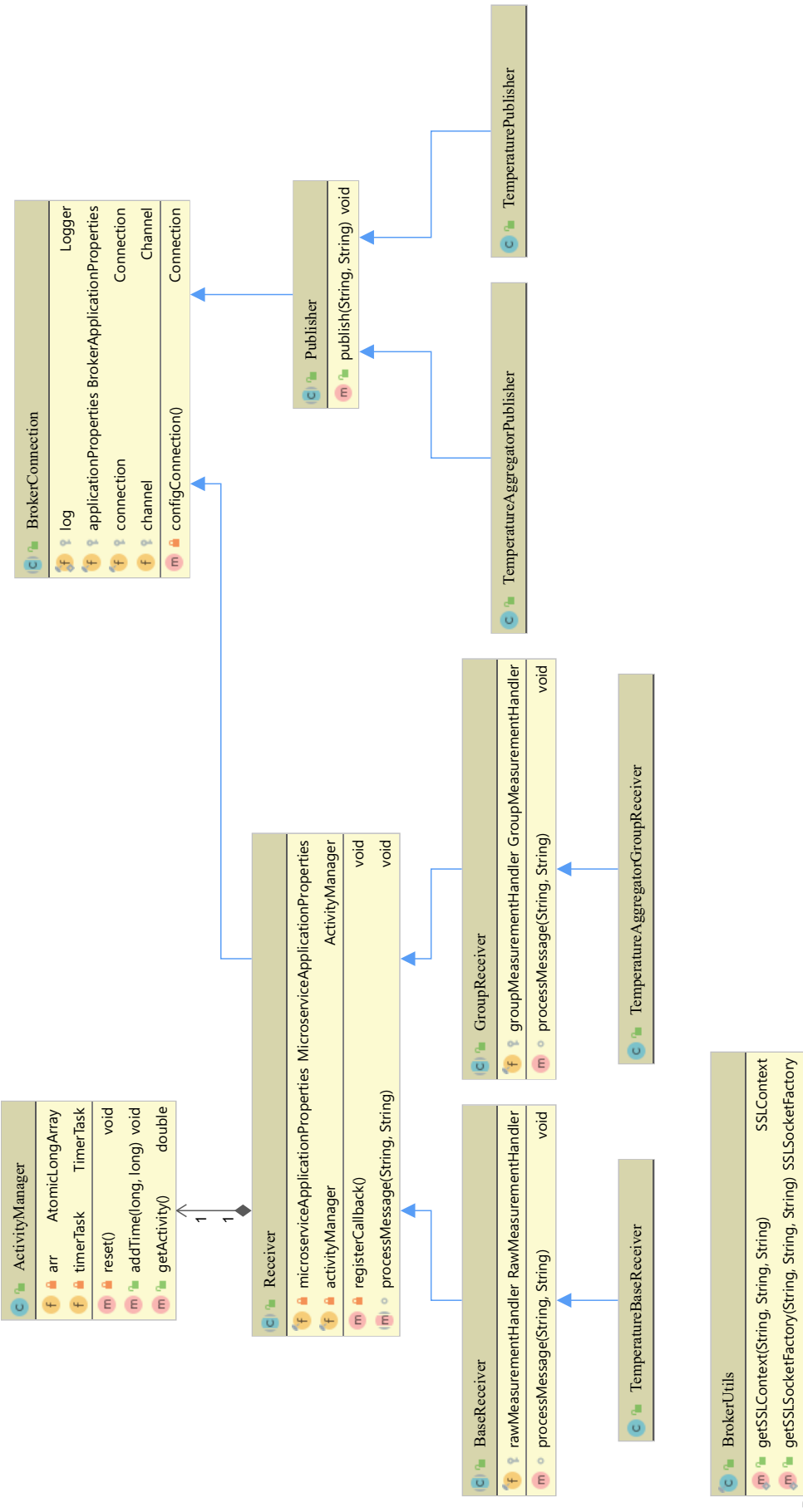


Abbildung B.6: Klassendiagramm Broker

B.11 Klassendiagramm Handler

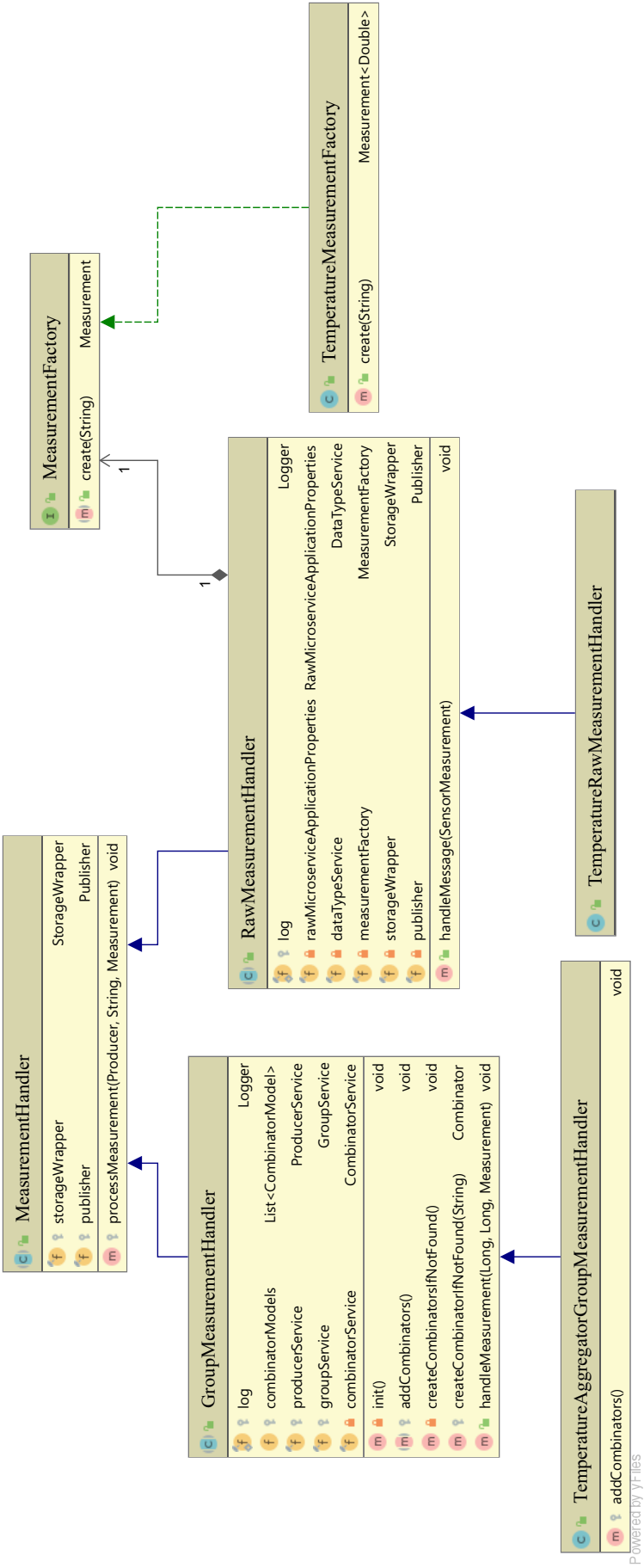


Abbildung B.7: Klassendiagramm Handler

B.12 Klassendiagramm Storage

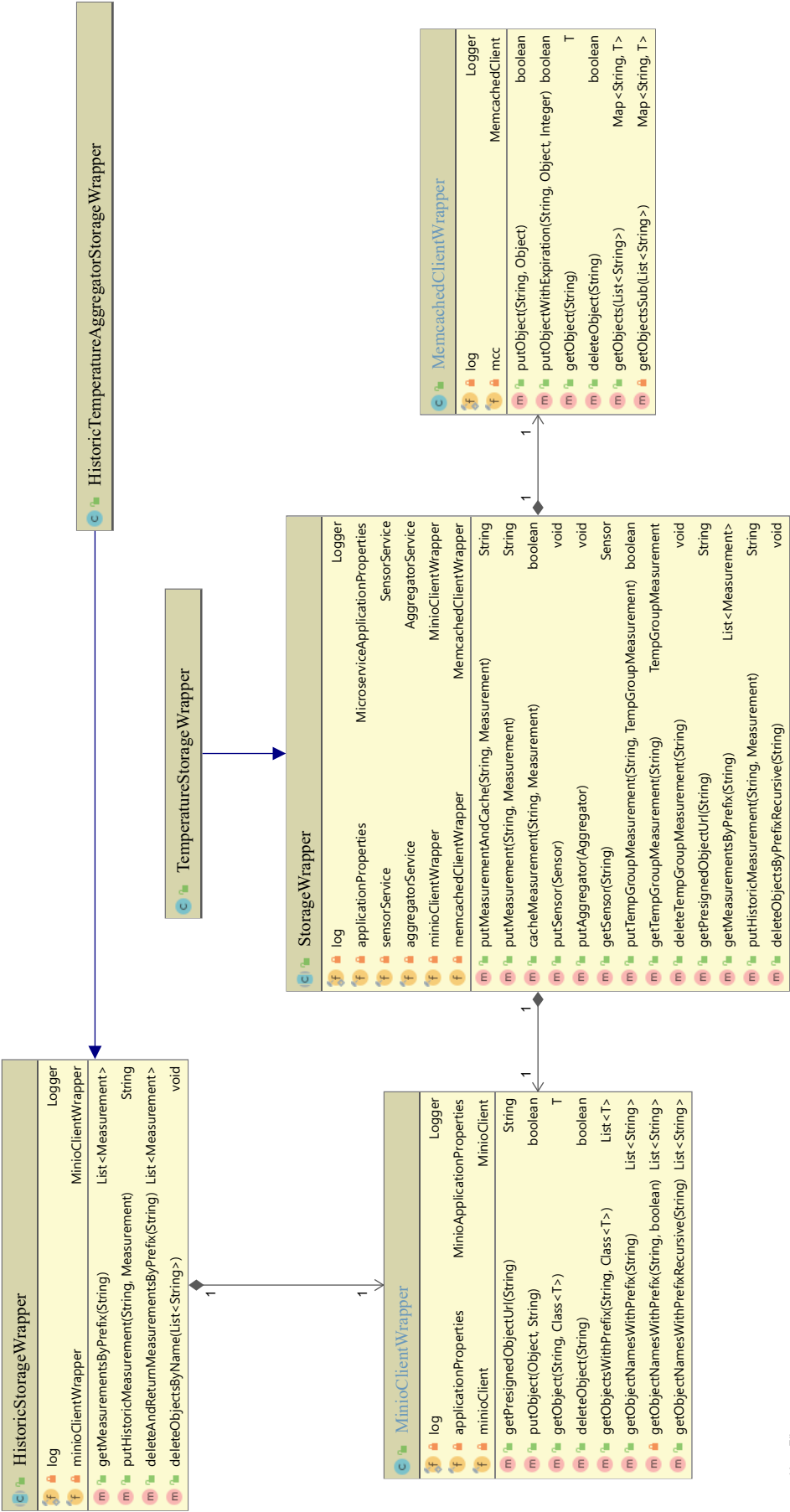


Abbildung B.8: Klassendiagramm Storage

B.13 Klassendiagramm Exporter

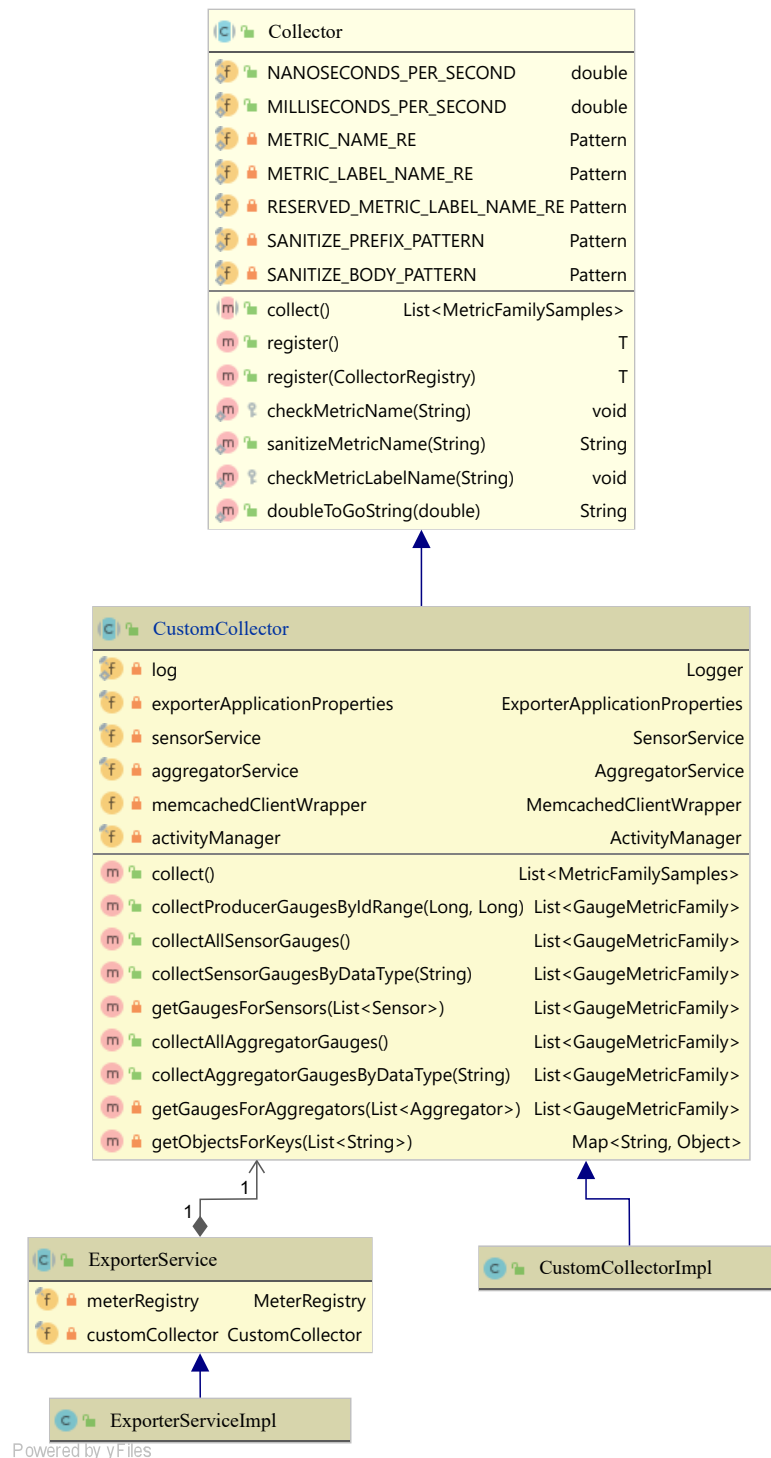


Abbildung B.9: Klassendiagramm Exporter

B.14 Prometheus Metriken

```

1 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor3",location_name="Antarctica",location_x="-78.2",
  location_y="16.4",} -49.87
2 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor5",location_name="New York City",location_x="40.7",
  location_y="-74.0",} 25.08
3 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor1",location_name="Saarbruecken",location_x="49.2",
  location_y="6.9",} 26.9
4 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor4",location_name="Rome",location_x="41.9",
  location_y="12.5",} 26.82
5 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor2",location_name="Moscow",location_x="55.6",
  location_y="36.8",} 15.26
6 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor6",location_name="Tokyo",location_x="35.7",
  location_y="139.7",} 20.12
7 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor7",location_name="Homburg",location_x="49.3",
  location_y="7.3",} 26.06
8 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor8",location_name="Merzig",location_x="49.5",
  location_y="6.6",} 27.09
9 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor9",location_name="Neunkirchen",location_x="49.4",
  location_y="7.2",} 25.62
10 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor10",location_name="Saarlouis",location_x="49.3",
  location_y="6.8",} 27.1
11 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor11",location_name="St.Wendel",location_x="49.5",
  location_y="7.2",} 25.8
12 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor14",location_name="Berlin",location_x="52.5",
  location_y="13.4",} 25.51
13 sensor_measurement{datatype="temperature",sensor_name="data/aggregator/
  temperature/sensor12",location_name="Stuttgart",location_x="48.8",
  location_y="9.2",} 26.25
14 ...
15 sensor_measurement{datatype="humidity",sensor_name="data/aggregator/
  humidity/sensor3",location_name="Antarctica",location_x="-78.2",
  location_y="16.4",} 95.0
16 sensor_measurement{datatype="humidity",sensor_name="data/aggregator/
  humidity/sensor5",location_name="New York City",location_x="40.7",
  location_y="-74.0",} 68.0
17 sensor_measurement{datatype="humidity",sensor_name="data/aggregator/
  humidity/sensor1",location_name="Saarbruecken",location_x="49.2",
  location_y="6.9",} 81.0
18 sensor_measurement{datatype="humidity",sensor_name="data/aggregator/
  humidity/sensor4",location_name="Rome",location_x="41.9",location_y=
  "12.5",} 60.0
19 sensor_measurement{datatype="humidity",sensor_name="data/aggregator/
  humidity/sensor2",location_name="Moscow",location_x="55.6",
  location_y="36.8",} 48.0
20 sensor_measurement{datatype="humidity",sensor_name="data/aggregator/
  humidity/sensor6",location_name="Tokyo",location_x="35.7",location_y
  ="139.7",} 90.0
21 ...
22 sensor_measurement{datatype="water",sensor_name="data/aggregator/water/

```

B Weiterführende Informationen zur Implementierung

```
    sensor3",location_name="Antarctica",location_x="-78.2",location_y="
    16.4",} 0.0
23 sensor_measurement{datatype="water",sensor_name="data/aggregator/water/
    sensor5",location_name="New York City",location_x="40.7",location_y="
    -74.0",} 0.0
24 sensor_measurement{datatype="water",sensor_name="data/aggregator/water/
    sensor1",location_name="Saarbruecken",location_x="49.2",location_y="
    6.9",} 1.0
25 sensor_measurement{datatype="water",sensor_name="data/aggregator/water/
    sensor4",location_name="Rome",location_x="41.9",location_y="12.5",}
    0.0
26 sensor_measurement{datatype="water",sensor_name="data/aggregator/water/
    sensor2",location_name="Moscow",location_x="55.6",location_y="36.8"
    ,} 1.0
27 sensor_measurement{datatype="water",sensor_name="data/aggregator/water/
    sensor6",location_name="Tokyo",location_x="35.7",location_y="139.7"
    ,} 0.0
28 ...
29 group_measurement{group="dewpoint-group-1",datatype="dewpoint",
    combinator="dewpoint-combinator",location_name="Saarbruecken",
    location_x="49.2",location_y="6.9",} 12.92
30 group_measurement{group="dewpoint-group-2",datatype="dewpoint",
    combinator="dewpoint-combinator",location_name="Moscow",location_x="
    55.6",location_y="36.8",} 14.38
31 group_measurement{group="dewpoint-group-3",datatype="dewpoint",
    combinator="dewpoint-combinator",location_name="Antarctica",
    location_x="-78.2",location_y="16.4",} -49.07
32 group_measurement{group="dewpoint-group-saarland-avg",datatype="dewpoint
    ",combinator="dewpoint-combinator",location_name="Saarland",
    location_x="null",location_y="null",} 12.49
33 group_measurement{group="saarland-temperature",datatype="temperature",
    combinator="double-avg",location_name="Saarland",location_x="null",
    location_y="null",} 15.94
34 group_measurement{group="saarland-temperature",datatype="temperature",
    combinator="double-max",location_name="Saarland",location_x="null",
    location_y="null",} 16.31
35 group_measurement{group="saarland-temperature",datatype="temperature",
    combinator="double-min",location_name="Saarland",location_x="null",
    location_y="null",} 15.35
36 group_measurement{group="saarland-humidity",datatype="humidity",
    combinator="double-avg",location_name="Saarland",location_x="null",
    location_y="null",} 80.0
37 group_measurement{group="saarland-humidity",datatype="humidity",
    combinator="double-min",location_name="Saarland",location_x="null",
    location_y="null",} 74.0
38 group_measurement{group="saarland-humidity",datatype="humidity",
    combinator="double-max",location_name="Saarland",location_x="null",
    location_y="null",} 92.0
39 group_measurement{group="saarland-water",datatype="water-sum",combinator
    ="true-count",location_name="Saarland",location_x="null",location_y=
    "null",} 2.0
40 ...
```

Listing B.6: Prometheus Metriken

B.15 Grafana Dashboards

B.15.1 Übersicht Microservices



Abbildung B.10: Dashboard Microservices Monitoring Teil 1

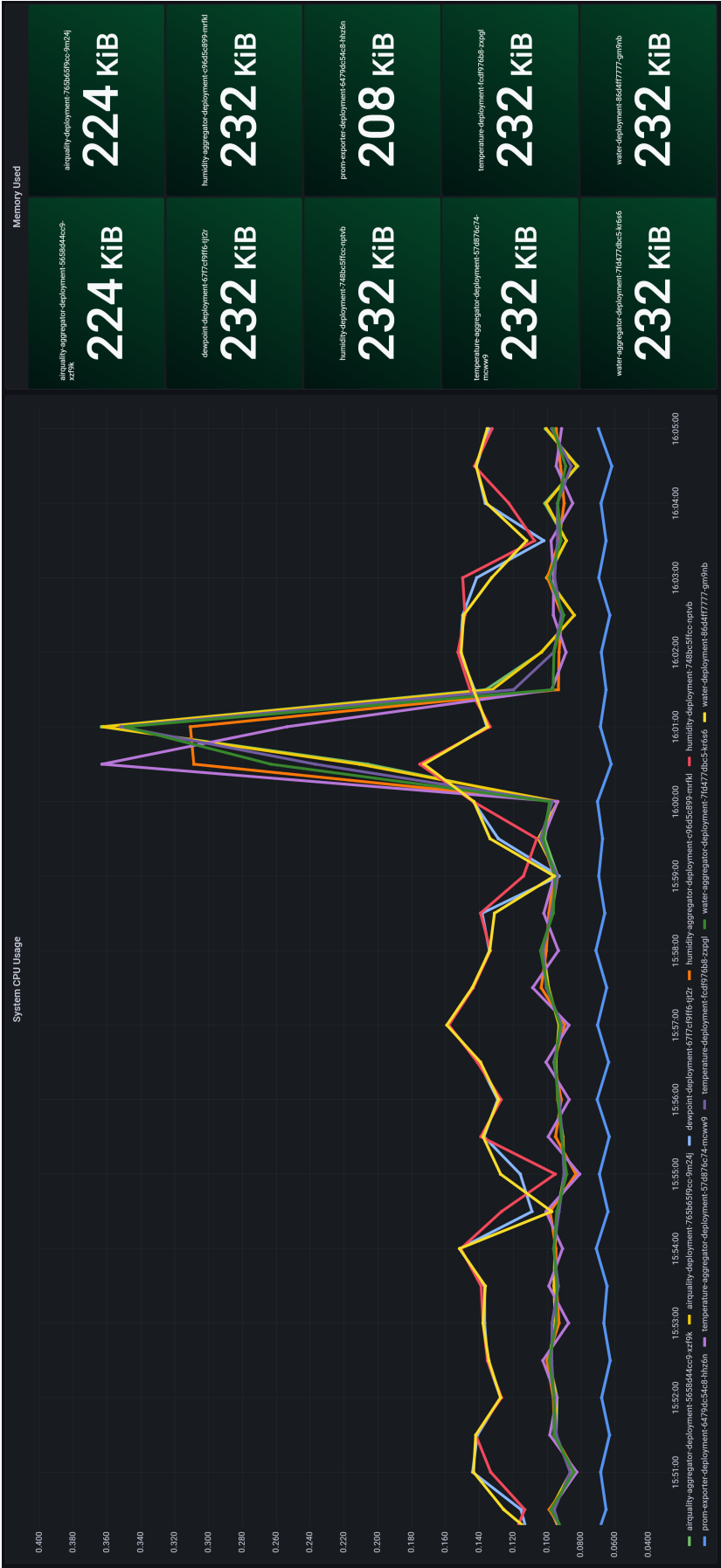


Abbildung B.11: Dashboard Microservices Monitoring Teil 2

B.15.2 RabbitMQ Dashboard

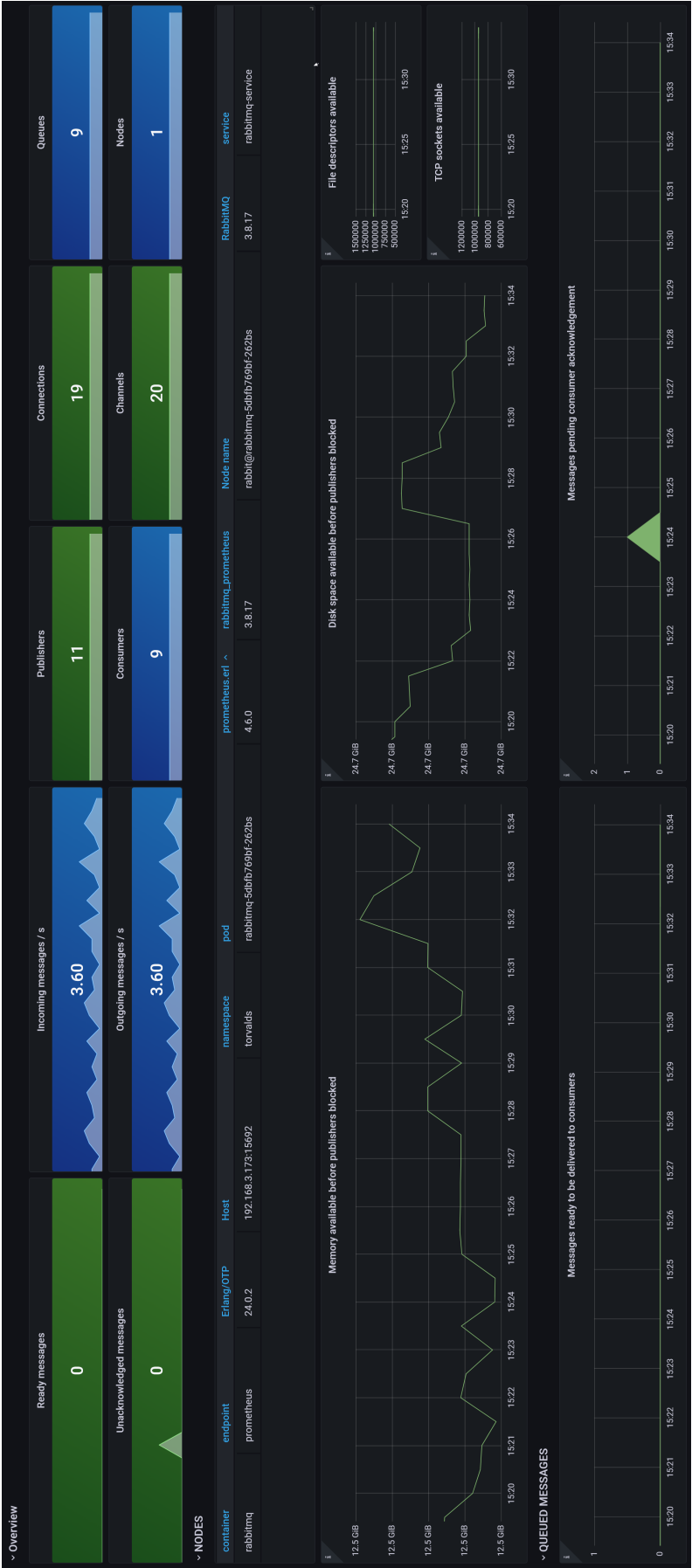


Abbildung B.12: Dashboard RabbitMQ Monitoring

B.15.3 Übersicht Messwerte

B Weiterführende Informationen zur Implementierung

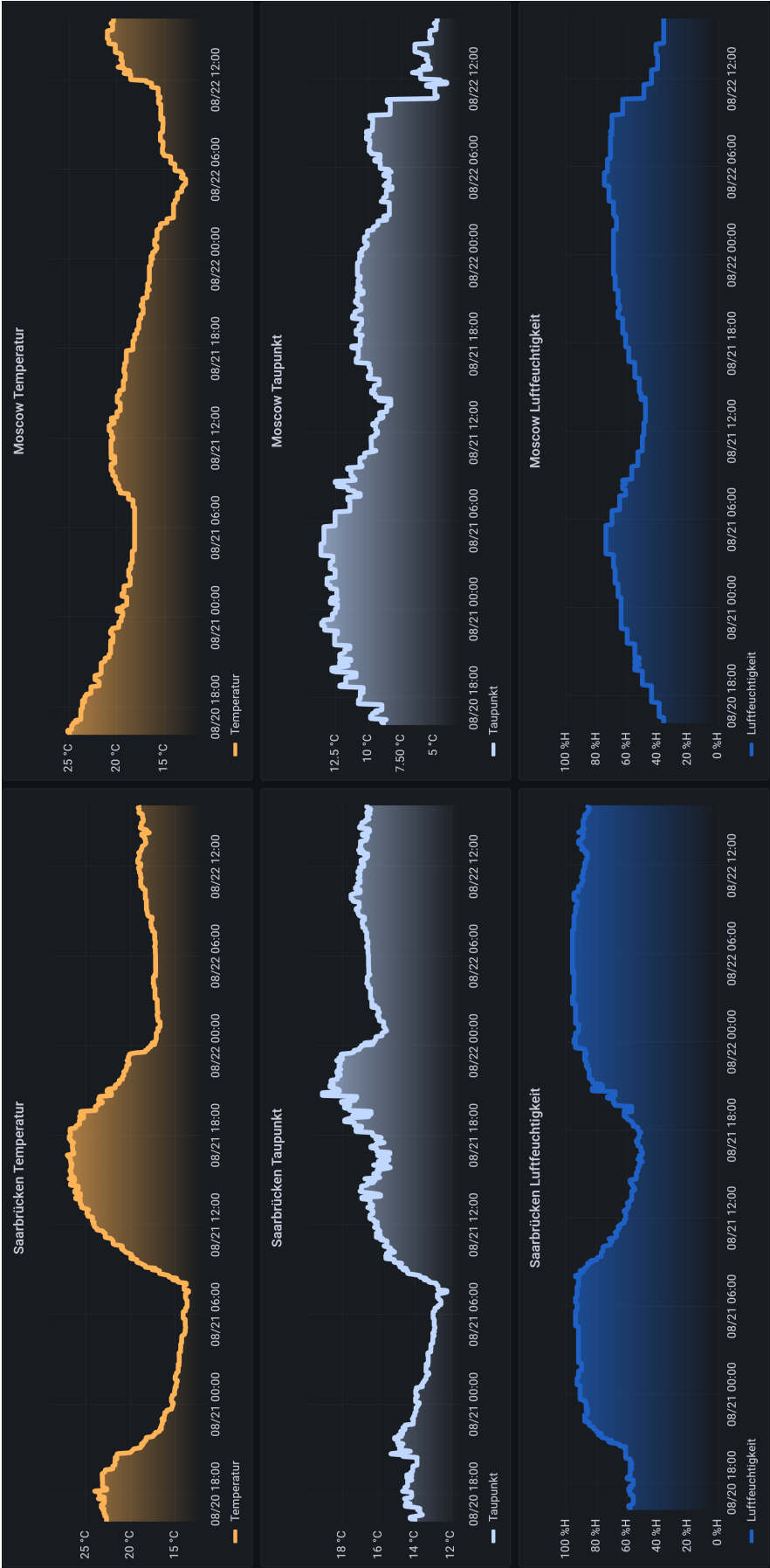


Abbildung B.13: Dashboard Sensor Metriken Rohdaten

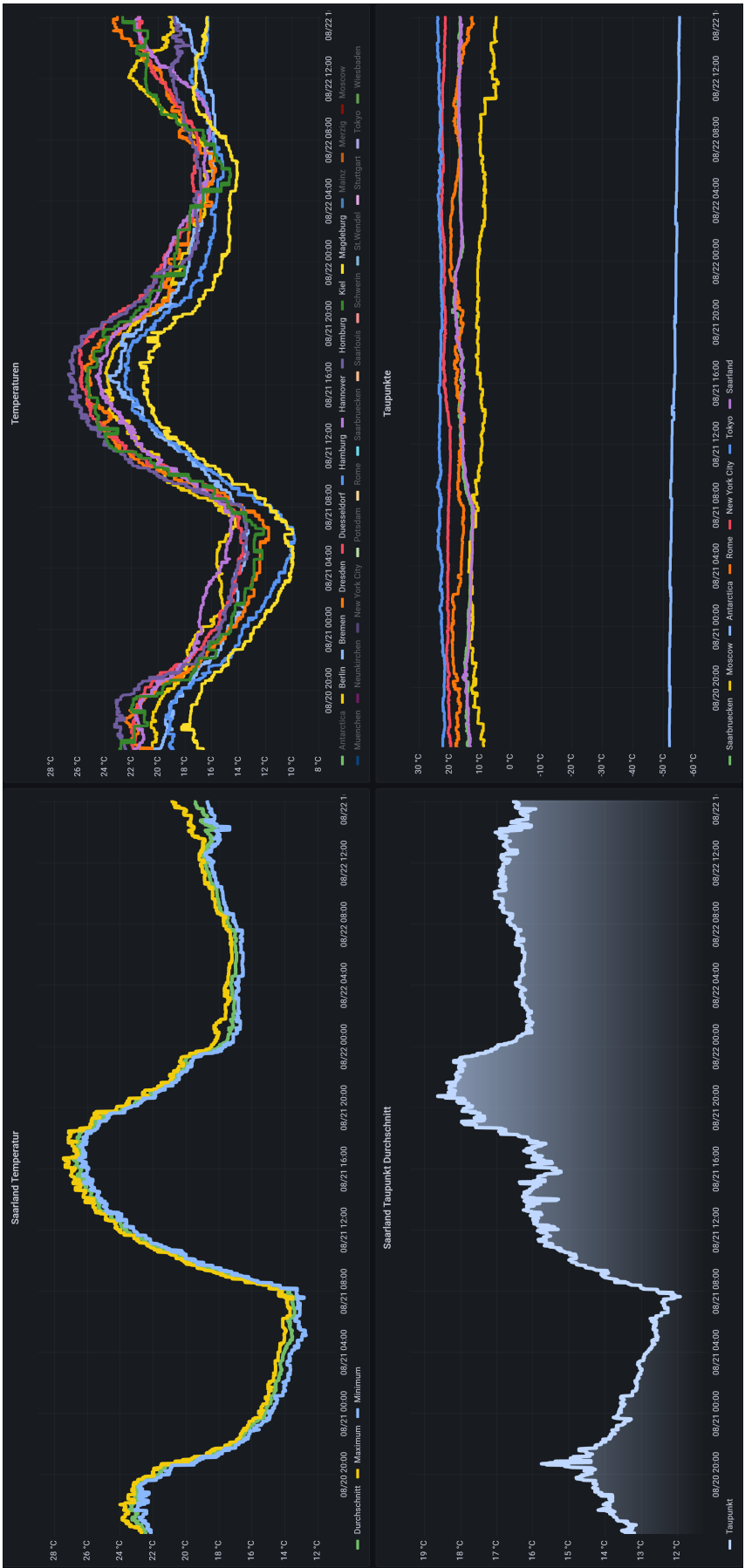


Abbildung B.14: Dashboard Sensor Metriken Aggregate

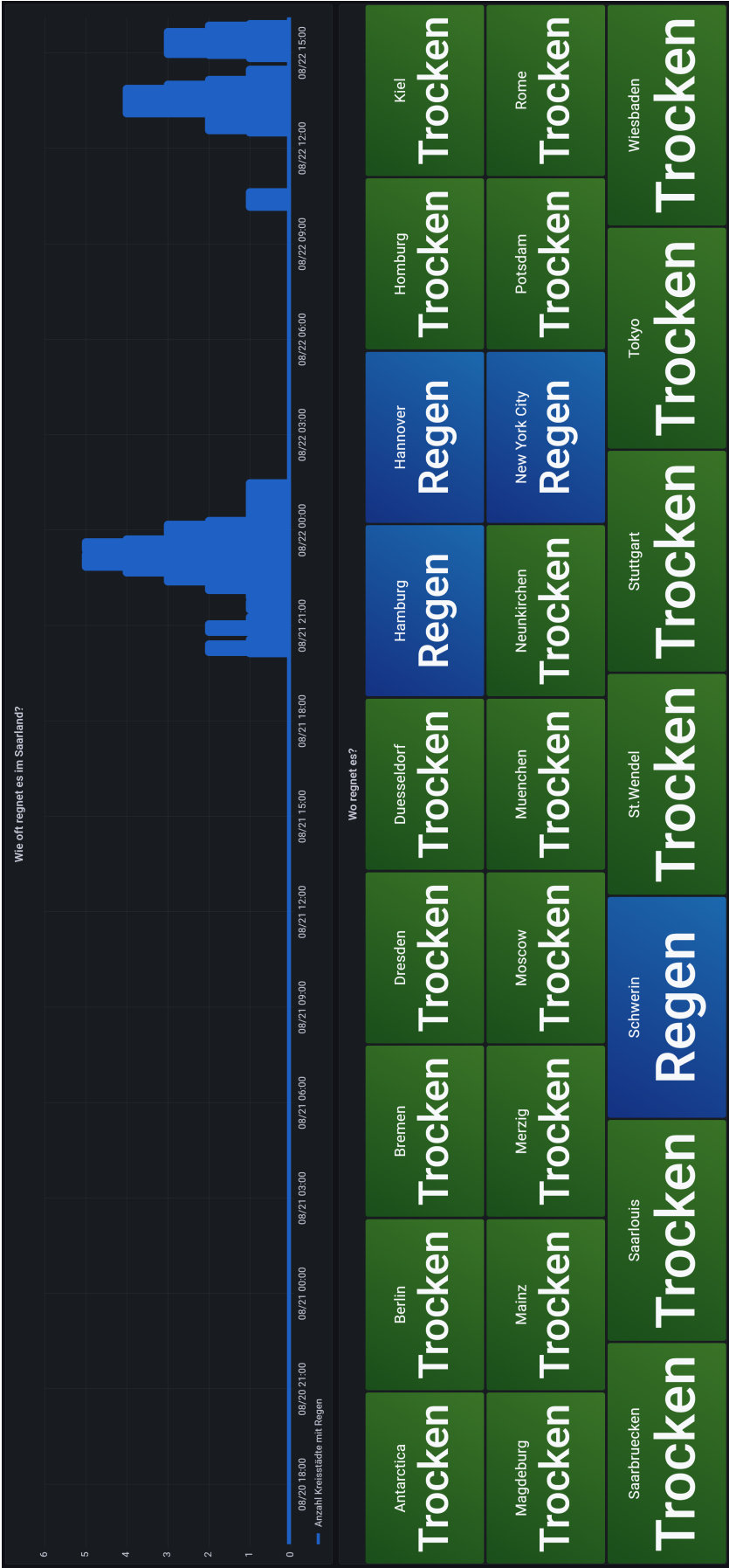


Abbildung B.15: Dashboard Sensor Metriken Wassermessungen und Aggregate

B.16 Deployment RabbitMQ Testumgebung

```

1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: rabbitmq
5    labels:
6      name: rabbitmq
7  data:
8    rabbitmq.conf: |-
9      loopback_users.guest = false
10     log.console = true
11     log.console.level = debug
12     log.exchange = true
13     log.exchange.level = debug
14     prometheus.return_per_object_metrics = true
15   enabled_plugins: |-
16     [rabbitmq_management, rabbitmq_prometheus, rabbitmq_tracing,
17       rabbitmq_mqtt, rabbitmq_event_exchange].
18 ---
19 apiVersion: apps/v1
20 kind: Deployment
21 metadata:
22   name: rabbitmq
23   labels:
24     app: rabbitmq
25 spec:
26   replicas: 1
27   selector:
28     matchLabels:
29       app: rabbitmq
30   template:
31     metadata:
32       labels:
33         app: rabbitmq
34     spec:
35       containers:
36       - name: rabbitmq
37         image: user/rabbitmq-mqtt:latest
38         imagePullPolicy: Always
39         ports:
40         - containerPort: 15672
41           name: http
42         - containerPort: 5672
43           name: amqp
44         - containerPort: 15692
45           name: prometheus
46         - containerPort: 1883
47           name: mqtt
48         volumeMounts:
49         - name: rabbitmq-config-map
50           mountPath: /etc/rabbitmq/
51     volumes:
52     - name: rabbitmq-config-map
53       configMap:
54         name: rabbitmq
55 ---
56 apiVersion: v1
57 kind: Service
58 metadata:
59   name: rabbitmq-service
60   labels:

```

```
60     app: rabbitmq
61 spec:
62   type: NodePort
63   selector:
64     app: rabbitmq
65   ports:
66     - port: 15672
67       name: http
68       nodePort: 31600
69     - port: 5672
70       name: amqp
71       nodePort: 31500
72     - port: 15692
73       name: prometheus
74     - port: 1883
75       name: mqtt
76       nodePort: 31000
77 ---
78 apiVersion: monitoring.coreos.com/v1
79 kind: ServiceMonitor
80 metadata:
81   name: rabbitmq-service-monitor
82   labels:
83     release: my-prom
84 spec:
85   selector:
86     matchLabels:
87       app: rabbitmq
88   endpoints:
89     - port: prometheus
```

Listing B.7: Deployment RabbitMQ Testumgebung

C Weiterführende Informationen zur Evaluation

C.1 Benchmarking Microservice Deployment

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: benchmarking-ms
5  spec:
6    volumes:
7      - name: result-storage
8        persistentVolumeClaim:
9          claimName: result-storage-claim
10   containers:
11     - name: benchmarking-ms
12       image: user/benchmarking-ms
13       envFrom:
14         - configMapRef:
15             name: general-config
16         - secretRef:
17             name: general-secrets
18       resources:
19         requests:
20           memory: "512Mi"
21           cpu: "1000m"
22         limits:
23           memory: "512Mi"
24           cpu: "1000m"
25         volumeMounts:
26           - mountPath: "/out"
27             name: result-storage
28       restartPolicy: Never
29 ---
30 apiVersion: v1
31 kind: PersistentVolumeClaim
32 metadata:
33   name: result-storage-claim
34 spec:
35   storageClassName: glusterfs
36   accessModes:
37     - ReadWriteOnce
38   resources:
39     requests:
40       storage: 10Mi
```

Listing C.1: Benchmarking Microservice Deployment

C.2 Ergebnisse Benchmarking Rohdatenverarbeitung

C.2.1 Auszug JMH-Ergebnisdatei

```
1  [
2  {
3    "jmhVersion" : "1.22",
4    "benchmark" : "de.htw.saar.smartcity.aggregator.benchmarking.
      MicroserviceBenchmark.executeBenchmark",
5    "mode" : "avgt",
6    "threads" : 1,
7    "forks" : 0,
8    "jvm" : "/opt/openjdk-14/bin/java",
9    "jvmArgs" : [
10     "-server"
11   ],
12   "jdkVersion" : "14-ea",
13   "vmName" : "OpenJDK 64-Bit Server VM",
14   "vmVersion" : "14-ea+33",
15   "warmupIterations" : 2,
16   "warmupTime" : "10 s",
17   "warmupBatchSize" : 1,
18   "measurementIterations" : 10,
19   "measurementTime" : "10 s",
20   "measurementBatchSize" : 1,
21   "params" : {
22     "sizeInKB" : "1"
23   },
24   "primaryMetric" : {
25     "score" : 38.15213796006326,
26     "scoreError" : 1.7087374474397616,
27     "scoreConfidence" : [
28       36.4434005126235,
29       39.860875407503016
30     ],
31     "scorePercentiles" : {
32       "0.0" : 35.97635158992806,
33       "50.0" : 38.122120176806085,
34       "90.0" : 39.4283813015748,
35       "95.0" : 39.43420296456693,
36       "99.0" : 39.43420296456693,
37       "99.9" : 39.43420296456693,
38       "99.99" : 39.43420296456693,
39       "99.999" : 39.43420296456693,
40       "99.9999" : 39.43420296456693,
41       "100.0" : 39.43420296456693
42     },
43     "scoreUnit" : "ms/op",
44     "rawData" : [
45       [
46         38.09953327756654,
47         39.43420296456693,
48         39.096326101167314,
49         36.911951590405906,
50         39.02965028015564,
51         37.89380970075758,
52         38.14470707604563,
53         39.37598633464567,
54         37.558860685393256,
55         35.97635158992806
56       ]
57     ]
58   }
59 ]
```

```

58 }
59 },
60 {
61   "jmhVersion" : "1.22",
62   "benchmark" : "de.htw.saar.smartcity.aggregator.benchmarking.
        MicroserviceBenchmark.executeBenchmark",
63   "mode" : "avgt",
64   "threads" : 1,
65   "forks" : 0,
66   "jvm" : "/opt/openjdk-14/bin/java",
67   "jvmArgs" : [
68     "-server"
69   ],
70   "jdkVersion" : "14-ea",
71   "vmName" : "OpenJDK 64-Bit Server VM",
72   "vmVersion" : "14-ea+33",
73   "warmupIterations" : 2,
74   "warmupTime" : "10 s",
75   "warmupBatchSize" : 1,
76   "measurementIterations" : 10,
77   "measurementTime" : "10 s",
78   "measurementBatchSize" : 1,
79   "params" : {
80     "sizeInKB" : "10"
81   },
82   "primaryMetric" : {
83     "score" : 38.25928130046481,
84     "scoreError" : 1.2459724154441016,
85     "scoreConfidence" : [
86       37.013308885020706,
87       39.50525371590891
88     ],
89     "scorePercentiles" : {
90       "0.0" : 37.2076732267658,
91       "50.0" : 37.95516979166666,
92       "90.0" : 39.533009758823255,
93       "95.0" : 39.542158490118574,
94       "99.0" : 39.542158490118574,
95       "99.9" : 39.542158490118574,
96       "99.99" : 39.542158490118574,
97       "99.999" : 39.542158490118574,
98       "99.9999" : 39.542158490118574,
99       "100.0" : 39.542158490118574
100     },
101     "scoreUnit" : "ms/op",
102     "rawData" : [
103       [
104         38.2009409351145,
105         37.57188270411985,
106         37.2076732267658,
107         39.542158490118574,
108         37.741280550943394,
109         37.93710971212121,
110         39.450671177165354,
111         37.97322987121212,
112         37.80906263396226,
113         39.158803703125
114       ]
115     ]
116   },
117 },
118 ...

```

```

119 {
120   "jmhVersion" : "1.22",
121   "benchmark" : "de.htw.saar.smartcity.aggregator.benchmarking.
      MicroserviceBenchmark.executeBenchmark",
122   "mode" : "avgt",
123   "threads" : 1,
124   "forks" : 0,
125   "jvm" : "/opt/openjdk-14/bin/java",
126   "jvmArgs" : [
127     "-server"
128   ],
129   "jdkVersion" : "14-ea",
130   "vmName" : "OpenJDK 64-Bit Server VM",
131   "vmVersion" : "14-ea+33",
132   "warmupIterations" : 2,
133   "warmupTime" : "10 s",
134   "warmupBatchSize" : 1,
135   "measurementIterations" : 10,
136   "measurementTime" : "10 s",
137   "measurementBatchSize" : 1,
138   "params" : {
139     "sizeInKB" : "1000"
140   },
141   "primaryMetric" : {
142     "score" : 65.65565993880003,
143     "scoreError" : 2.3459856622591397,
144     "scoreConfidence" : [
145       63.30967427654089,
146       68.00164560105917
147     ],
148     "scorePercentiles" : {
149       "0.0" : 64.02055177707007,
150       "50.0" : 65.27980494505985,
151       "90.0" : 68.70355336904477,
152       "95.0" : 68.84115232876712,
153       "99.0" : 68.84115232876712,
154       "99.9" : 68.84115232876712,
155       "99.99" : 68.84115232876712,
156       "99.999" : 68.84115232876712,
157       "99.9999" : 68.84115232876712,
158       "100.0" : 68.84115232876712
159     },
160     "scoreUnit" : "ms/op",
161     "rawData" : [
162       68.84115232876712,
163       64.96369381168832,
164       67.46516273154363,
165       64.75043890967741,
166       64.28008078205129,
167       64.36565663461539,
168       66.4932945629139,
169       65.59591607843137,
170       64.02055177707007,
171       65.78065177124184
172     ]
173   },
174 }
175 }
176 }
177 ]

```

Listing C.2: Auszug JMH-Ergebnisdatei

C.2.2 Durchschnittswerte über mehrere Iterationen

Tabelle C.1: Durchschnittliche Verarbeitungszeit in Relation Größe der Messwerte

Größe Datensatz in KB	Verarbeitungszeit in ms/op
1	38,15213796
10	38,2592813
20	38,46945207
30	38,7383948
40	39,21032651
50	39,49136057
60	40,04654974
70	40,39722052
80	41,03778844
90	40,96642274
100	41,33199793
200	44,98934113
300	48,05080696
400	50,33477504
500	52,80539687
750	59,52155169
1000	65,65565994

C.3 Ergebnisse Benchmarking Prometheus Exporter

Tabelle C.2: Durchschnittliche Verarbeitungszeit in Relation zur Anzahl an Datensätzen

Anzahl Datensätze	Verarbeitungszeit in ms/op
1	1,413577799
50	2,068314123
100	2,796162584
200	4,142339093
300	5,497531946
400	6,869952783
500	8,078926773
750	11,34999075
1000	14,75934873
2500	35,30927926
5000	72,06826925
10000	151,4025463
20000	329,2121068
30000	552,4815047
40000	805,3213282
50000	1113,429745
60000	1373,59728
70000	1598,233674
80000	2184,714442
90000	2351,789398
100000	2496,970048
125000	3403,058247
150000	5063,299098
175000	6029,007074
200000	7260,253242
225000	8505,157518
250000	10561,60832

C.4 Ergebnisse Benchmarking Prometheus Exporter Optimierung

Tabelle C.3: Durchschnittliche Verarbeitungszeit in Relation zur Anzahl an Datensätzen

Anzahl Datensätze	Verarbeitungszeit in ms/op
1	1,412063585
100	2,843512148
1000	14,76321847
5000	57,81408434
10000	122,2719786
25000	353,859376
50000	856,5415796
75000	1478,2302
100000	2088,595749
125000	2685,677641
150000	3638,564738
175000	4374,187944
200000	5942,08363
225000	6407,789569
250000	7281,381586
275000	7913,653575
300000	8360,141476
325000	9704,128409
350000	10520,32916
375000	10816,98754
400000	11443,71789
425000	13604,3905
450000	14947,72928
475000	15507,44021
500000	16313,65081

Kolophon

Dieses Dokument wurde mit der L^AT_EX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 1.0). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt