

# Giving Structure to Open Data in the JValue ODS

MASTER THESIS

Alexander Mahler

Eingereicht am 1. September 2021



Friedrich-Alexander-Universität Erlangen-Nürnberg  
Technische Fakultät, Department Informatik  
Professur für Open-Source-Software

Betreuer:  
M.Sc. Georg Schwarz  
Prof. Dr. Dirk Riehle, M.B.A.



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
TECHNISCHE FAKULTÄT



# Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

---

Erlangen, 1. September 2021

# License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see <https://creativecommons.org/licenses/by/4.0/>

---

Erlangen, 1. September 2021



# Abstract

Nowadays the internet provides a lot of open data for public use. Those can be written in various data types and cover plenty of subjects. Because of that the absence of a standard results into the main problem. Every provider can decide for himself how the data is constructed.

The JValue project is dedicated to this problem and aims to be the central point where those open data are gathered and optimized. Currently the JValue Open-Data-Service (ODS) provides the extraction, transformation and retrieving of open data supporting numerous protocols and data formats.

However until now there is only a very generic interface for the retrieval of those open data since the system currently ignores any data structure. In addition to that any provider can alter their data structure and upload it after the adjustment process, since they are not bound to any restrictions. This can lead to major restrictions or even the loss of the data gathering process.

To counteract this behavior a process shall be introduced, which allows the ODS to structure those open data. Furthermore a schema recommendation for the data should be generated, which then will be the foundation of the remaining data gathering process.

As a consequence of the introduced data schema there is now a possibility to also derive fitting database tables from those schema. This tables should be created and filled dynamically and provide the user a fully and easy accessible interface. As an implication of the persistent structured data, the earlier mentioned problem of frequently changing data structures can now be easily solved. The schema can be used to validate those imported and transformed data. By also adding a corresponding visual state to those data configurations, the user will be able to react up on changed data structures.



# Zusammenfassung

Im Internet finden sich heutzutage sämtliche für die Öffentlichkeit bereitgestellten Daten. Diese werden in den verschiedensten Datentypen angeboten und decken eine immense Menge an Themen und Bereiche ab. Das zentrale Problem der offenen Daten jedoch ist das Fehlen eines Standards. Es ist den Anbietern freigestellt nach Belieben ihre Daten darzustellen.

Das JValue Projekt widmet sich diesem Problem und möchte eine zentrale Stelle schaffen, an welcher solche offenen Daten gesammelt und aufbereitet werden können. Derzeit erlaubt der JValue ODS die Extraktion, die Transformation und das Laden (ETL) von offenen Daten mit unterschiedlichen Protokollen und Datenformaten. Allerdings kann bisher nur eine sehr generische Schnittstelle zum Laden der Daten bereitgestellt werden, da das System die Daten unabhängig ihrer Struktur behandelt. Des Weiteren können Anbieter von offenen Daten die Struktur ihrer veröffentlichten Daten jederzeit ändern, was zur Einschränkung oder gar Verlust des Datengewinnungsprozesses führen kann.

Um dem entgegenzuwirken soll ein Prozess eingeführt werden, mit welchem der ODS die offenen Daten in Struktur betrachtet. Unter Anderem soll ein Schema-Vorschlag zu den jeweiligen Daten generiert werden, welcher den Grundbaustein für den restlichen ETL-Prozess legt.

Als Konsequenz der eingeführten Datenschemata können dynamisch passende Tabellen erstellt und befüllt werden, um letztendlich eine optimierte Lade-Schnittstelle zur Verfügung zu stellen. Des Weiteren erlaubt der Abgleich von extrahierten/importierten Daten mit dem erwarteten Schema die Indikation des Zustands der einzelnen ETL Schritte. Änderungen im Datenformat oder andere Schnittstellenänderungen auf Daten-Anbieterseite können dadurch in Form eines Ampel-Systems visualisiert werden. Dabei kann der Zustand der importierten und transformierten Daten überwacht und angezeigt werden.





# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Anforderungen</b>	<b>3</b>
2.1	Zielsetzung dieser Arbeit . . . . .	3
2.2	Beschaffung der Anforderungen . . . . .	3
2.3	Funktionelle Anforderungen . . . . .	4
2.3.1	Anreicherung von Datenquellen mit einem Schema . . . . .	4
2.3.2	Anreicherung von Pipelines mit einem Schema . . . . .	5
2.3.3	Überprüfung der Schemakonformität . . . . .	5
2.3.4	Optimierte Query API . . . . .	5
2.4	Nicht-funktionelle Anforderungen . . . . .	6
<b>3</b>	<b>Architektur und Design</b>	<b>7</b>
3.1	Bestehende Architektur . . . . .	7
3.1.1	Microservices . . . . .	7
3.1.2	Service: UI . . . . .	8
3.1.3	Service: Adapter . . . . .	8
3.1.4	Service: Pipeline . . . . .	8
3.1.5	Service: Storage . . . . .	8
3.1.6	Message-Broker: AMQP . . . . .	8
3.2	Vergleich von strukturierten und unstrukturierten Daten . . . . .	9
3.2.1	Strukturierte Daten mittels SQL . . . . .	9
3.2.2	Unstrukturierte Daten mittels NoSQL . . . . .	9
3.2.3	Projektion auf den Anwendungsfall ODS . . . . .	10
3.2.4	Auswahl der passenden Datenbank . . . . .	10
3.3	Vergleich etablierter Schemarepräsentationen . . . . .	11
3.3.1	JsonSchema . . . . .	11
3.3.2	XML Schema Definition . . . . .	12
3.3.3	Vergleich beider Tools . . . . .	12
3.4	Algorithmus für den Schemavorschlag . . . . .	13
3.5	Design der Nutzerinteraktion . . . . .	15
3.5.1	Interaktion für Datenquellen . . . . .	15

3.5.2	Interaktion für Pipelines . . . . .	16
3.5.3	Healthchecker . . . . .	16
3.5.4	Interaktion mit der Bereitstellung der Daten . . . . .	17
3.6	Verantwortlichkeiten der Services/Komponenten . . . . .	17
3.6.1	Service: Schema . . . . .	17
3.6.2	Open-Data-Service . . . . .	18
3.6.3	PostGraphile Schnittstelle . . . . .	18
3.7	API Design . . . . .	19
<b>4</b>	<b>Implementierung</b>	<b>21</b>
4.1	Verwendete Sprachen . . . . .	21
4.2	Docker . . . . .	21
4.3	Aufteilung der Arbeitspakete . . . . .	22
4.4	Implementierungsdetails Schema-Service . . . . .	22
4.4.1	API . . . . .	22
4.4.2	Ontology Generierung . . . . .	22
4.4.3	Artefakte der Ontology . . . . .	24
4.4.4	Jsonschema parsing . . . . .	24
4.5	Implementierungsdetails UI . . . . .	27
4.5.1	Datasource . . . . .	27
4.5.2	Pipeline . . . . .	28
4.6	Implementierungsdetails Adapter . . . . .	29
4.6.1	Datasource . . . . .	29
4.6.2	DataImport . . . . .	29
4.6.3	DatasourceManager . . . . .	30
4.6.4	Validator . . . . .	31
4.7	Implementierungsdetails Pipeline . . . . .	32
4.7.1	Datenmodelle . . . . .	32
4.7.2	PipelineTransformationDataRepository . . . . .	32
4.7.3	PipelineTransformationDataManager . . . . .	33
4.7.4	OutboxEventPublisher . . . . .	33
4.7.5	PipelineConfigManager . . . . .	33
4.7.6	Validator . . . . .	33
4.8	Implementierungsdetails Storage . . . . .	34
4.8.1	EventHandler . . . . .	34
4.8.2	PostgresParser . . . . .	34
4.8.3	StorageStructureRepository . . . . .	37
4.8.4	StorageContentRepository . . . . .	38
4.8.5	Postgraphile . . . . .	38
4.9	Probleme und mögliche Lösungen . . . . .	41
4.9.1	Docker . . . . .	41
4.9.2	Windows . . . . .	42
4.9.3	M1 . . . . .	43

<b>5</b>	<b>Evaluierung</b>	<b>45</b>
5.1	Evaluierung des Architektur Designs . . . . .	45
5.2	Evaluierung anhand der Anforderung . . . . .	46
5.2.1	Anreicherung von Datenquellen mit einem Schema . . . . .	46
5.2.2	Anreicherung von Pipelines mit einem Schema . . . . .	47
5.2.3	Überprüfung der Schemakonformität . . . . .	48
5.2.4	Optimierte Query API . . . . .	49
5.2.5	Nicht-funktionelle Anforderungen . . . . .	49
<b>6</b>	<b>Ausblick</b>	<b>51</b>
6.1	Erweiterung des Schemavorschlags . . . . .	51
6.2	Erweiterung durch weitere Schemata . . . . .	51
6.3	Erweiterung durch verschiedene Validatoren . . . . .	52
6.4	Nutzen der Ontology für alternative Zwecke . . . . .	52
Anhang A	Beispiele der Ontologyanwendung . . . . .	53
	<b>Literaturverzeichnis</b>	<b>59</b>



# Abkürzungsverzeichnis

**ODS** Open-Data-Service

**REST** Representational State Transfer

**API** Application Programming Interfaces

**XSD** XML Schema Definition

**XML** Extensible Markup Language

**JSON** JavaScript Object Notation



# 1 Einleitung

Im Internet finden sich heutzutage sämtliche für die Öffentlichkeit bereitgestellten Daten. Diese können von verschiedensten Nutzern wie Universitäten, Unternehmen, Analytikern oder auch Privatpersonen nach Belieben angefragt, analysiert und verarbeitet werden. Die Bandbreite an Diversität ist in diesem Bereich besonders hoch. Nicht nur die unterschiedlichsten Motivationen, sondern auch die Vielfalt der angebotenen Datenformate, wie zum Beispiel JavaScript Object Notation (JSON), Extensible Markup Language (XML) und viele mehr erhöhen die Komplexität dieses Anwendungsfalles. Die Themen und Bereiche, wie zum Beispiel aktuelle Verkehrsdaten, bevorzugte Vornamen, aber auch Wasserstände von Flüssen, die die offenen Daten abdecken, sind praktisch endlos.

Trotz der großen Verfügbarkeit und Freiheit dieser Daten gibt es keine übergeordnete Verwaltung. Das hat bei einer derartig großen Vielfalt zur Folge, dass die angebotenen Daten nicht verschiedener hinsichtlich ihrer Struktur sein könnten. Der Mangel eines Standards macht einen präzisen und einheitlichen Zugriff auf diese im Internet frei verfügbaren Daten praktisch unmöglich. Dieses Verhalten ist zudem auch, wenn man einzelne offene Datenquellen betrachtet, zu finden. Denn wie es im Allgemeinen keinen Standard gibt, so müssen sich auch die einzelnen Datenquellen auf keine feste Struktur festlegen. Das bedeutet, dass eine Schnittstelle vom Nutzer auf eine bestimmte Datenquelle angepasst, auch in binnen weniger Tagen wieder untauglich sein kann.

Das JValue Projekt widmet sich diesem Problem und möchte eine zentrale Stelle schaffen, an welcher solche offenen Daten gesammelt und aufbereitet werden können. Derzeit erlaubt der JValue ODS die Extraktion, die Transformation und das Laden (ETL) von offenen Daten mit unterschiedlichen Protokollen und Datenformaten. Dies behebt nicht das Problem der Abwesenheit eines Standards, dennoch kann der Nutzer hier die Rolle des oben genannten Verwalters übernehmen. Im eigenen Kontext kann der Nutzer so eine Sammlung für sich interessanter Datenquellen erstellen und macht den ersten Schritt in eine einheitliche Schnittstelle. Allerdings kann bisher nur ein sehr generisches Interface zum Laden der Daten bereitgestellt werden, da das System die Daten unabhängig ihrer Struktur behandelt. Des Weiteren besteht immer noch das Problem der unsicheren Struktur

bisherig eingepflegter Daten. Der Nutzer kann zwar nun gebündelt an einem zentralen Ort auf seine Daten zugreifen, ist aber nach wie vor der Strukturänderung durch die Anbieter ausgesetzt. Dies kann im System des Users zu unerwarteten Fehlverhalten oder gar zu Ausfällen gesamter Systeme führen.

Um dem entgegenzuwirken, soll nun ein Prozess eingeführt werden, mit welchem der ODS die offenen Daten in Struktur betrachtet. Dabei ist der Nutzer weiterhin in der Rolle der Beschaffung der Datenquellen und muss diese in den ODS selbstständig einpflegen. Nun soll der Nutzer aber die Wahl haben, ob er seine Daten wie gewohnt unsicher und strukturlos persistiert haben möchte oder die neue Funktion in Anspruch nimmt. Diese soll die Daten aus der Datenquelle hinsichtlich ihrer Struktur analysieren und ein passendes Schema dazu automatisch generieren. Für Nutzer, die ein fundiertes Verständnis im Bereich der Schemadarstellung und Softwareentwicklung haben, aber auch für Neulinge und experimentierfreudige, soll das vorgeschlagene Schema jederzeit vollständig transparent einsehbar und veränderbar sein.

Als Konsequenz der eingeführten Datenschemata können für eben jene ehemals unstrukturierte Daten nun maßgeschneiderte Datenbanktabellen erstellt und zur Verfügung gestellt werden. Dabei kann anhand des generierten Schemas dynamisch und zur Laufzeit eine entsprechende Tabelle erstellt werden und beim Anlegen neuer Daten auch befüllt werden. Dies ermöglicht nun nicht nur einen zentralen Punkt für offene Daten, sondern auch eine optimierte Lade-Schnittstelle zur freien Verfügung für den Nutzer. Durch die genaue Kenntnis der Struktur der Daten und einer Historie, in Form einer Datenbank aus importierten und strukturierten Daten, eröffnet sich eine weitere Schlüsselfunktion. Änderungen im Datenformat oder andere Schnittstellenänderungen auf Daten-Anbieterseite können durch einen Vergleich mit den vorherigen Daten und ihrer definierten Struktur nun sofort aufgedeckt werden und dem Nutzer zum Beispiel in Form eines Ampel-Systems visuell dargestellt werden. Somit kann sich der Nutzer immer sicher über den Zustand seiner zentralen Datenquelle im ODS sein.



## 2 Anforderungen

### 2.1 Zielsetzung dieser Arbeit

Viele Nutzer im kommerziellen, forschenden als auch privaten Bereich arbeiten mit verschiedenen offenen Daten, welche von unterschiedlichen Anbietern im Internet bereitgestellt werden. Diese Datenquellen sind jedoch sehr heterogen und folgen keinem bestimmten Standard, vor allem in Bezug auf ihre Struktur. Hierdurch ist der Nutzer gezwungen, sich bei jeder neuen Datenquelle eine passende Struktur zu erarbeiten. Diese erarbeitete Struktur muss bei einer Nutzung über einen längeren Zeitraum kontinuierlich mit den Daten des Anbieters gegengeprüft werden, um garantieren zu können, dass das System fehlerfrei und mit der Struktur der Datenquellen übereinstimmt.

Diese Arbeit soll zum einen Teil im Rahmen des ODS evaluieren, welche Technologien und Methodiken sich am vielversprechendsten eignen, um den Anwendungsfall hinsichtlich der Performance, Usability und Stabilität zu realisieren. Zum anderen Teil soll sie die Lösung dieser Problematik modellieren und implementieren. Hierbei soll ein semi-automatischer Prozess vom Nutzer angegebene Datenquellen analysieren und diesem dann das Schema zur optionalen Bearbeitung unterbreiten. Nach Bestätigung des Strukturvorschlags soll der Prozess fortgeführt werden, indem über das Schema die Datenquelle verarbeitet wird. Infolgedessen erhält der Nutzer eine auf seinen Anwendungsfall optimierte Query-Schnittstelle, über welche er auf alle Daten der Quelle standardisiert und unkompliziert zugreifen kann. Zur Interaktion mit dem gesamten Prozess wird eine UI bereitgestellt. Des Weiteren soll diese den Nutzer mittels eines sogenannten Healthindikators stets über den aktuellen Zustand und Aktualität des Systems informieren. So ist zu jeder Zeit bekannt, ob das aktuelle Schema mit der Datenquelle und der Query-Schnittstelle kompatibel ist.

### 2.2 Beschaffung der Anforderungen

Die Anforderungen sollen die Applikation in eine Richtung lenken, die dem Anwender den größtmöglichen Nutzen, wie auch Komfort bieten kann. Dazu wurden

zunächst im Rahmen einer Projektarbeit einige Evaluierungsschritte getätigt, um die Möglichkeiten und Einschränkungen des Systems bezüglich der Integration eines Schemas aufzuzeigen. Zu Beginn dieser Arbeit wurden dann verschiedene Nutzer des ODS hinzugezogen, um bekannte Probleme und gewünschte Features zu erfragen. Anschließend wurden die Informationen aus den beiden Prozessen zusammengetragen und sowohl entwicklungsnahe als auch nutzergetreue Anforderungen festgehalten.

### 2.3 Funktionelle Anforderungen

Der folgende Abschnitt zeigt jegliche Anforderungen auf, die die Applikation um gewisse Funktionalitäten erweitern.

#### 2.3.1 Anreicherung von Datenquellen mit einem Schema

1. Evaluierung passender Schemarepräsentationen  
Es sollen verschiedene Repräsentationsarten erarbeitet und evaluiert werden. Dabei sollen diese Schemarepräsentationen hinsichtlich Übersichtlichkeit und Effizienz bei der Darstellung der Daten geprüft werden. Zudem soll diese für den Nutzer leicht zu lesen und schreiben sein.
2. Optionale Eingabemöglichkeit eines Schemas für Datenquellen  
Der Nutzer soll zu jeder Zeit in der Lage sein, ein eigens angefertigtes Schema anzugeben und in die Konfiguration übernehmen zu können.
3. Generierung eines Schema-Vorschlags  
Anhand der übergebenen Datenquelle soll ein entsprechendes Schema erstellt werden und dem Nutzer als Vorschlag dargeboten werden. Hierbei gibt es folgende zwei Alternativen: Qualitative Generierung oder performante Generierung. Siehe 2.4
4. Optionale Veränderung des Vorschlags  
Als optionaler Folgeschritt zu 3 soll der Nutzer das Schema anpassen können. Sollte er dies nicht wünschen, wird das ursprünglich vorgeschlagene Schema verwendet.
5. (optional) Graphische Repräsentation des Schemas  
Je nach Größe der einzelnen Daten kann das vorgeschlagene Schema entsprechend unübersichtlich werden. Damit der Nutzer einen besseren Überblick über das Schema bekommen kann, wird hierzu eine graphische Repräsentation angeboten. Diese kann zum Beispiel als Graph dargestellt werden.

### 2.3.2 Anreicherung von Pipelines mit einem Schema

1. Evaluierung einer passenden Datenbank
2. Optionale Eingabemöglichkeit eines Schemas für Pipelines  
Der Nutzer soll zu jeder Zeit in der Lage sein ein eigens angefertigtes Schema anzugeben und in die Konfiguration übernehmen zu können.
3. Generierung eines Schema-Vorschlags  
Anhand der transformierten Daten soll ein entsprechendes Schema erstellt werden und dem Nutzer als Vorschlag dargeboten werden. Hierbei gibt es folgende zwei Alternativen: Qualitative Generierung oder performante Generierung. Siehe 2.4
4. Optionale Veränderung des Vorschlags  
Als optionaler Folgeschritt zu 3 soll der Nutzer das Schema anpassen können. Sollte er dies nicht wünschen, wird das ursprünglich vorgeschlagene Schema verwendet.
5. Transferierung der Daten in die Datenbank  
Nach erfolgreicher Anreicherung der Pipeline mit einem Schema kann der Transferprozess gestartet werden. Hierbei soll eine entsprechende Tabelle für die transformierten Daten angelegt sein und diese befüllt werden.

### 2.3.3 Überprüfung der Schemakonformität

1. Healthindikator für Datenquellen  
Die Datenquelle ist unabhängig vom ODS. Somit sind Änderungen seitens der Anbieter der Daten jederzeit möglich. Dies soll ständig entgegen geprüft werden und mittels eines Indikators dem Nutzer sichtbar gemacht werden. Es sollen verschiedene Zustände in der Anzeige möglich sein.
2. Healthindikator für Pipelines  
Wie auch für die Datenquellen soll hier ein Indikator zur Verfügung stehen, der die Differenz zwischen den transformierten Daten und des aktuellen Schemas überprüft und anzeigt.

### 2.3.4 Optimierte Query API

1. Query auf Datensätze unterschiedlicher Zeitpunkte  
Die Daten sollen mit der Möglichkeit zur zeitlichen Unterscheidung in die Datenbank transferiert werden. So hat der Nutzer die Möglichkeit auf Daten unterschiedlicher Zeitpunkte zuzugreifen.

### 2. Filtern der Felder

Die Query-API bietet dem Nutzer die Möglichkeit Abfragen speziell für gewünschte Felder zu senden.

### 3. Filtern nach Attributen

## 2.4 Nicht-funktionelle Anforderungen

### 1. Vermeidung von Code-Duplikaten

Der Code soll möglichst frei von Redundanzen sein. Dadurch ist eine gute Wartbarkeit und bessere Übersicht für die Implementierung gesichert.

### 2. Performance für Schemavorschlag

Die Analyse und Generierung des Schemas soll möglichst performant ablaufen. Hierbei soll keine lange Wartezeit für den Nutzer entstehen. Dieser Punkt kann im Konflikt zu 3 stehen.

### 3. Qualität des Schemavorschlags

Die Analyse und Generierung des Schemas soll möglichst qualitativ hochwertig sein. Die Qualität lässt sich anhand der Genauigkeit und Tiefe des Schemas bestimmen. Dieser Punkt kann im Konflikt zu 2 stehen

### 4. Erlernbarkeit

Der Code soll weitestgehend selbsterklärend sein. Hierbei soll es keine weitere Erklärung eines Dritten benötigen, um mit dieser Implementierung zu arbeiten.

# 3 Architektur und Design

## 3.1 Bestehende Architektur

Die Architektur des ODS [ODS-Team, 2020] lässt sich als eine Zusammensetzung mehrerer Microservices (siehe Abschnitt 3.1.1) beschreiben. Diese sind in eigenen, untergeordneten Verzeichnissen voneinander getrennt und kommunizieren zur Laufzeit über Representational State Transfer (REST) Schnittstellen. Hierbei findet sich zudem eine Trennung von Frontend zum Backend. Diese sind über einen Edge-Server-Pattern, dem Reverse-Proxy, miteinander verbunden.

### 3.1.1 Microservices

Die Microservices Architektur ist ein Architekturdesign, welches große, komplizierte Applikationen aus mehreren Services zusammensetzt. Diese sind in der Regel unabhängig voneinander bereitgestellt und sollten ebenfalls lose gekoppelt sein. Durch diese Unabhängigkeit in der Bereitstellung ist keine Koordination beim Starten der Services nötig, da jede dieser auch als einzelner Service lauffähig ist. Dabei ist ein Microservice auf einen Aufgabenbereich spezialisiert und dafür zuständig. Dieser sollte so klein wie möglich gehalten werden, sodass die Erweiterung und Wartung des Services jederzeit und schnell möglich ist. Zudem benötigt ein Microservice kein Wissen über die gesamte Architektur. Es erfüllt lediglich seinen eigenen Zweck und hat über die Logik anderer Services keine Informationen. Außerdem soll in dieser Architektur unabhängig von einer Programmiersprache entwickelt werden. Im Konkreten bedeutet das, dass jeder Microservice in einer anderen Sprache entwickelt worden sein kann. Dies ist möglich, da die Kommunikation über eine sprachneutrale Application Programming Interfaces (API) wie REST abläuft. Der Vorteil einer sprachneutralen Architektur ist, dass für jeden Service im gegebenen Anwendungsfall die passende Programmiersprache gewählt und verwendet werden kann. [Vennam, August 2005]

### 3.1.2 Service: UI

Der WebClient/UI-Service dient als Bedienoberfläche für den Nutzer. Hier kann dieser verschiedene Datenquellen und Pipelines anlegen und konfigurieren. Des Weiteren bietet dieser Service eine Übersicht aller aktuell genutzten Datenquellen und deren Pipelines.

### 3.1.3 Service: Adapter

Der Adapter hat eine Anbindung zur Datenbank die für Clients, wie die UI, über eine REST-API zur Verfügung gestellt wird. Hierbei werden die vom User angelegten Datenquellenkonfigurationen an den Adapter gesendet und dort entsprechend weiterverarbeitet. Zur Verarbeitung zählen die Imports, Persistierung und Wiederbeschaffung der Daten.

### 3.1.4 Service: Pipeline

Die Pipeline kommuniziert, wie auch der Adapter, mit der UI, dem Storage-Service und der Datenbank. Die Aufgaben sind analog zu dem des Adapters. Der Unterschied besteht darin, dass die Pipeline keine Daten mehr importiert, sondern diese über die definierte Transformationsfunktion transformiert. Die verarbeiteten Daten werden danach persistiert und zur Bereitstellung für den Nutzer an den Storage-Service weitergegeben.

### 3.1.5 Service: Storage

Der Storage-Service arbeitet eng mit der Datenbank zusammen. Dieser Service ist ausschließlich für das Schreiben und Lesen der finalen Daten zuständig. Wird eine Pipelinekonfiguration erstellt, erstellt der Storage eine neue Tabelle für die transformierten Daten dieser Konfiguration. Der Nutzer kann sich über einen Menüpunkt der Pipeline diese Daten schließlich anzeigen lassen.

### 3.1.6 Message-Broker: AMQP

Die AMQP dient zur allgemeinen Kommunikation zwischen den Services. Dieser ist ein Message Broker, wodurch die einzelnen Services unter bestimmten Topics ihre Nachrichten hinterlassen können. Die einzelnen Clients können auf diese Topics lauschen und die hinterlegten Nachrichten abfragen und verarbeiten.

## 3.2 Vergleich von strukturierten und unstrukturierten Daten

Nachdem der Nutzer eine Datenquellenkonfiguration angelegt hat, werden über diese die entsprechenden Daten angefordert. Im Anschluss gilt es diese in eine Datenbank zu persistieren. Hierbei bieten sich zwei Möglichkeiten.

### 3.2.1 Strukturierte Daten mittels SQL

Strukturierte Daten werden nach einem festgelegten Schema in einer relationalen Datenbank mittels SQL persistiert. Wie der Name bereits andeutet, können in dieser Art der Speicherung auch Beziehungen zwischen den einzelnen Tabellen hergestellt werden. Dadurch lassen sich Beziehungen in den importierten Daten darstellen. Diese Eigenschaft der SQL-Datenbanken erlauben dem Nutzer aufwändige und komplexe Anfragen an die Datenbank zu stellen und auch jegliche Daten nach Belieben zu lesen. Da die Tabellen alle relevanten Attribute der importierten Daten bereits vorher enthalten müssen, werden jedoch werden jedoch keine neuen oder veränderten Attribute zur Laufzeit automatisch in die Datenbank aufgenommen.

### 3.2.2 Unstrukturierte Daten mittels NoSQL

Unstrukturierte Daten können mittels NoSQL Datenbanken gespeichert werden. Hierbei gibt es viele Ansätze wie dies geschehen kann. Eines dieser Ansätze ist die Key-Value. Damit werden die angeforderten Daten ohne jegliches Schema unter einer einzigartigen ID gespeichert. Dieses Verfahren ähnelt dem einer Map. Somit kann zur Laufzeit jede Art von Datentyp in diese Form der NoSQL-Datenbank geschrieben werden. Auf Grund dieses Vorgehens können die gespeicherten Daten jedoch nur mit entsprechender ID wieder aus der Datenbank gelesen werden. Somit erzielt man ein Verhalten, welches schnelle Zugriffe und Schreiboperationen erlaubt, jedoch vollends auf den Zugriff über ausschließlich der ID abhängig ist. [Klößner, Dezember 2015], [Jablonski, 2011] Eine weitere Methode ist das Dokumentenmodell. Hierbei werden die Daten in JSON bzw. JSON-ähnlichen formaten als Dokument in der Datenbank gespeichert. Dabei werden die Werte aus den Daten einzeln in dieses Dokument geschrieben und erhalten zudem eine eindeutige ID zugewiesen. Auch diese Methodik ist schemafrei, wodurch zur Laufzeit jegliche Daten in diesem Format gespeichert werden können. Im Gegenzug zur Key-Value Methode lassen hier auch die einzelnen Werte der Daten anfragen, womit man gezielter mit diesen Daten arbeiten kann. [Jablonski, 2011]

### 3.2.3 Projektion auf den Anwendungsfall ODS

Da beliebig viele Nutzer das ODS nutzen können, werden auch dementsprechend viele Datenimporte stattfinden, wobei sich der überwiegende Teil in der Form der Daten unterscheiden wird. Diese Tatsache begünstigt die Verwendung einer Darstellung als unstrukturierte Daten. Jedoch würde dies den Endbenutzer vor zwei bedeutenden Problemen stellen. Die importierten Daten wären hier zwar an einem zentralen Punkt gesammelt und verfügbar, jedoch hat der Nutzer kaum bessere Möglichkeiten auf diese Daten zuzugreifen als direkt von der Quelle selbst. Das zweite Problem ist, dass der Nutzer nicht über eine eventuell veränderte Datenstruktur seiner importierten Daten informiert wird, da die unstrukturierten Daten keinem bestimmten Schema entsprechen. Anders ist es mit den strukturierten Daten. Diese erhöhen die Komplexität des Codes immens, bieten jedoch dem Nutzer den maximalen Nutzen. Durch Analyse der Daten und das Erstellen dynamischer Schemata können die angelegten Tabellen die importierten Daten aufnehmen und dem Kunden so bereitstellen. Dieser kann dann über gezielte Anfragen (SQL) diese Daten nach Belieben anfordern. Da auch ein Schema existiert, ist es möglich, dem Nutzer bei Änderungen der importierten Daten mitzuteilen, dass seine angelegte Tabelle nicht mehr vollständig den neuen Daten entspricht. Somit kann dieser ein neues Schema anfordern, bzw. seine Tabelle dahingehend aktualisieren.

### 3.2.4 Auswahl der passenden Datenbank

Da im Abschnitt 3.2.3 elaboriert wurde, dass sich die Arbeit auf strukturierte Daten konzentriert, wird in diesem Abschnitt eine passende SQL-Datenbank gewählt. Dazu gibt es jegliche Formen der SQL Datenbanken. Als Open-Source Möglichkeiten kann hier PostgreSQL oder MySQL genutzt werden. Die folgenden Eigenschaften treffen auf beide Datenbanken zu. Sie unterstützen den aktuellen SQL-Standard 'SQL-92'. Um die Relationen zwischen den einzelnen Tabellen herzustellen, werden hierbei sogenannte Primär- und Fremdschlüssel genutzt. Diese verbinden zusammengehörige Datensätze aus verschiedenen Tabellen, wodurch die Zugriffe auf solche Daten ermöglicht wird. Des Weiteren werden beide Datenbanken in vielen Programmiersprachen unterstützt, wodurch eine Integration in fast jedem Code möglich ist. Dennoch gibt es auch Unterschiede in diesen Datenbanken, welche für die Nutzung von PostgreSQL sprechen. Zum einen bietet PostgreSQL die Möglichkeit benutzerdefinierte Datentypen innerhalb der Datenbank zu verwenden. Dies ist ein kritischer Punkt, da viele verschiedene Daten in dem Anwendungsfall der Arbeit importiert werden und auch die Entwicklung für dynamische Tabellen hierdurch unterstützt wird. Ein weiterer Punkt ist die grundsätzliche Erweiterbarkeit von PostgreSQL. Dies gibt der Entwicklung mehr Flexibilität in der Zukunft. Damit kann die aktuell verwendete PostgreSQL-Datenbank im ODS auch mit bestem Gewissen im Rahmen dieser Arbeit verwen-



det werden. [Douglas, Februar 2003]

### 3.3 Vergleich etablierter Schemarepräsentationen

Die gängigsten Datenformate bei der Kommunikation im Web sind aktuell die Formate JSON und XML. Davon ausgehend lassen sich die in Frage kommenden Schemarepräsentationen auf eine übersichtliche Menge reduzieren. Gemeint sind hier JsonSchema und XML Schema Definition (XSD).

#### 3.3.1 JsonSchema

JsonSchema (siehe Abbildung 3.1) ist ein Werkzeug zur Beschreibung und Validierung von JSON-Objekten. Das Schema selbst wird als JSON-Objekt dargestellt. JSON ist ein leichtgewichtiges Datenaustauschformat, welches für den Menschen leicht zu lesen und schreiben sein soll. Dieses Format wird aktuell in vielen Systemen zur Kommunikation von Daten genutzt. Dies wird unter anderem ermöglicht, in dem JSON trotz seines Namens unabhängig von jeder Programmiersprache arbeitet. Dennoch basiert es auf der Objektnotation von JavaScript. Da die Darstellung des Jsonschemas identisch zu einem JSON ist, lassen sich alle obigen genannten Punkte ebenfalls auf dieses Tool anwenden.

```
1   {
2     "$schema": "http://json-schema.org/draft-07/schema",
3     "type": "object",
4     "required": [
5       "test",
6       "secondTest"
7     ],
8     "properties": {
9       "test": {
10        "type": "string"
11      },
12      "secondTest": {
13        "type": "integer"
14      }
15    },
16    "additionalProperties": true
17  }
```

---

Abbildung 3.1: JsonSchema Beispiel Darstellung

#### 3.3.2 XML Schema Definition

XSD (siehe Abbildung 3.2) ist ein Werkzeug zur Beschreibung von XML-Objekten. Dieses Schema verwendet XML als Datenstruktur, weswegen folgende Aussagen, welche auf XML bezogen sind, sich analog auf XSD anwenden lassen. Wie auch das oben genannte JSON, ist XML vom Menschen einfach zu lesen und zu schreiben. Es bietet die Möglichkeit, Objekte serialisiert darzustellen und wird als Datenübertragungsformat genutzt. [Yergeau, November 2008]

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <xs:schema attributeFormDefault="unqualified" elementFormDefault="
   qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="root">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element name="test" type="xs:string" />
7          <xs:element name="secondTest" type="xs:unsignedByte" />
8        </xs:sequence>
9      </xs:complexType>
10   </xs:element>
11 </xs:schema>
```

Abbildung 3.2: XSD Beispiel Darstellung

#### 3.3.3 Vergleich beider Tools

Beide Formate sind vollends in der Lage, die gewünschte Aufgabe in dieser Arbeit zu erfüllen. Sie erfüllen alle wichtigen Kriterien, die hier nötig sind. Sie sind vom Menschen einfach zu lesen und zu schreiben, können mit Validatoren die beschriebenen Daten validieren, unabhängig der Programmiersprache verwendbar und finden jeweils weitreichend im Webbereich Anwendung.

Im direkten Vergleich erweist sich jedoch JSON in Sachen Performance und Ressourcen als vorteilhafter. Dies wird an folgendem Szenario deutlich. Es werden sowohl mit XML, als auch mit JSON als Datenformat 20.000, 40.000, 60.000, 80.000 und 100.000 Objekte versendet und dabei die oben genannten Punkte beobachtet. Wie der Abbildung 3.3 entnommen werden kann, werden die JSON-Objekte im Vergleich zu XML um ein Vielfaches schneller bearbeitet. Die Abbildung 3.4 zeigt deutlich, dass die Verarbeitung der Daten mithilfe von JSON nur ein Drittel der System-CPU Auslastung bei Verwendung von XML benötigt. Was den Speicherverbrauch belangt, bewegen sich beide Formate im gleichen Bereich. [Izurieta, 2009]

	JSON	XML
Trial 1 Number Of Objects	20000	20000
Trial 1 Total Time (ms)	2213.15	61333.68
Trial 1 Average Time (ms)	0.11	3.07
Trial 2 Number Of Objects	40000	40000
Trial 2 Total Time (ms)	3127.99	123854.59
Trial 2 Average Time (ms)	0.08	3.10
Trial 3 Number Of Objects	60000	60000
Trial 3 Total Time (ms)	4552.38	185936.27
Trial 3 Average Time (ms)	0.08	3.10
Trial 4 Number Of Objects	80000	80000
Trial 4 Total Time (ms)	6006.72	247639.81
Trial 4 Average Time (ms)	0.08	3.10
Trial 5 Number Of Objects	100000	100000
Trial 5 Total Time (ms)	7497.36	310017.47
Trial 5 Average Time (ms)	0.07	3.10

Abbildung 3.3: Timing JSON vs XML [Izurieta, 2009]

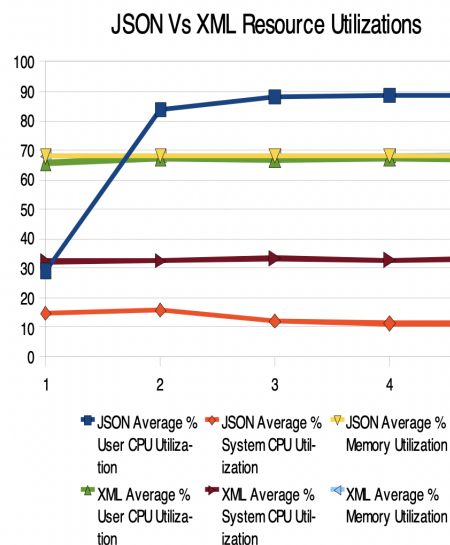


Abbildung 3.4: Ressourcenauslastung JSON vs XML [Izurieta, 2009]

Da abhängig der Datenquellen und der Abfragefrequenz die Menge der Daten sehr hoch sein kann, ist es von großem Vorteil, hierbei das JSON-Format zu nutzen, um dem Nutzer eine performante Anwendung bieten zu können. Aus diesem Grund wird in dieser Arbeit das JsonSchema verwendet.

### 3.4 Algorithmus für den Schemavorschlag

Basierend auf den eingehenden Daten erstellt der Algorithmus zunächst eine sogenannte Ontology (siehe Abbildung 3.5). Diese ist ein Zusammenschluss mehrerer

Knoten, wobei diese den Schlüssel als Namen, den dazugehörigen Typen und weitere untergeordnete Knoten abspeichern. Bei eingehenden Daten wird zunächst ein Rootknoten erstellt, dessen Name und Type 'root' und das Knotenarray leer ist. Die nächsten Schritte werden abhängig vom Typen wiederholt.

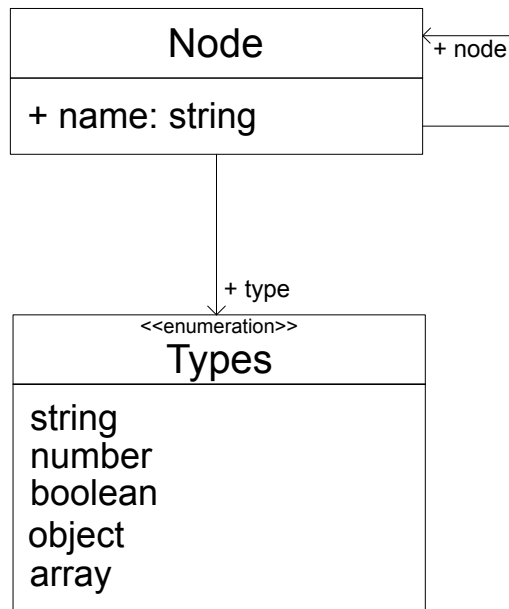


Abbildung 3.5: Klassendiagramm der Ontologyklasse

#### Primitiver Typ

Für primitive Typen wird der Schlüssel als 'name' und der dazugehörige Typ unter 'types' gespeichert. Das Knotenarray bleibt in diesem Fall immer leer. Siehe Abbildung 6.1 und 6.2 für passende Fallbeispiele.

#### Object

Für Objekte kann der Ablauf wie bei einem primitiven Typen gesehen werden. Im Gegenzug zum oben genannten Fall kann ein Objekt aus ein bis mehreren primitiven und komplexen Typen bestehen. Diese werden in das Knotenarray des Objektknoten gespeichert. Siehe Abbildung 6.3 und 6.4 für passende Fallbeispiele.

#### Array

Arrays werden in einer Schleife abgearbeitet. Die Anzahl der Elemente, die geprüft werden sollen, werden von dem User bestimmt. Diese werden dann per Zufall aus dem gesamten Datensatz ausgewählt und analysiert. Hierbei muss unterschieden werden, welche Art von Daten im Array gespeichert sind. Im Falle von weiteren komplexen Daten wie Objekten wird der Typ im neuen Knoten als 'array' deklariert und weitere Knoten werden nun in das Knotenarray gespeichert. Im Falle von primitiven Daten wie Strings oder

Integers wird der Knoten mit dem entsprechenden Typen als Array festgehalten. Beide Fälle enthalten den Schlüssel als Namen. Siehe Abbildung 6.5 und 6.6 für passende Fallbeispiele.

Die oben genannten Fälle werden so über den festgelegten Teil der Daten abgearbeitet und im Anschluss als Ontology zurückgegeben.

### 3.5 Design der Nutzerinteraktion

Alle Interaktionen bezüglich des Schemas sind für den Nutzer optional, um die gewohnten Funktionen der ursprünglichen Applikation zu gewährleisten.

#### 3.5.1 Interaktion für Datenquellen

Die Interaktion für den Anwender ist in mehrere Schritte aufgeteilt. Zunächst kann dieser, sollte keine Datenquelle vorhanden sein, eine Datenquelle angeben. Hierbei sollte es sich um eine Quelle im JSON Format handeln. Nun kann der Nutzer sich hierzu einen Schemavorschlag generieren lassen. Um die Dauer und Qualität der Schemagenerierung anpassen zu können, kann der User den gegebenen Regler verwenden und somit den Detailgrad<sup>1</sup> der Generierung bestimmen. Der Grad der Generierung bestimmt, wie viele einzelne Elemente aus einem Array überprüft werden sollen. Das Beispiel aus der Abbildung 3.6 zeigt ein Array, welches zwar einheitlich Objekte enthält, diese jedoch verschiedene Attribute haben. Wählt man hier einen sehr geringen Detailgrad, z. B. 'precision = 1', so würde nur ein zufälliges Objekt dieses Arrays kontrolliert werden. Im Gegenzug hätte man allerdings eine schnellere Analyse der Daten erreicht. Verzichtet man auf die Geschwindigkeit und wählt 'precision = 3', so würde ein Schema generiert werden, welches alle drei Attribute enthält.

---

<sup>1</sup>Im Code precision genannt

```
1  [
2    {
3      name: 'foo'
4    },
5    {
6      birthday: '20.01.2001'
7    },
8    {
9      adress: 'bar'
10   }
11  ]
```

---

**Abbildung 3.6:** Beispiel JSON-Objekt mit einem Array mit drei verschiedenen Objekten

Nach der Einstellung des Detailgrades und Senden der Anfrage, erhält der Nutzer einen entsprechenden Schemavorschlag. Diesen kann er nun im Textfeld einsehen und selbstständig bearbeiten und speichern. Anschließend muss ein Intervall für die Ausführung der gewählten Einstellung gesetzt werden.

#### 3.5.2 Interaktion für Pipelines

Ist eine Datenquelle angelegt, kann der Nutzer eine bis n Pipelines für diese erstellen. Hierbei ist der Ablauf ähnlich dem der Datenquellen. Der User kann hier für die importierten Daten eine Transformationsfunktion definieren, mit welcher dieser die Daten wie gewünscht anpassen kann. Anschließend kann auch hier wieder entschieden werden, ob ein Schema generiert werden soll. Die Generierung des Vorschlags wird in diesem Schritt auf den bereits transformierten Datensatz angewendet und dem Nutzer vorgeschlagen. Nach der Anpassung seitens des Users wird das Schema gespeichert und in der Pipelinekonfiguration hinterlegt.

#### 3.5.3 Healthchecker

Um den Zustand der angelegten Datenquellen oder Pipelines zu überwachen und beschreiben zu können, ist hierfür der sogenannte Healthcheck entwickelt worden. Dabei werden vor jedem Import oder jeder Transformation die entsprechenden Daten aus der Quelle/Pipeline übergeben. Waren die jeweiligen Prozesse zur Datenbeschaffung erfolgreich werden, falls ein Schema vorhanden ist, die importierten/transformierten Daten gemäß dem Schema validiert. Der Healthchecker kann dem Nutzer vier verschiedene Zustände anzeigen:

- Grau, wenn kein Import/keine Transformation stattgefunden hat
- Grün, wenn sowohl Import/Transformation als auch die Validierung fehlerfrei waren
- Gelb, wenn die Daten nicht mehr dem Schema des Nutzers entsprechen
- Rot, wenn der Import/die Transformation nicht möglich war

### 3.5.4 Interaktion mit der Bereitstellung der Daten

Der Nutzer ist nach erfolgreichem Ablauf der Pipeline in der Lage über eine API alle Daten, welche dieser im Schema festgelegt hat, über GraphQL [Foundation, 2021] abzufragen und nach Belieben weiter zu verarbeiten. Da die befragten Nutzer des ODS Wert auf eine leichte und trotzdem umfangreiche Art der Datenabfrage legen, ist PostGraphile [Team, 2021] hier die optimale Lösung. Es verbindet die hier vorteilhafte Querysprache GraphQL als auch die bereits vorhandene Datenbank PostgreSQL. Sie bietet eine effiziente Anbindung an das bestehende Datenbanksystem und ermöglicht zugleich jegliche Interaktion mit dieser Datenbank. Dabei hat der Nutzer den Vorteil, basierend auf seinen Anwendungsfall, die bereitgestellte API von PostGraphile zu nutzen oder die mehr nutzerfreundliche Art über die verfügbare UI.

## 3.6 Verantwortlichkeiten der Services/Komponenten

Die Implementierung der Arbeit wird sich aus einem eigenen Mikroservice, der Erweiterung des bestehenden Systems und einer Schnittstelle zur Abfrage aller gespeicherten Daten zusammensetzen.

### 3.6.1 Service: Schema

Dieser Mikroservice dient als Schnittstelle zur Generierung von Schemata für Daten im JSON-Format. Hier werden zunächst die importierten bzw. transformierten Daten empfangen und anschließend in ein eigens entwickeltes Schema, der sogenannten Ontology, umgewandelt. Diese Ontology stellt die empfangenen JSON-Daten in einem Schema dar und soll als Ausgangspunkt für die Umwandlung in weitere Schemata genutzt werden. Hierbei ist ein wichtiger Punkt der Entwicklung nicht abhängig von einem spezifischen Schema zu sein. Ausgehend von der Ontology kann basierend auf das gewählten Schema in der Hauptapplikation, das ausgegebene Schema jederzeit geändert werden. Für diese Arbeit wurde JsonSchema gewählt. Somit wird die Ontology von einem Parser in ein JsonSchema-Format umgewandelt und an den anfragenden Service zurückgesendet.

### 3.6.2 Open-Data-Service

Die Hauptapplikation bietet, wie oben genannt, bereits einige Services an, welche im Zuge dieser Arbeit weiter ausgebaut worden sind.

#### UI

Die UI muss dem User die Möglichkeit bieten, den Schemaservice in Anspruch nehmen zu können. Aus diesem Grund wurde diese sowohl im Bereich der Pipeline, als auch im Datenquellenbereich um einen weiteren Schritt, der Schemagenerierung, erweitert. Dies ist die Anbindung für den User an den Schema-Service.

#### Adapter

Der Adapter-Service prüft den Zustand der erstellten Datenquellen hinsichtlich der Importe und der Gültigkeit im Bezug auf ein eventuell vorhandenes Schema. Dies wird in der Datenbank persistiert und somit den anderen Services zur Verfügung gestellt.

#### Pipeline

Der Pipeline-Service prüft den Zustand der erstellen Pipeline hinsichtlich der Transformationen und der Gültigkeit im Bezug auf ein eventuell vorhandenes Schema. Zudem soll die Pipeline die transformierten Daten bei jedem Intervall in der Datenbank persistieren. Sowohl die transformierten Daten, als auch die Konfiguration sind für die weiteren Services erreichbar und verwendbar.

#### Storage

Die Schnittstelle in dem Storage-Service wurde ebenfalls hinsichtlich dem Schema-Support erweitert. Hier ist es möglich, dass anhand dem übergebenen Schema ein Create-Statement dynamisch für die unterliegende Datenbank erstellt und ausgeführt werden kann. Im Anschluss wird periodisch und dynamisch zu den transformierten Daten ein Insert-Statement generiert. Diese Schritte machen es möglich, die Datenbank völlig automatisch und dynamisch zu den jeweilig gegebenen Daten zu generieren.

### 3.6.3 PostGraphile Schnittstelle

Dem Nutzer wird eine Schnittstelle geboten, mit welcher dieser über das Kommunikationsmuster GraphQL die gewünschten Daten abfragen und weiterverarbeiten kann.



## 3.7 API Design

Diese API-Schnittstellen arbeiten eng mit den dazugehörigen Modellen zusammen. Aus diesem Grund ist es oft ausreichend nur diese zu ändern. Somit müssen in der Ausgangsapplikation keine bis kaum Änderungen an der API vorgenommen werden.

### Adapter

Um die Schemagenerierung gemäß den Anforderungen nutzen zu können, musste sowohl die REST-API des Adapter, als auch dessen Kommunikation mit der AMQ-Schnittstelle um das Attribut 'schema' erweitert werden.

### Pipeline

Die Pipeline unterstützt jetzt auch die Speicherung der transformierten Daten und deren Meta-Daten. Dadurch wurde eine weitere Route eingeführt. Mit dieser lassen sich die letzten Transformationsdaten der angefragten Pipelinekonfiguration aus der Datenbank lesen.

```
1     getLatest = async (  
2         req: express.Request,  
3         res: express.Response  
4     ): Promise<void> => {
```

---

**Abbildung 3.7:** Schnittstelle für die aktuellsten transformierten Daten

Die REST-API und AMQP Änderungen sind analog zum Adapter zu sehen.

### Storage

Dieser Service beinhaltet ebenfalls Änderungen analog zum Adapter.

### Schema

Hier kann der UI-Service sowohl bei der Datenquellen- als auch der Pipelinekonfiguration eine Schemagenerierung angefragt werden. Um dies zu ermöglichen, bietet sie eine API-Schnittstelle mit zwei Anbindungspunkten. Über die Route '/alive' kann abgefragt werden, ob der Service verfügbar ist. Die zweite Route '/suggestion' benötigt die Daten und den Detailgrad, in welchem das Schema generiert werden soll.



# 4 Implementierung

## 4.1 Verwendete Sprachen

Da diese Arbeit ein Teil einer bereits existierenden Applikation ist, sind in der Sprachauswahl die Möglichkeiten begrenzt. Aus diesem Grund ist die hauptsächlich verwendete Sprache TypeScript und im Falle des Adapter-Service Java

## 4.2 Docker

Das Containertool Docker wird im Rahmen der Arbeit genutzt, um jedem Service seinen eigenen Container zu geben. Dabei können diese Container über einen Reverse Proxy miteinander kommunizieren. Die Container sind zudem konfiguriert, dass beim Bau dieser der Code vorher getestet wird und somit sichergestellt sein kann, dass man in einer funktionierenden Umgebung arbeitet. Außerdem bietet diese Vorgehensweise den Vorteil, die Github-Actions zu vereinfachen. Somit sind keine separaten Unittests von Nöten. Während der Entwicklung ergaben sich ebenso Nachteile dieser Container. Durch die Trennung hinsichtlich des Repository des Schema-Services vom restlichen ODS müssen beide Projekte separat voneinander gebaut werden. Damit die Container sich gegenseitig wie gewohnt ansprechen können, muss auch bei der Art und Weise, wie diese gestartet werden, ein Punkt beachtet werden. Die Ausführung des gesamten Projekts, ODS und Schema, funktioniert wie folgt:

```
'docker-comopse -f ./docker-compose.yml -f ./<Schema-Service Ordner>/docker-compoe.yml up'
```

Dieser Befehl muss aus dem ODS-repository aus gestartet werden. Durch das Argument '-f' verweist man auf die docker-compose.yml Datei. Indem beide compose Dateien angegeben sind, werden diese zu einem kombiniert und ausgeführt. Des Weiteren muss beachtet werden, dass der Container für den Schema-Service trotzdem direkt aus dessen Ordner gebaut wurde.

### 4.3 Aufteilung der Arbeitspakete

Um dem Implementierungsprozess eine Struktur zu geben und zielgerichtet entwickeln zu können, wurden zunächst die Anforderungen betrachtet. Diese wurden anschließend in passende Arbeitspakete aufgeteilt, um darauf basierend eine zeitliche Orientierung für die Arbeit zu haben. Die Arbeitspakete sind wie folgt gewählt:

- Schemagenerierung für die Datenquellen
- Healthindikator für Datenquellen
- Schemagenerierung für die Pipelines
- Healthindikator für Pipelines
- Übertragen der Daten in die Datenbank und Bereitstellung mittels einer Query-API.

### 4.4 Implementierungsdetails Schema-Service

Wie bereits in 4.1 erwähnt, ist die hier verwendete Sprache TypeScript. Der Service wird hauptsächlich vom UI-Service verwendet um die Schemavorschläge anzufordern.

#### 4.4.1 API

Die Anbindung erfolgt über die eine REST-API, welche in der 'index.ts' definiert ist. Um diese zu realisieren, wurde hierfür die Library Express verwendet. [Wilson, 2021]. Hier gibt es zwei Routes:

##### **/alive**

Über diese Route kann der UI-Service abfragen, ob der Schema-Service gestartet und verfügbar ist. War die Anfrage erfolgreich, wird eine Nachricht mit dem Text 'alive' als Antwort gesendet.

##### **/suggestion**

Hierüber wird der Schemavorschlag angefordert. Dazu müssen die Daten aus der Datenquelle mit übermittleit werden. Daraufhin wird zunächst vom SchemaGenerator eine Ontology für diese Daten erstellt. Als Nächstes wird diese zu einem JsonSchema geparsed und als Antwort zurückgesendet.

#### 4.4.2 Ontology Generierung

Dies ist der erste Schritt zur Generierung des Schemavorschlags. Die Ontology (siehe Abbildung 3.5) selbst ist ein Kompositum aus Knoten. Das Feld 'name' re-

präsentiert einen Schlüssel aus dem Übergebenen JSON-Objekt. 'type' beschreibt den Typen des Schlüssels. Wenn der Schlüssel ein Objekt oder Array beinhaltet, werden die untergeordneten Attribute in dessen 'node' als weitere Knoten hinterlegt.

Zum Beginn der Ontologygenerierung wird die Funktion 'startNodeGeneration' aufgerufen.

```
1 function startNodeGeneration (data: any, precision = 1): Node
```

---

Diese erwartet zwingend die Daten für den Schemavorschlag als JSON-Objekt. Der Parameter 'precision' reguliert den Detailgrad, welcher von eins bis max definierbar ist, der Ontologygenerierung. Die Funktion selbst erstellt der RootKnoten und gibt dies mit allen Parametern weiter an die generateNode Funktion.

```
1 function generateNode (rootNode: Node, data: any, precision = 1): Node
```

---

Hier wird geprüft, ob das nächste Element ein Array oder Objekt ist. Dieser Schritt ist mindestens einmal im gesamten Algorithmus zwingend erforderlich, da jede Datei im JSON-Format entweder als Array oder Objekt beginnt. Abhängig vom vorliegenden Typen wird die entsprechende Funktion aufgerufen.

Handelt es sich um ein Objekt, wird die Funktion handleObjectNode aufgerufen.

```
1 function handleObjectNode (rootNode: Node, keys: string[], data: any,  
    precision = 1): void
```

---

Die übergebenen Schlüssel werden hinsichtlich ihrer Typen überprüft. Sollte es sich weder um ein Array, noch um ein Objekt handeln, wird die Funktion für primitive Datentypen verwendet. Handelt es sich um ein Objekt, wird wieder die oben genannte generateNode Funktion ausgeführt. Sollte es ein Array sein, wird handleArrayNode ausgeführt.

```
1 function handleArrayNode (rootNode: Node, rootName: string, data: any,  
    precision = 1): void
```

---

In dieser Funktion ist nun der Parameter 'precision' relevant. Da Arrays beliebig groß mit ähnlichen Daten befüllt sein können, wird hier zwischen Qualität und Performance abgewogen. Je größer die 'precision' ist, desto länger arbeitet der Algorithmus. Es werden zunächst gemäß der im Parameter festgelegten Anzahl an zufälligen Indizes im Bereich des Datenarrays ermittelt. Die zufällig ausgewählten

Indizes aus dem Datenarrays werden nun gegen ihren Typen getestet. Werden in verschiedenen Feldern des Arrays unterschiedliche Attribute gefunden, können sie so in dem Schema zusammengesetzt werden. Je größer also der Präzisionswert ist, desto höher ist die Wahrscheinlichkeit alle Attribute eines Arrays zu finden.

```
1 function handleAttributeNode (rootNode: Node, currentNodeName: string,  
    valueType: string): void
```

---

Diese Funktion erstellt die Blätter bzw. Endknoten im gesamten Kopositum. Diese haben keine weiteren Knoten und stellen somit einfache Datentypen wie Integer, String und ähnliches dar.

### 4.4.3 Artefakte der Ontology

Zu Beginn sollte der Healthindikator über eine API-Schnittstelle im Schema-Service angeboten werden. Das würde jedoch, wie nach späterer Evaluierung festgestellt, potentiell zu hohem Netzwerkverkehr kommen. Grund dafür ist die Ausführung der beiden Konfigurationen in der Datenquelle und der Pipeline. Da diese immer in einem bestimmten Intervall eine Validierung auslösen würden, müsste für jede dieser Konfiguration pro Ausführung einmal die API verwendet werden. Dadurch ist ein Artefakt aus der bisherigen Entwicklung übrig geblieben.

Sie arbeitet mit der Ontology zusammen. Dabei kann dieser Klasse ein Ontologyschema und die dazugehörigen Daten übergeben werden. Im Anschluss kann nun anhand des Schema gezielt der aktuell erste Datensatz aus den übergebenen Daten extrahiert werden. Diese Funktion kann bei Bedarf erweitert werden, in dem man gezielt eine Menge von Daten aus dem Datensatz extrahiert. Darauf basierend wäre es auch möglich, einen Validierer zu entwickeln. Aus diesem Grund ist diese Implementierung für zukünftige Anwendungsfälle bestehen geblieben.

### 4.4.4 Jjsonschema parsing

Die Klasse 'JsonSchemaParser' implementiert das Interface 'Parser'. Die Funktion benötigt dazu die Ontology in einem Knoten-Objekt dargestellt und gibt ein Objekt, welches das geparsete Schema beinhaltet, zurück. Dazu wird hier JsonSchema verwendet. Im ersten Schritt wird im Schema die aktuelle Version des Schemas angegeben. Dies kann benötigt werden, um beispielsweise externen Libraries problemlos die verwendete Version zu übermitteln, um damit die für diese Version angemessene Vorgehensweise zu verwenden. Nun wird unterschieden, um welchen Typen es sich beim aktuellen Knoten handelt. Im Falle eines Objekts wird dazu die Funktion 'buildObject' aufgerufen.

```
1 private buildObject (  
2     schema: Node,  
3     rootId: string,  
4     parentIsArray = false  
5 ): JSONSchema7
```

---

**Abbildung 4.1:** Funktion zur Erfassung von Objekten im JsonSchema

Wie der Name bereits andeutet, werden hier Objekte als Schema dargestellt. Dazu wird zunächst der 'type' als 'object' definiert. Zudem wird die Eigenschaft 'additionalProperties' auf wahr gesetzt. Dies bedeutet, dass ein Objekt, welches zusätzliche Attribute, welche nicht im Schema aufgeführt sind, dennoch als gültiges Objekt erachtet werden kann. Für das aktuelle Objekt werden anschließend alle Schlüssel erarbeitet und diese als zwingend notwendige Attribute festgelegt. Diese Attribute werden darauf folgend unter dem Schlüssel 'properties' hinterlegt. Dabei ist jedes Attribut ein Objekt mit dem Schlüssel als Namen und der dazugehörige Typ als Attribut. Diesen Schritt übernimmt die Funktion 'buildProperties'.

```
1 {  
2     type: 'object',  
3     additionalProperties: true,  
4     required: [  
5         'shortname',  
6         'longname'  
7     ],  
8     properties: {  
9         shortname: {  
10            type: 'string'  
11        },  
12        longname: {  
13            type: 'string'  
14        }  
15    }  
16 }
```

---

**Abbildung 4.2:** JsonSchema für ein Objekt mit zwei string Feldern

Sollte es sich beim aktuellen Typen um ein Array handeln, wird stattdessen die Funktion 'buildArray' aufgerufen.

## 4. Implementierung

---

```
1 private buildArray (  
2     schema: Node,  
3     rootId: string  
4 ): JSONSchema7
```

---

**Abbildung 4.3:** Funktion zur Erfassung von Arrays im JsonSchema

Zu Beginn wird der Typ für den aktuellen Knoten als 'array' festgelegt. Das Attribut 'additionalItems' wird auf wahr gesetzt, um dem Schema die Möglichkeit zu geben, Arrays mit mehr als den definierten Attributen als gültig zu erkennen. Nun wird wieder situationsabhängig gehandelt. Sollte es sich um ein Array aus primitiven Typen handeln, wird dies auch dementsprechend in das Schema eingetragen. Bei einem leeren Array wird dieses Property gänzlich ignoriert, da hierbei nicht der Typ bestimmt werden kann. Andernfalls wird die Funktion 'buildObject' aufgerufen, wobei dessen Rückgabewert unter dem Attribut 'items' gespeichert wird.

```
1 {  
2   type: 'array',  
3   additionalItems: true,  
4   items: {  
5     type: 'object',  
6     additionalProperties: true,  
7     required: [  
8       'uuid',  
9       'adress',  
10    ],  
11    properties: {  
12      uuid: {  
13        type: 'number'  
14      },  
15      adress: {  
16        type: 'string'  
17      }  
18    }  
19  }  
20 }
```

---

**Abbildung 4.4:** JsonSchema für ein Array von Objekten mit zwei Attributen




## 4.5 Implementierungsdetails UI

Über die UI hat der User die Möglichkeit, auf eigenen Wunsch ein Schema für seine gewünschten Daten anzufordern, diese zu verwalten und nach Bedarf jederzeit zu verändern.

### 4.5.1 Datasource

In diesem Reiter kann der Nutzer eine Konfiguration für seine Datenquelle anlegen. Vor der Verwendung wird beim Schema-Service angefragt, ob dieser verfügbar ist. Ist dies nicht der Fall, kann die Erstellung dennoch wie in der ursprünglichen Applikation vorgenommen werden. Im gegenteiligen Fall wird dem Ablauf der Punkt 'Generated schema' hinzugefügt.



3 Generated schema  
Customize the generated schema

1 2 4 8 16 32 64 GENERATE SCHEMA

Datasource schema suggestion

BACK NEXT

**Abbildung 4.5:** Form zur Anforderung und Bearbeitung eines Schemavorschlags

Hier kann der Nutzer sich einen Schemavorschlag generieren lassen, indem er auf den 'GENERATE SCHEMA' Button klickt. Dabei kann er den Detailgrad (siehe Kapitel 3.5.1) über den daneben stehenden Slider im Voraus anpassen. Diese Anfrage wird anschließend über die REST-API im 'SchemaSuggestionREST' versendet. Hierzu wird die Route '/suggestion' verwendet, bei welcher die Daten aus der Quelle und die Präzision mit übergeben werden. Der Vorschlag wird bei erfolgreicher Ausführung anschließend im Textfeld angezeigt. Hier steht es dem Nutzer frei, das Schema nach Belieben und zu jeder Zeit zu verändern. Dazu kann von diesem auch ein eigens erarbeitetes Schema eingefügt werden. Mit dem Klicken auf den 'NEXT'-Button wird das Schema in die Konfiguration übernommen und wird im letzten Schritt der Konfiguration ebenfalls in die Datenbank übernommen. Dem Nutzer steht es frei, diesen Schritt jederzeit zu überspringen

## 4. Implementierung

und ohne eine Schemagenerierung fortzufahren. Bei abgeschlossener Konfiguration kehrt der Nutzer zurück zur Übersicht aller Datenquellenkonfigurationen (Abbildung: 4.8). Hier kann dieser anhand des Statusicons erkennen, in welchem Zustand sich die betrachtete Datenquelle befindet.

Id	Name	Author	Location (URL)	Periodic	Actions	Status
5	DatasourceExample		https://geo.sv.rostock.de/download/opedata/vornamen_von_neugeborenen_2019/vornamen_von_maennlichen_neugeborenen_2019.json	<input checked="" type="checkbox"/>		

**Abbildung 4.6:** Übersicht aktueller Datenquellenkonfigurationen

Um den Status abzufragen, wird über die API des Adapters ein Datenpaket der letzten importierten Daten angefordert. In den zurückgegebenen Daten kann anschließend der aktuelle Zustand ausgelesen werden und auf der UI aktualisiert werden.

### 4.5.2 Pipeline

Hier kann der Nutzer eine neue Pipelinekonfiguration für eine bereits angelegte Datenquellenkonfiguration erstellen. Sollte die Schemagenerierung verfügbar sein, wird auch hier wie im oberen Abschnitt 4.5.1 beschrieben, eine neue Form verfügbar sein.

3 Generated schema  
Customize the generated schema

1 2 4 8 16 32 64 GENERATE SCHEMA

Pipeline schema suggestion

BACK NEXT

**Abbildung 4.7:** Form zur Anforderung und Bearbeitung eines Schemavorschlags

Die Funktionalität ist analog zu dem aus der Datenquellenkonfiguration zu sehen. Jedoch werden hier statt den importierten die transformierten Daten übermittelt. Auch hier hat der Nutzer nach Erhalt des Schemavorschlags die Möglichkeit,

dieses nach eigenem Belieben zu verändern und anzupassen. Bei abgeschlossener Konfiguration kehrt der Nutzer zurück zur Übersicht aller Pipelinekonfigurationen (Abbildung: 4.8). Hier kann dieser anhand des Statusicons erkennen, in welchem Zustand sich die betrachtete Pipeline befindet.

Id	Datasource ID	Pipeline Name	Author	Action	Status
1	1	ExamplePipeline		DATA EDIT DELETE NOTIFICATIONS	●

Abbildung 4.8: Übersicht aktueller Pipelinekonfigurationen

Um den Status abzufragen, wird über die API der Pipeline ein Datenpaket der letzten transformierten Daten angefordert. In den zurückgegebenen Daten kann anschließend der aktuelle Zustand ausgelesen werden und auf der UI aktualisiert werden.

## 4.6 Implementierungsdetails Adapter

Im Nachfolgenden werden alle betroffenen Sektionen behandelt, welche für die Realisierung der Schemaverwendung und Datenvalidierung notwendig sind.

### 4.6.1 Datasource

Diese Klasse ist für die Speicherung der Datenquellen-Konfigurationen zuständig. Um auch eventuelle Schemata für den Nutzer speichern zu können, wurde eine neues Attribut hinzugefügt.

Um das Schema als JsonSchema, demnach ein JSON-Objekt, in der Datenbank persistieren zu können, muss diese als 'jsonb'-Typ definiert werden.

### 4.6.2 DataImport

Der DataImport ist für die Speicherung der importierten Daten, ihren Metadaten sowie für die dazugehörige Datenquellenkonfiguration zuständig. Um den User beim Import den aktuellen Zustand zu zeigen, wurde diese Klasse um das Attribut 'health' erweitert.

```

1 @Enumerated(EnumType.STRING)
2 private ValidationMetaData.HealthStatus health;

```

## 4. Implementierung

---

Diese wird in der Datenbank als ein String gespeichert. Intern wird es über das Enum Healthstatus [siehe: 4.10] dargestellt. Die Bedeutung der Zustände ist der Beschreibung aus dem Architekturabschnitt 3.5.3 zu entnehmen.

Da der 'Healthstatus' auch vom Validator [siehe: 4.6.4] abhängig ist, dient das Attribut 'errorMessages' zur Speicherung möglicher Fehlermeldungen bei der Validierung.

```
1  @Column(columnDefinition = "text []")
2  @Type(type=
3  "org.jvalue.\ac{ODS}.adapterservice.datasources
4  .model.types.CustomStringArrayType")
5  private String[] errorMessages;
```

---

**Abbildung 4.9:** Definition der Variable mit benutzerdefiniertem Typ

Da PostgreSQL in Kombination mit JPA keine Stringarrays unterstützt, wurde der benutzerdefinierte Typ 'CustomStringArrayType' hinzugefügt. Über diesen Typen wird für die Datenbankzugriffe definiert, auf welche Art und Weise die Daten gelesen und geschrieben werden sollen. Mit der Funktion 'nullSafeGet' werden die entsprechenden Daten aus der Datenbank gelesen und in ein Stringarray umgewandelt. Entsprechend wird bei der Funktion 'nullSafeSet' das entsprechende Stringarray aus dem Datenmodell 'DataImport' in eine für die PostgreSQL nutzbare Form umgewandelt.

### 4.6.3 DatasourceManager

Im DatasourceManager wurde die Ausführung des Imports angepasst. Vor dem tatsächlichen Import wird ein Datenimport mit dem Zustand 'FAILED' angelegt. Anschließend wird der Import versucht. War dieser erfolgreich, wird dieser im zuvor angelegten Datenimport mit dem Status 'OK' gespeichert. Im nächsten Schritt werden, falls ein Schema vorhanden ist, die Daten gegen das Schema validiert. Ist auch dieser fehlerfrei, bleibt der Status bestehen und in dieser Konstellation persistiert. Sollte die Validierung eine Exception werfen, werden die importierten Daten mitsamt der Fehlermeldung des Validators in die Datenbank geschrieben.

#### 4.6.4 Validator

Durch die Integration der Schemata ist es möglich, die importierten Daten zu validieren. Hierzu ist die Schnittstelle 'Validator' definiert worden. Hierdurch muss eine Validierungsklasse die Funktion `validate` implementieren.

```
1 public ValidationMetaData validate(DataImport dataImport);
```

---

Damit ist es möglich, den Validator in Zukunft auch durch neue Implementierung für z.B. andere Schemarepräsentationen einbinden zu können. Die Funktion benötigt einen `DataImport` und gibt anschließend Metadaten der Validierung zurück.

```
1 public class ValidationMetaData {
2     private HealthStatus healthStatus;
3     private String[] errorMessages;
4
5     ...
6
7     public static enum HealthStatus {
8         OK {
9             public String toString() {
10                return "OK";
11            }
12        },
13        WARNING {
14            ...
15        },
16        FAILED {
17            ...
18        }
19    }
}
```

---

**Abbildung 4.10:** Klasse für Metadaten einer Validierung

Die Metadaten setzen sich dabei aus dem Zustand des Imports und den dazugehörigen Fehlermeldungen zusammen. Des Weiteren ist auch hier ein Enum hinterlegt, um die Zustände der importierten Daten einheitlich beschreiben zu können. Da diese Arbeit `JsonSchema` nutzt, um die Schemata darzustellen, ist hierfür ein spezieller Validator implementiert worden. Dieser nutzt zur Validierung die Library `everit` [„everit“, 2021]. Wurde ein JSON-Objekt übergeben, wird dieses mittels `GSON` [„GSON“, 2021] in ein für Java verwendbares JSON-Objekt umgewandelt. Anschließend wird dieses Objekt gegen das verfügbare Schema geprüft.

Bei fehlerfreiem Ablauf wird ein entsprechendes `ValidationMetaData` erstellt und in dem übergebenen `DataImport` gespeichert. Sollte ein Fehler bei der Validierung gefunden werden, werden die für die Exception verantwortlichen Zeilen als Stringarray umgewandelt und wieder in ein `ValidationMetaData` gespeichert und im `DataImport` hinterlegt.

## 4.7 Implementierungsdetails Pipeline

Im Nachfolgenden werden alle betroffenen Sektionen behandelt, welche für die Realisierung der Schemaverwendung und Datenvalidierung notwendig sind.

### 4.7.1 Datenmodelle

Um in den Pipelines Schemata zu verwenden, wurden die bereits existierenden Modelle `'PipelineConfig'` und `'PipelineConfigDTO'` mit dem optionalen Attribut `'schema'` erweitert. Des Weiteren sollten analog zum `DataImport` auch die transformierten Daten in der Datenbank hinterlegt werden. Dazu wurden die Modelle `'PipelineTransformedData'` und `'PipelineTransformedDataDTO'` eingeführt.

```
1  export interface PipelineTransformedData {
2      id: number
3      pipelineId: number
4      healthStatus: HealthStatus
5      data: unknown
6      schema?: object
7      createdAt?: string
8  }
```

---

Abbildung 4.11: Interface für transformierte Daten

### 4.7.2 PipelineTransformationDataRepository

Die Kapselung der Datenbankzugriffe für die transformierten Daten erfolgt in dieser Klasse. Hier sind alle nötigen Funktionen implementiert, um sowohl schreibend als auch lesend auf die `'TransformedData'` zugreifen zu können. Um die gelesenen Daten in weiterführenden Klassen problemlos verwenden zu können, werden diese nach der erfolgreichen SQL-Abfrage sofort dem oben genannten Format [siehe: 4.11, `PipelineTransformedData`] zugeordnet.

### 4.7.3 PipelineTransformationDataManager

Der Manager beinhaltet sämtliche für den Use-Case nötigen Funktionalitäten und verbindet diese mit den aus der zuvor genannten Funktionen der Repositoryklasse. Da diese Daten ausschließlich gespeichert und gelesen werden sollen, werden hier lediglich alle Standardfunktionalitäten für diese Daten angeboten.

### 4.7.4 OutboxEventPublisher

Der Eventoutboxer ist für das Schreiben von Nachrichten auf bestimmten Topics im AMQP zuständig. Diese werden zu einem späteren Zeitpunkt im Ablauf von einem Service aus dem Storage ausgelesen und anschließend weiterverarbeitet. Damit auch der Storage-Service mit Schemata arbeiten kann, wurde diese Klasse mit optionalen Schemaattributen erweitert. Nun werden, falls Schemata vorhanden sind, diese bei der Erstellung von Pipelinekonfigurationen und bei erfolgreichem Ausführen von Datentransformationen ebenfalls mit übergeben.

### 4.7.5 PipelineConfigManager

Dieser Manager bietet alle Funktionen die für eine Pipelinekonfiguration notwendig ist. Unter anderem befindet sich hier der Handler für die in einem festgelegten Intervall wiederholte Datentransformation. Diese überprüft zunächst die Transformation selbst. Im Anschluss werden diese bei Vorhandensein eines Schemas mit Hilfe dessen validiert. Im Falle einer erfolgreichen Transformation werden in jedem weiteren Folgefall die transformierten Daten in das TransformedData Objekt gespeichert. Sollte daraufhin die Validierung einen Fehler aufweisen, wird dieser ebenfalls in das gegebene Objekt gespeichert. Die 'TransformedData' wird anschließend in die Datenbank geschrieben und über den Outboxer ein entsprechendes 'publishSuccess' ausgeführt.

### 4.7.6 Validator

Zur Validierung wird ein Interface zur Verfügung gestellt, um eine einfache und sichere Erweiterung mit neuen Validatoren möglich zu machen.

```
1 validate: (config: PipelineConfig, data: unknown) =>  
    PipelineTransformedDataDTO
```

---

Implementiert ist diese Schnittstelle durch die Klasse 'JsonSchemaValidator', da JsonSchema hier die gewählte Repräsentationsform ist. Die Validierung erfolgt über die Library ajv [„ajv“, 2021]. Nach Abschluss der Überprüfung wird das TransformedData Objekt mit dem entsprechenden Status befüllt und an den Aufrufer zurückgegeben.

### 4.8 Implementierungsdetails Storage

Im Nachfolgenden werden alle betroffenen Sektionen behandelt, welche für die Realisierung der Schemaverwendung und Datenvalidierung notwendig sind.

#### 4.8.1 EventHandler

Im Storage-Service sind zwei Handler für Ereignisse verfügbar. Zum einen muss auf das Erstellen einer Pipelinekonfiguration und zum anderen auf die Ausführung dieser Konfiguration reagiert werden. Beide Eventhandler prüfen, ob sich in der AMQP-Nachricht ein Schema befindet. Sollte dies der Fall sein, wird zusätzlich in jeweiligen Szenario eine entsprechende Funktion aufgerufen. Andernfalls wird der Handler wie in der ursprünglichen Applikation ausgeführt.

#### 4.8.2 PostgresParser

Da die Daten in einer PostgreSQL-Datenbank zur Verfügung gestellt werden sollen, wurde hierfür eine Parser-Schnittstelle definiert. Diese bietet zum einen eine Funktion zur Erstellung eines PostgreSQL-Schemas und zum Anderen zum Einfügen in eine PostgreSQL-Tabelle. Eine solche Funktion benötigt folgendes:

- Das aktuelle Schema, bezogen auf die Daten
- Den Namen des PostgreSQL-Schemas
- Den Namen der anzulegenden Tabelle
- Ausschließlich beim Einfügen die relevanten transformierten Daten aus der AMQP-Nachricht.

Das Interface wird aktuell durch die Klasse 'JsonSchemaParser' implementiert, da die eingehenden Daten auf das JsonSchema basieren.

#### **parseCreateStatement**

Diese Funktion soll das Schema auslesen und davon ausgehend dynamisch eine PostgreSQL-Anfrage zur Kreierung einer passenden Tabelle erstellen.



```
1  async parseCreateStatement (
2    schema: any,
3    pgSchemaName: string,
4    tableName: string,
5    index: number = 0,
6    parentName: string = ''
7  ): Promise<string[]>
```

**Abbildung 4.12:** Funktion zur Generierung eines oder mehrerer Create-Statements

Das auszulesende Schema im JsonSchema-Format ist im Parameter 'schema' hinterlegt. Die Kombination aus 'pgSchemaName' und 'tableName' ergeben das Schema und die dazugehörige Tabelle, die erstellt werden soll. Der 'index' ist ein interner Parameter, welcher Queries in der richtigen Reihenfolge voneinander trennt. Als letzter Parameter wird 'parentName' übergeben. Dieser verändert sich immer mit der Tiefe der Aufrufe der 'parseCreateStatment' Funktion. Sie enthält zu jeder Zeit den tableName des übergeordneten Aufrufs.

Zu Beginn überprüft die Funktion den Typen des ersten Elements im Schema um den richtigen Einstiegspunkt für die Folgefunktion zu definieren. Anschließend wird die Funktion 'doParseCreate' mit den entsprechenden Werten aufgerufen. Dort wird zu Beginn immer der erste Teil einer Create-Query in den späteren Endquery angefügt. Dieser Create-Query enthält dabei immer den Namen der zu erstellenden Tabelle, wie auch die Spalten 'id' und 'createdAt'. Anschließend wird für jeden Schlüssel der zugehörige Typ überprüft. Handelt es sich um ein komplexen Typen, also ein Objekt oder Array, wird mit passenden Schemaabschnitt, dem 'tableName' kombiniert mit dem aktuellen Schlüssel als neuer 'tableName' und dem alten 'tableName' als 'parentName' die Funktion 'parseCreateStatement' aufgerufen. Ausgenommen sind Arrays, welche Arrays von primitiven Typen sind wie z.B. ein 'string[]'. Diese werden wie auch andere primitive Typen direkt an den aktuellen Querystring angefügt. Sind alle Schlüssel abgearbeitet, wird bei Bedarf eine Fremdschlüsseldefinition angefügt. Als letzter Schritt folgt immer eine Primärschlüsseldefinition. Vor der Rückgabe der generierten CreateStatements werden diese noch einmal final angepasst und anschließend zurück an den Anforderer gesendet.

## 4. Implementierung

---

```
1 'CREATE TABLE IF NOT EXISTS "TESTSCHEMA"."TESTTABLE" (' +
2   '"id" bigint NOT NULL GENERATED ALWAYS AS IDENTITY, ' +
3   '"createdAt" timestamp not null default CURRENT_TIMESTAMP, ' +
4   '"ColumnName" text, "SecondColumnName" integer,' +
5   'CONSTRAINT "Data_pk_TESTSCHEMA_TESTTABLE" PRIMARY KEY (id))'
```

---

**Abbildung 4.13:** Beispiel für ein Create-Statement

### parseInsertStatement

Da das PostgreSQL-Query dynamisch generiert ist, müssen dementsprechend auch die Anfragen zum Einfügen der Daten dynamisch generiert werden. Dafür ist die Funktion 'parseInsertStatement' zuständig.

```
1 async parseInsertStatement (
2   schema: any,
3   data: any,
4   pgSchemaName: string,
5   tableName: string,
6   parentId: number,
7   index: number = 0,
8   parentName: string = ''
9 ): Promise<string>
```

---

**Abbildung 4.14:** Funktion zur Generierung eines oder mehrerer Insert-Statements

Die Parameter 'schema', 'pgSchemaName', 'tableName', 'index' und 'parentName' erfüllen denselben Zweck wie bereits in 4.8.2 beschrieben worden ist. 'data' beinhaltet die transformierten Daten, welche in die Datenbank eingefügt werden sollen. Um beim Einfügen gewährleisten zu können, dass die Fremdschlüssel auf die richtigen Primärschlüssel verweisen, muss beim initialen Aufruf dieser Funktion die letzte verwendete ID in der durch 'tableName' definierten Tabelle übergeben werden. Außerdem werden hier zwei separate globale Variablen zur Zwischenspeicherung der SQL-Anfrage verwendet. Die erste Variable enthält den Beginn des Statements und die Spaltennamen, wobei die Zweite die einzufügenden Werte und das Ende des Statements beinhaltet.

Im ersten Schritt überprüft die Funktion 'parseInsertStatement', ob es sich bei den JSON-Daten um ein Array oder Objekt handelt. Dies ist notwendig, um das Schema und die Daten in den gleichen Ausgangszustand zu bringen. Wurde dies

determiniert, wird die entsprechende Funktion aufgerufen. Im Falle eines Objektes wird 'doParseInsertObject' verwendet. Hier werden die beiden oben genannten globalen Variablen beim aktuellen Index mit den jeweiligen Anfangsstrings initialisiert. Anschließend werden die im Schema definierten Schlüssel in einer Schleife abgearbeitet. Sind diese von einem primitiven Typen oder ein Array aus primitiven Typen, werden diese an die beiden Variablen angefügt. Sollte es sich um ein Array oder Objekt handeln, werden wieder die passenden Funktionen aufgerufen.

Die Funktion 'parseInsertArray' wird beim Auftreten eines Arraytypen aufgerufen. Hier werden ebenfalls die globalen Variablen für den aktuellen Index initialisiert. Da alle Daten eingefügt werden müssen, wird über jedes Element in dem übergebenen Daten iteriert. Die Abhandlung der verschiedenen Typen erfolgt analog zu dem in der 'parseInsertObject Funktion. Nach einem Iterationsschritt muss sowohl der 'index' als auch der 'parentId' inkrementiert werden. Dies muss asynchron erfolgen, um gewährleisten zu können, dass die Daten in der richtigen Reihenfolge eingefügt werden und somit die Fremdschlüssel garantiert auf die korrekten Primärschlüssel verweisen.

Im letzten Schritt werden die beiden globalen Variablen zusammengeführt. Um auch hier garantieren zu können, dass die Daten korrekt eingefügt werden, wird diese Anfrage als ein gesamter Transaktionsblock gestaltet. Dazu wird am Anfang aller Queries 'BEGIN' angefügt und am Ende 'END'. Die einzelnen Anfragen selbst sind jeweils durch ein Semikolon getrennt.

```

1  'BEGIN;' +
2  'INSERT INTO "TESTSCHEMA"."TESTTABLE" (' +
3  '"ColumnName","SecondColumnName")' +
4  ' VALUES (\text\', 69420)' +
5  ' RETURNING *;' +
6  'END;'

```

Abbildung 4.15: Beispiel für ein Insert-Statement

### 4.8.3 StorageStructureRepository

Diese Klasse verwaltet den Datenbankzugriff im Storage für die Erstellung der PostgreSQL-Schemata. Um hier die von der Schemagenerierung abhängigen Tabellen zu erstellen, wurde das Interface um die Funktion 'CreateForSchema' erweitert. Diese wird ausschließlich aufgerufen, wenn ein Schema vorhanden ist. Die vom JsonSchemaParser erstellten Anfragen werden in einer Schleife an die Datenbank gesendet.

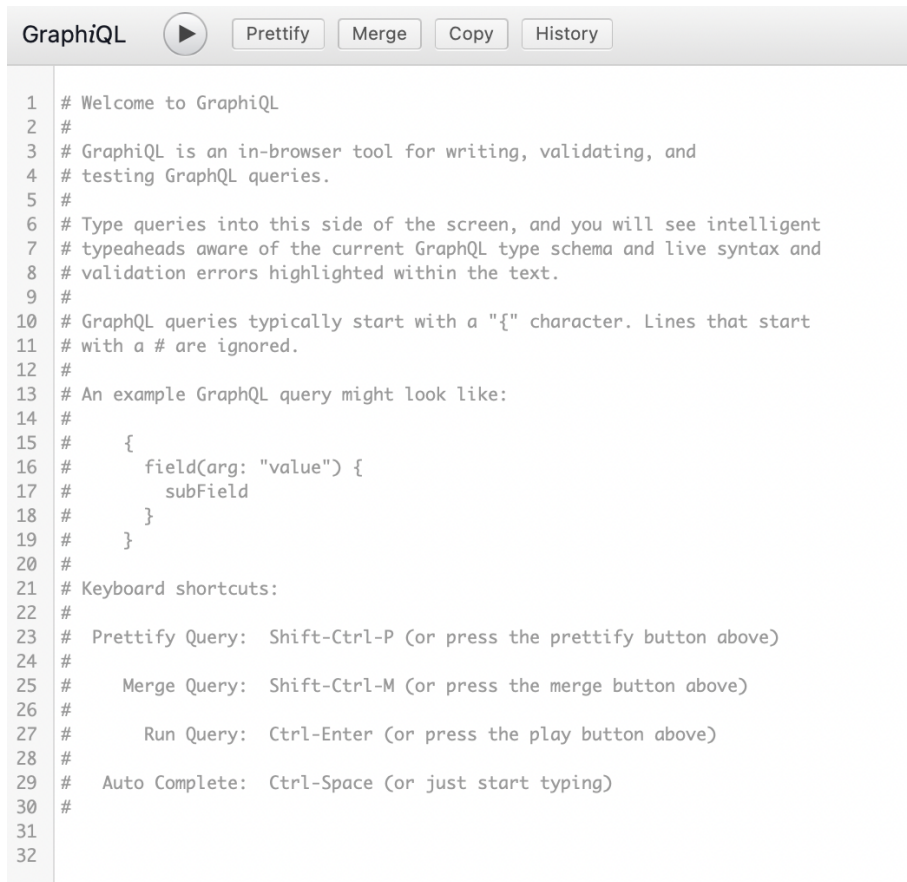
### 4.8.4 StorageContentRepository

Hier ist die Verwaltung für die Datenbankzugriffe, die für das Einfügen der Daten verantwortlich ist. Für die Schemaunterstützung wurde das Interface um die Funktion 'saveContentForSchema' eingeführt. Analog zum 'StorageStructureRepository' wird auch hier nur beim Vorhandensein eines Schema diese Funktion ausgeführt. Die Funktion fragt bei der Datenbank zunächst das letzte eingefügte Element in der relevanten Tabelle ab. Dadurch wird die nächste ID definiert und anschließend an die 'parseInsertStatement' Funktion übergeben. Die zurückgegebene Query wird nun als eine gesamte Transaktion an die Datenbank gesendet und ausgeführt.

### 4.8.5 Postgraphile

Dies ist die Schnittstelle für den Nutzer, um nun auf alle Daten die bisher in die Datenbank eingefügt worden sind, zuzugreifen. Dazu kann dieser die bereitgestellte API unter der Url 'http://<hostdomain>:5432/graphql' erreichen und Anfragen senden. Für mehr Informationen und bessere Übersicht über die zu Verfügung stehenden Tabellen und Daten kann zudem die bereitgestellt UI verwenden. Diese ist unter der Url 'http://<hostdomain>:5432/graphiql' erreichbar. Im Folgenden wird die Schnittstelle für eine bessere Visualisierung über die UI-Ansicht erläutert.

Beim Verbinden mit der UI wird dem Nutzer ein Texteditor, wie in Abbildung 4.16 zu sehen ist, angezeigt. Dieser übermittelt zu Beginn alle nötigen Grundkenntnisse für den Nutzer bezüglich der Querysprache GraphQL.



```

GraphiQL ▶ Prettify Merge Copy History
1 # Welcome to GraphiQL
2 #
3 # GraphiQL is an in-browser tool for writing, validating, and
4 # testing GraphQL queries.
5 #
6 # Type queries into this side of the screen, and you will see intelligent
7 # typeaheads aware of the current GraphQL type schema and live syntax and
8 # validation errors highlighted within the text.
9 #
10 # GraphQL queries typically start with a "{" character. Lines that start
11 # with a # are ignored.
12 #
13 # An example GraphQL query might look like:
14 #
15 #   {
16 #     field(arg: "value") {
17 #       subField
18 #     }
19 #   }
20 #
21 # Keyboard shortcuts:
22 #
23 # Prettify Query:  Shift-Ctrl-P (or press the prettify button above)
24 #
25 # Merge Query:    Shift-Ctrl-M (or press the merge button above)
26 #
27 # Run Query:      Ctrl-Enter (or press the play button above)
28 #
29 # Auto Complete:  Ctrl-Space (or just start typing)
30 #
31
32

```

Abbildung 4.16: PostGraphile initiale UI

Hier ist der Nutzer in der Lage nach der Syntax von GraphQL beliebige Query zu definieren und im Anschluss über den oberen Playbutton auszuführen. Der Editor selber verfügt über eine automatische Textvervollständigung und Syntaxvorschläge.



```

1 {
2   examplePipelineById(id: "1") {
3     v|
4   }
5 }
6
7
8

```

vorname  
String Self descriptive.

Abbildung 4.17: PostGraphile Syntaxvorschlag

Nun kann der Nutzer entscheiden, ob dieser gezielt über den Primärschlüssel auf einzelne Datensätze zugreift oder eine Anfrage über alle Dateneinträge. Um nun

## 4. Implementierung

---

auf einen einzigen Datensatz zuzugreifen, muss der Nutzer den Namen der Tabelle angeben und die gesuchte ID als Parameter übergeben.

```
1 {
2   examplePipelineById(id: "1") {
3     vorname
4     anzahl
5     position
6   }
7 }
8
9
10
11
12
13
14
15
```

```
{
  "data": {
    "examplePipelineById": {
      "vorname": "Ben",
      "anzahl": 33,
      "position": 1
    }
  }
}
```

**Abbildung 4.18:** Abfrage einer spezifischen ID

Die Anfrage, zu sehen in der Abbildung 4.18, gibt den Vornamen, die Anzahl und die Position eines Datensatzes zurück, welcher die ID eins hat. Des Weiteren kann der Nutzer auf alle Daten gleichzeitig zugreifen. Dabei kann dieser auch die Daten nach eigenem Belieben nach bestimmten Felder sortieren, bzw filtern.

```
1 {
2   allExamplePipelineIS(first: 5) {
3     nodes {
4       vorname
5     }
6   }
7 }
8
9
10
11
12
13
14
```

```
{
  "data": {
    "allExamplePipelineIS": {
      "nodes": [
        {
          "vorname": "Ben"
        },
        {
          "vorname": "Theo"
        },
        {
          "vorname": "Moritz"
        },
        {
          "vorname": "Fiete"
        },
        {
          "vorname": "Emil"
        }
      ]
    }
  }
}
```

**Abbildung 4.19:** Abfrage der Vornamen aller Daten

Das Beispiel aus Abbildung 4.19 zeigt auf, dass zum einen verschiedene Funktionalitäten der Anfrage als Parameter übergeben werden können. Zum Anderen ist hier aufgezeigt, wie alle Reihen aus der Spalte von ExamplePipeline gefunden und ausgegeben werden. Dies sind jedoch nur die groben Grundfunktionen von Post-

Graphile und somit bietet diese Library eine große Auswahl an Funktionalitäten und stellt somit eine sehr optimierte Schnittstelle für Queries dar.

## 4.9 Probleme und mögliche Lösungen

In diesem Abschnitt werden abschließend alle Probleme, die während der Entwicklung aufgekommen sind, beschrieben und mögliche Lösungen vorgestellt. Damit sollen zukünftige Entwicklungen reibungsloser und effizienter ablaufen.

### 4.9.1 Docker

Im Laufe der Arbeit traten einige Probleme im Bezug auf Docker auf. Die Implementierung erfolgte ausschließlich auf einem Windows 10 Rechner in Verbindung mit Docker Desktop WSL2. Bei dauerhafter Arbeit an der Applikation wurde durch Docker der Speicher immens gefüllt, bis zu teilweise hundertprozentiger Auslastung der Systemplatte. Das führte dazu, dass es nicht mehr möglich war, jegliche Container zu bauen, geschweige dessen diese zu starten. Um eine solche Auslastung zu vermeiden, konnte eine Verschiebung der virtuellen Platte durchgeführt werden. Zunächst muss WSL beendet werden

```
1 wsl --shutdown
```

---

Anschließend müssen die aktuellen Dockerdaten gesichert und exportiert werden.

```
1 wsl --export docker-desktop-data docker-desktop-data.tar
```

---

Im nächsten Schritt wird die Verbindung zur aktuellen virtuellen Platte aufgehoben werden.

```
1 wsl --unregister docker-desktop-data
```

---

Zuletzt muss ein Verzeichnis auf dem gewünschten Zielmedium angelegt werden. Anschließend kann die Verbindung zur neuen virtuellen Platte mit den zuvor gesicherten Daten angelegt werden.

```
1 wsl --import docker-desktop-data <Verzeichnis>/ docker-desktop-data.tar  
--version 2
```

---

## 4. Implementierung

---

Die angelegte 'docker-desktop-data.tar' Datei kann nun gelöscht werden, da diese nicht weiter benötigt wird.

Ein weiteres Problem war das Timing beim Starten der Container. Da viele Container in der Applikation voneinander abhängig sind und ein schwach performanter Rechner zur Entwicklung verwendet wurde, kam es zu teilweisen Verbindungsfehlern zwischen den einzelnen Dockercontainern. Dies konnte mit dem Tool 'Lazydocker' effizient beobachtet werden und bei Bedarf behoben werden.

```
Projekt
C:\WINDOWS\system32

-Container-
running open-data-service_graphql_1 7bbc1a8276e5e1e87f383a382
running open-data-service_adapter_1 e88c9fb00962e78b6155d5278
running open-data-service_pipeline_1 0af5baf4abac88c64094fa6c7
running open-data-service_storage_1 5c82ae29299936f01e4b1cc5e
running open-data-service_pipeline-outboxer_1 344b5bf612d8ffec2e7f95e1a
running open-data-service_scheduler_1 58ef89dc618158a7edeeead
running open-data-service_notification_1 0d4120753d18b993e04d70336
running open-data-service_adapter-outboxer_1 344b5bf612d8ffec2e7f95e1a
running open-data-service_storage-mq_1 3eb6e0efcfee3ba252527c27f
exited (0) open-data-service_storage-db-liquibase_1 96f41b58f89cf60254caebce5
running open-data-service_pipeline-db_1 d3a70afc848f415052a47e45
running open-data-service_edge_1 ddb0268e5b5fb0a4a9cb3f9e1
running open-data-service_schema_1 cd70f3869490133cd3658cf61
running open-data-service_ui_1 f54ab48fb951e0865fdc0205a
running open-data-service_adapter-db_1 d3a70afc848f415052a47e45
running open-data-service_rabbitmq_1 cc23b962e1c9ef390696ce83a
running open-data-service_adminer_1 203328a6b506bce7892dad315
running open-data-service_storage-db_1 d3a70afc848f415052a47e45
running open-data-service_notification-db_1 d3a70afc848f415052a47e45
```

Abbildung 4.20: Lazydocker

Hier können einzelne Container ausgewählt werden. Somit kann bei einem fehlerhaften Start diese gezielt ausgewählt werden und erneut gestartet werden.

### 4.9.2 Windows

Das Repository des Projekts ist für ein Linuxsystem ausgelegt. Das bedeutet, dass jegliche Dateien, die in diesem Versionierungsverzeichnis liegen, die Codierung 'LF' verwenden. Beim Klonen des Repositories auf einem Windowssystem wird teilweise für die Dateien CRLF verwendet. Dies hat dazu geführt, dass die Container 'storage-db-liquibase' und 'storage' aus dem Projekt nicht gestartet werden konnten. Dies konnte ebenfalls mit dem Tool 'Lazydocker' identifiziert werden.



```

Container
exited (127) open-data-service_storage_1          563d9b4ff4db4ef2d9af27f
exited (34)  open-data-service_graphql_1         073cf42d1027abe8341ff6a
running     open-data-service_storage-mq_1       12e74558a6c4755c6640b55
exited (1)  open-data-service_adapter_1         50dd8e0039887f4052893e5
running     open-data-service_pipeline_1        64f42813bfda41456fc2cb0
exited (1)  open-data-service_adapter-outboxer_1 344b5bf612d8ffec2e7f95e
exited (1)  open-data-service_pipeline-outboxer_1 344b5bf612d8ffec2e7f95e
running     open-data-service_notification_1     e6b131e53a371bdeba9244e
running     open-data-service_scheduler_1        2c9c4fcd79cb0bb0d2a7374
exited (127) open-data-service_storage-db-liquibase_1 e22bcc24f5703ed481ab00d
running     open-data-service_adminer_1          203328a6b506bce7892dad3
running     open-data-service_adapter-db_1        d3a70afc848f415052a47e
running     open-data-service_rabbitmq_1          cc23b962e1c9ef390696ce8
running     open-data-service_edge_1                ddb0268e5b5fb0a4a9cb3f9
running     open-data-service_schema_1              cd70f3869490133cd3658cf
running     open-data-service_ui_1                e5e417fea613a6237782db1
running     open-data-service_storage-db_1          d3a70afc848f415052a47e
running     open-data-service_pipeline-db_1         d3a70afc848f415052a47e
running     open-data-service_notification-db_1     d3a70afc848f415052a47e

```

Abbildung 4.21: Fehlerhafte Container erkenntlich über Lazydocker

```

-Protokoll - Statistiken - Konfiguration - Top-
2021-09-01T09:24:02.241891500Z /bin/sh: 1: /entrypoint.sh: not found
2021-09-01T09:24:19.055927800Z /bin/sh: 1: /entrypoint.sh: not found

```

Abbildung 4.22: Ausgabe eines fehlerhaften Containers

Im Rahmen der Arbeit konnte dieser Fall behoben werden, in dem die beiden Dateien 'entrypoint.sh' auf die LF Codierung geändert worden sind. Auf Dauer kann dies in der Dokumentation hinterlegt werden und diesen Schritt automatisch beim Klonen des Verzeichnisses konsequent auf LF umgestellt werden.

### 4.9.3 M1

Um eine angenehmere Arbeitsumgebung zu schaffen, wurde für diese Arbeit ein Apple MacBook mit M1-Chip angeschafft. Nachdem das Gerät vollständig eingerichtet wurde und die Repositories geklont worden sind, stellte sich heraus, dass einige Container nicht baubar sind. Hierbei ist der Auslöser dieses Problems der eingebaute M1-Chip. Dieser baut auf einer ARM-Architektur aus, welcher nach aktuellen Stand beim Kompilieren nicht unterstützt ist. Gelöst werden kann dies durch allgemeiner Anpassung aller betroffenen Dockerfiles. Dort kann man Baseimages verwenden, die weiterhin alle Funktionalitäten unterstützen und dennoch auch auf einer ARM-Architektur kompilierbar sind. Dazu muss auch das docker-compose.yml angepasst werden. Alternativ können ein separate Dockerfile angelegt werden. So muss der Entwickler beim Kompilieren seine Systemarchitektur kennen und manuell die passenden Dockerfiles verwenden.

## 4. Implementierung

---

# 5 Evaluierung

Im diesem Kapitel werden sowohl die getroffenen Designentscheidungen, als auch die Umsetzung der Anforderungen evaluiert.

## 5.1 Evaluierung des Architektur Designs

Im Kapitel 3.6 wurde das Design der Zielarchitektur festgelegt. Nach diesem Konzept sollte zunächst ein weiterer Service implementiert werden. Dieser kümmert sich um die Analyse von Daten und Generierung entsprechend gewünschter Schemata. Die Implementierung des Services war erfolgreich und zeigt gleichzeitig weiteres Potential dieser Implementierung bezüglich zukünftiger Entwicklung. Auf der anderen Seite ergaben sich Komplikationen in Verbindung mit Docker. Hierbei muss explizit beachtet werden, dass die beiden 'docker-compose.yml' Dateien beim Starten des ODS und Schemaservices, kombiniert werden. Nur so kann die API des Schemaservices von dem UI-Client erreicht werden. Schemagenerierung mittels der Ontology und Parsing in das geforderte JsonSchema-Schema funktionieren einwandfrei.

Das Design der UI sollte es ermöglichen, die Schemagenerierung optional verwenden zu können. Die Implementierung setzt das gezielt um und bietet dem Nutzer nach wie vor die Möglichkeit, das ODS wie vor der Erweiterung durch diese Arbeit zu nutzen. Das wird ermöglicht, indem der Nutzer einfach den Generierungsschritt überspringen kann. Startet der Schemaservice nicht, ist gemäß der Anforderung eines Microservices gewährleistet, dass alles unabhängig davon weiterhin problemlos funktioniert. Der Inhalt dieses Absatzes bezieht sich sowohl auf die Datenquellenkonfiguration, als auch auf die Pipelinekonfiguration.

Der Adapter ist für die Validierung der importierten Daten und bei einem gegebenenfalls vorhandenen Schema die Daten gegen dieses zu validieren, zuständig. Anhand der UI ist zu sehen, dass diese Schritte erfolgreich implementiert wurden. Eine erstellte Datenquellenkonfiguration zeigt zu jedem Zeitpunkt den aktuellen Zustand der importierten Daten an. Somit hat der Nutzer das gewünschte Feedback vom System, um entsprechend zu reagieren.

Die Pipeline soll analog zum Adapter arbeiten. Dazu sollte ähnlich zur Adapterklasse 'DataImport' eine Klasse 'TransformedData' eingeführt werden, um alle Metadaten bezüglich einer Datentransformation zu speichern. Die Validierung soll auf die Transformation der Daten angewandt werden. Zudem sollen bei einem gegebenenfalls vorhandenen Schema die Daten gegen dieses validiert werden. Hier kann ebenfalls anhand der UI bei einer erstellten Pipelinekonfiguration entnommen werden, dass die Designansprüche erfüllt wurden. Zur Anzeige des Zustandes der transformierten Daten muss das letzte erstellte 'TransformedData' verarbeitet werden. Da die Zustände korrekt und aktuell angezeigt werden, ist davon auszugehen, dass das Design erfüllt wurde.

Der Storage sollte in der Lage sein, bei Vorhandensein eines Schemas, dieses zu analysieren und daraus eine PostgreSQL Anfrage zur Erstellung und Einfügen der Daten zu erstellen. Dieses Verhalten funktioniert wie gewünscht. Dies kann anhand der Datenbank-UI 'adminer' observiert werden. Beim Angeben eines Schemas wird entsprechend dem Schema eine Tabelle in der Datenbank angelegt und anschließend im gewählten Intervall befüllt. Alles funktioniert vollkommen dynamisch, was sich beim Verwenden verschiedener Datenquellen erkenntlich zeigt.

Um den Nutzer auch die Möglichkeit zu geben auf diese Daten angenehm zugreifen zu können, soll eine Schnittstelle über 'Postgraphile' geboten werden. Nach der Konfiguration der Datenquellen und Pipelines ist es möglich, unter der URL des PostGraphile-Containers auf diese Daten mittels GraphQL-Syntax auszulesen und anzeigen zu lassen. Dabei hat der Nutzer die Wahl zwischen einer GUI und einer API.

## 5.2 Evaluierung anhand der Anforderung

Die Kapitel 2.3 und 2.4 definieren Anforderungen an die Arbeit und die Implementierung. Zusammenfassend waren insgesamt 19 Anforderungen definiert worden, von welchen 18 vollständig erfüllt und eine nicht erfüllt wurde. Für die Evaluierung werden schrittweise alle festgehaltenen Anforderungen betrachtet.

### 5.2.1 Anreicherung von Datenquellen mit einem Schema

Die nachfolgenden Punkte sind die Anforderungen zur Integration der Schemagenerierung in den Datenquellenprozess.

#### **Evaluierung passender Schemarepräsentationen**

Hierzu wurden zwei aktuell am weit verbreitetsten Schemarepräsentationen miteinander verglichen. Das Ergebnis aus dem Kapitel 3.3.3 zeigt, dass beide dem Nutzer die nötige Übersichtlichkeit und Verständnis bieten können. Aufgrund der besseren Testergebnisse in Sachen Performance wurde

JsonSchema sowohl als Übersichtlichere, als auch als effizientere Schema-repräsentation gewählt. Die für die Schemagenerierung konzipierte Lösung kann allerdings für weitere Schema-Sprachen erweitert werden.

### **Optionale Eingabemöglichkeit eines Schemas für Datenquellen**

Wie im Abschnitt 4.5.1 beschrieben wurde, steht dem Nutzer frei, ob dieser ein Schema angeben möchte oder nicht. Tut er dies jedoch nicht, so profitiert er nicht von dem automatisiertem Schema-Abgleich nach jedem Verarbeitungsschritt und bekommt letztendlich nur eine unoptimierte Query-Schnittstelle zur Verfügung gestellt.

### **Generierung eines Schema-Vorschlags**

Die Abschnitte 4.5.1 und 4.4 zeigen gemeinsam den gesamten Ablauf zu Generierung eines Schemavorschlags auf. Diese beginnt zunächst bei der Konfiguration der Datenquelle. Im entsprechenden Abschnitt kann der Nutzer anschließend ein Schema anfordern. Daraufhin analysiert der Schemaservice die erhaltenen Daten und sendet diese zurück an den UI-Client.

### **Optionale Veränderung des Vorschlags**

Diese Anforderung baut auf die obige evaluierte Schema-Generierung auf. Ist diese erfolgt, kann sich laut Abschnitt 4.5.1 der Nutzer entscheiden, ob der Schema-Vorschlag so angenommen wird oder von diesem angepasst wird.

### **(optional) Graphische Repräsentation des Schema**

Diese optionale Anforderung wurde nicht erfüllt. Jedoch wurde im 6.4 eine Möglichkeit zur Implementierung eines solchen Features elaboriert und vorgestellt.

## **5.2.2 Anreicherung von Pipelines mit einem Schema**

Die nachfolgenden Punkte sind die Anforderungen zur Integration der Schemagenerierung in den Pipelineprozess

### **Evaluierung einer passenden Datenbank**

Im Abschnitt 3.2 werden zunächst, um ein Grundverständnis zu schaffen, strukturierte und unstrukturierte Daten aufgezeigt und erklärt. Anschließend werden die strukturierten Daten als Form der Darstellung gewählt. Basierend auf den obigen Informationen wurde PostgreSQL als beste Wahl für diesen Anwendungsfall bestimmt.

### **Optionale Eingabemöglichkeit eines Schemas für Pipelines**

Wie in Abschnitt 4.5.2 beschrieben, ist dieser Schritt analog zu der gleichnamigen Anforderung für die Datenquelle zu sehen.

### **Generierung eines Schema-Vorschlags**

Wie in Abschnitt 4.5.2 beschrieben, ist dieser Schritt analog zu der gleichnamigen Anforderung für die Datenquelle zu sehen.

### **Optionale Veränderung des Vorschlags**

Wie in Abschnitt 4.5.2 beschrieben, ist dieser Schritt analog zu der gleichnamigen Anforderung für die Datenquelle zu sehen.

### **Transferierung der Daten in die Datenbank**

Die beiden Abschnitte 4.7 und 4.8 beschreiben den gesamten Prozess dieser Anforderung. Um diese zu erfüllen, muss neben dem Pipeline-Service auch der Storage-Service hinzugezogen werden. Dabei ergeben sich zwei zeitlich abgegrenzte Prozesse. Zum einen muss bei der Erstellung der Pipelinekonfiguration vom Storage-Service eine entsprechende Tabelle angelegt werden, wenn ein Schema verfügbar ist. Der zweite Prozess ist das Einfügen der transformierten Daten. Dabei werden im definierten Intervall die transformierten Daten in der Datenbank hinterlegt, welche dann vom Storage-Service ebenfalls analysiert werden. Basierend auf diese Analyse wird ein Insert-Statement erstellt und dieses dann ausgeführt. Somit erfüllen diese beiden Services die Anforderung vollständig.

## **5.2.3 Überprüfung der Schemakonformität**

Die nachfolgenden Punkte behandeln die Anforderungen bezüglich der Validierung der Schemata.

### **Healthindikator für Datenquellen**

Die Kapitel 4.5.1 und 4.6.4 beschreiben die Realisierung dieser Anforderung. Diese Anforderung wurde des Weiteren um die Kontrolle des allgemeinen Imports erweitert. So kann das ODS nun auch prüfen, ob der grundsätzliche Import der Daten funktioniert. Um das Schema zu prüfen, wurde im Adapter-Service ein Validator eingeführt, welcher die importierten Daten aus der Datenquelle gegen dem aktuell hinterlegten Schema testet. Die Zustände, resultierend aus erfolgreichem Importieren und Schemavalidieren, werden in einer Metadatenklasse hinterlegt und zusammen mit den importierten Daten in der Datenbank gespeichert. Der UI-Service kann die aktuellsten Importdaten anfordern und dort den Zustand der Datenquelle auslesen. Diese wird anschließend dem Nutzer als Statussymbol in der Datenquellenkonfiguration angezeigt

### **Healthindikator für Pipelines**

Diese Anforderung wird ausführlich von den Kapiteln 4.5.2 und 4.7.6 aufgezeigt. Um hier die Validierung analog zu der der Datenquelle implementieren zu können, mussten ähnlich zum Datenimport die Informationen und transformierten Daten in einem eigenen Modell abgespeichert werden. So-

mit kann die Validierung analog zur Datenquelle im Pipeline-Service ausgeführt werden und in der Datenbank hinterlegt werden. Der UI-Service kann sich die Metadaten zur letzten Datentransformation auslesen und den in diesen Daten hinterlegten Zustand auf der Website darstellen

### 5.2.4 Optimierte Query API

Die nachfolgenden Punkte behandeln die Anforderungen bezüglich einer optimierten Query API

#### Query auf Datensätze unterschiedlicher Zeitpunkte

Der Abschnitt 4.8.2 zeigt im Unterpunkt `parseCreateStatement` die Realisierung dieser Anforderung. Hierbei werden den eingefügten Daten immer ein Zeitstempel beigefügt. Diese werden automatisch vom System zugewiesen.

#### Filtern nach Feldern und Attributen

Diese Anforderungen werden im Kapitel 4.8.5 beschrieben. Mit der Library `postgraphile` ist es möglich, sowohl nach Feldern, als auch nach Attributen zu filtern, da diese Library vom Aspekt der möglichen Funktionen auf einer Ebene wie eine SQL-Query bietet.

### 5.2.5 Nicht-funktionelle Anforderungen

Die nachfolgenden Punkte behandeln die Anforderungen bezüglich aller Nicht-funktionellen Anforderungen.

#### Vermeidung von Code-Duplikaten

Diese Anforderung bzw. Aussage wird zum Beispiel im Abschnitt 4.6.4 behandelt. Um die Wartbarkeit zu erhöhen und Codeduplikate zu vermeiden, wurden wie in dem Beispiel des Validators ein Interface definiert. Diese stellen sicher, dass die übergeordneten Schichten nur von dieser Schnittstelle erfahren müssen. Jegliche Logik und Implementierung ist nicht mehr in den oberen Layern benötigt. Grundsätzlich wurde auf Code-Duplikate verzichtet, indem z. B. Funktionen, welche über mehrere Klassen hinweg verwendet werden, in eine gemeinsame Klasse aufgenommen werden. Somit kann sich jede Komponente diese Klasse importieren und ihre Funktionen verwenden. Somit ist die Logik der Helferfunktion an einer einzigen Stelle implementiert und erhöht so auch die Wartbarkeit, da nur dieser Code für weitere Änderungen angepasst werden muss.

### **Performance/Qualität für einen Schemavorschlag**

Die Punkte 4.5.1 oder 4.5.2 erfüllen dieses Requirement, indem dem Nutzer die Möglichkeit gegeben wird, selbst zu entscheiden, wie detailliert seine Daten für einen Schemavorschlag sein sollen. Je detaillierter die Antwort sein soll, desto langsamer kann das System werden. Gleiches gilt umgekehrt.

### **Erlernbarkeit**

Es wurde konsistent darauf geachtet, dass die Funktionen in dieser Arbeit möglichst eine Verantwortung haben und diese in einer überschaubaren Größe umgesetzt sind. Bei zu großen Funktionen wurden diese als Helferfunktionen ausgelagert. Des Weiteren wurden jederzeit ausschließlich aussagekräftige Namen für Funktionen und Variablen gewählt. Dadurch ist der Code sehr angenehm zu verstehen und somit für weitere Entwickler einfach, sich einzuarbeiten.



## 6 Ausblick

In diesem Abschnitt werden etwaige Punkte erläutert, welche im Laufe der Implementierung und Konzeption erkannt worden und im Anschluss an der Arbeit genutzt werden können, um die Applikation zu verbessern und zu erweitern.

### 6.1 Erweiterung des Schemavorschlags

In der aktuellen Ausführung werden die übergebenen Daten zunächst, je nach Präzisionseinstellung, zufällig ausgewählt und analysiert. Bei der Analyse auftretende, sich unterscheidende Attribute, werden gemeinsam dem übergeordneten Objekt oder Array als Properties zugewiesen. Zudem sind diese in jedem Fall unter 'required' aufgeführt, wodurch Validatoren kein Schema als gültig erachten, bei welchen diese Attribute fehlen.

Dies gibt die Möglichkeit, die Analyse auszuweiten und zu ermöglichen, dass auch optionale Attribute möglich sind. Das würde ermöglichen, dass Attribute nicht zwingend nötig sind, jedoch durch das Vorhandensein im Schema es trotzdem möglich wäre, diese in die Datenbank einzufügen. Um dies also realisieren zu können, muss sowohl die Schemagenerierung, als auch die Methodik beim Erstellen der beiden PostgreSQL-Anfragen angepasst und erweitert werden.

### 6.2 Erweiterung durch weitere Schemata

Da jede Implementierung in dieser Arbeit, welche mit der Verarbeitung oder Generierung von Schemata betraut ist, eine dementsprechende Schnittstelle erweitert, bietet das die Möglichkeit, auch weitere Schemarepräsentation einzubinden. Dazu müssen vereinfacht dargestellt lediglich die jeweiligen Schnittstellen mit den passenden neuen Schemaklassen implementiert werden. Anschließend müssen die entsprechenden Objekte mit diesen neuen Klassen initialisiert werden und können von diesem Zeitpunkt an vollständig verwendet werden.

### 6.3 Erweiterung durch verschiedene Validatoren

Wie auch die Schematagenerierungen ein Interface implementieren, wurde dies auch analog bei den Validatoren angewandt. Da die Möglichkeit der Verwendung von verschiedenen Schemarepräsentationsformen besteht, muss auch gewährleistet sein, dass die Validatoren einfach und unkompliziert passend ausgetauscht werden können. So muss der weitere Code nicht verändert werden. Durch das Interface wird auch ermöglicht, dass bei gleichbleibender Repräsentation verschiedene Validatoren genutzt werden können, um beispielsweise diese dynamisch je nach benötigter Strategie auszutauschen.

### 6.4 Nutzen der Ontology für alternative Zwecke

Die Ontology ist ein unabhängiger und eigenständig entwickelter Algorithmus, welcher ein Schema aus gegebenen JSON-Daten erstellt. Dies wird bereits verwendet, um davon ausgehend eine JsonSchema-Repräsentation zu erstellen. Durch diesen Zwischenschritt lassen sich weitere Anwendungsmöglichkeiten erarbeiten, wobei alle einheitlich von der Ontology ausgehen würden. Als Beispielszenario kann hier für den eher visuell orientierten Nutzer von Vorteil sein, den Schemavorschlag als Graphen vor sich zu sehen. Um dies zu erreichen, kann an die Ontology ein weiterer Parser angebunden werden, welcher diese in eine Datenform bringt, mit welcher anschließend die Daten visuell angezeigt werden können.

## Anhang A Beispiele der Ontologyanwendung

```
1  {  
2    First: 'Hello'  
3  }
```

---

Abbildung 6.1: JSON mit einem string Feld

```
1  {  
2    name: 'root',  
3    type: 'root',  
4    node: [  
5      {  
6        name: 'First',  
7        type: 'string',  
8        node: []  
9      }  
10   ]  
11  }
```

---

Abbildung 6.2: Ontology zum Beispiel JSON aus Abbildung 6.1

```
1  {  
2    First: {  
3      insideFirst: 'Hello'  
4    }  
5  }
```

---

Abbildung 6.3: JSON-Objekt mit einem Objekt

```
1  {
2    name: 'root',
3    type: 'root',
4    node: [
5      {
6        name: 'First',
7        type: 'object',
8        node: [
9          {
10         name: 'insideFirst',
11         type: 'string',
12         node: []
13       }
14     ]
15   }
16 ]
17 }
```

---

Abbildung 6.4: Ontology zum Beispiel JSON aus Abbildung 6.3

```
1  {
2    First: [
3      {
4        insideFirst: "text",
5        alsoInsideFirst: 10
6      }
7    ],
8    Second: [ "Hello", "It's", "Me" ]
9  }
```

---

Abbildung 6.5: JSON-Objekt mit zwei unterschiedlichen Arrays

```
1  {
2    name: 'root',
3    type: 'root',
4    node: [
5      {
6        name: 'First',
7        type: 'array',
8        node: [
9          {
10         name: 'insideFirst',
11         type: 'string',
12         node: []
13       },
14       {
15         name: 'alsoInsideFirst',
16         type: 'number',
17         node: []
18       }
19     ]
20   },
21   {
22     name: 'Second',
23     type: 'string[]',
24     node: []
25   }
26 ]
27 }
```

---

Abbildung 6.6: Ontology zum Beispiel JSON aus Abbildung 6.5



# Abbildungsverzeichnis

3.1	JsonSchema Beispiel Darstellung . . . . .	11
3.2	XSD Beispiel Darstellung . . . . .	12
3.3	Timing JSON vs XML [Izurieta, 2009] . . . . .	13
3.4	Resourcenauslastung JSON vs XML [Izurieta, 2009] . . . . .	13
3.5	Klassendiagramm der Ontologyklasse . . . . .	14
3.6	Besipel JSON-Objekt mit einem Array mit drei verschiedenen Objekten . . . . .	16
3.7	Schnittstelle für die aktuellsten transformierten Daten . . . . .	19
4.1	Funktion zur Erfassung von Objekten im JsonSchema . . . . .	25
4.2	JsonSchema für ein Objekt mit zwei string Feldern . . . . .	25
4.3	Funktion zur Erfassung von Arrays im JsonSchema . . . . .	26
4.4	JsonSchema für ein Array von Objekten mit zwei Attributen . . . . .	26
4.5	Form zur Anforderung und Bearbeitung eins Schemavorschlags . . . . .	27
4.6	Übersicht aktueller Datenquellenkonfigurationen . . . . .	28
4.7	Form zur Anforderung und Bearbeitung eins Schemavorschlags . . . . .	28
4.8	Übersicht aktueller Pipelinekonfigurationen . . . . .	29
4.9	Definition der Variable mit benutzerdefiniertem Typ . . . . .	30
4.10	Klasse für Metadaten einer Validierung . . . . .	31
4.11	Interface für transformierte Daten . . . . .	32
4.12	Funktion zur Generierung eines oder mehrerer Create-Statements . . . . .	35
4.13	Beispiel für ein Create-Statement . . . . .	36
4.14	Funktion zur Generierung eines oder mehrerer Insert-Statements . . . . .	36
4.15	Beispiel für ein Insert-Statement . . . . .	37
4.16	PostGraphile initiale UI . . . . .	39
4.17	PostGraphile Syntaxvorschlag . . . . .	39
4.18	Abfrage einer spezifischen ID . . . . .	40
4.19	Abfrage der Vornamen aller Daten . . . . .	40
4.20	Lazydocker . . . . .	42
4.21	Fehlerhafte Container erkenntlich über Lazydocker . . . . .	43
4.22	Ausgabe eines fehlerhaften Containers . . . . .	43

6.1	JSON mit einem string Feld . . . . .	53
6.2	Ontology zum Beispiel JSON aus Abbildung 6.1 . . . . .	53
6.3	JSON-Objekt mit einem Objekt . . . . .	53
6.4	Ontology zum Beispiel JSON aus Abbildung 6.3 . . . . .	54
6.5	JSON-Objekt mit zwei unterschiedlichen Arrays . . . . .	54
6.6	Ontology zum Beispiel JSON aus Abbildung 6.5 . . . . .	55



# Literaturverzeichnis

- ajv*. (2021). Verfügbar 30. August 2021 unter <https://github.com/ajv-validator/ajv>
- Douglas, K. D. S. (Februar 2003). *PostgreSQL*.
- everit*. (2021). Verfügbar 30. August 2021 unter <https://github.com/everit-org/json-schema>
- Foundation, T. G. (2021). *GraphQL*. Verfügbar 30. August 2021 unter <https://www.graphql.org>
- GSON*. (2021). Verfügbar 30. August 2021 unter <https://github.com/google/gson>
- Izurieta, N. N. M. P. R. R. C. (2009). Comparison of JSON and XML Data Interchange Formats. <https://www.cs.montana.edu/izurieta/pubs/caine2009.pdf>
- Jablonski, R. H. S. (2011). *NoSQL evaluation*. Verfügbar 30. August 2021 unter <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6138544>
- Klößner, K. (Dezember 2015). *Im Vergleich: NoSQL vs. relationale Datenbanken*. Verfügbar 30. August 2021 unter [https://pi.informatik.uni-siegen.de/Mitarbeiter/mrindt/Lehre/Seminare/NoSQL/einreichungen/NOSQLTEC-2015\\_paper\\_9.pdf](https://pi.informatik.uni-siegen.de/Mitarbeiter/mrindt/Lehre/Seminare/NoSQL/einreichungen/NOSQLTEC-2015_paper_9.pdf)
- ODS-Team. (2020). *ODS Github* [Date Accessed: 20-04-2020]. Verfügbar 30. August 2021 unter <https://github.com/jvalue/open-data-service>
- Team, T. P. (2021). *PostGraphile*. Verfügbar 30. August 2021 unter <https://www.graphile.org>
- Vennam, S. D. N. V. D. K. E. C. M. F. D. G. V. G. M. G. S. J. V. L. M. M. S. N. R. (August 2005). *Microservices from Theory to Practice*. <https://www.redbooks.ibm.com/redbooks/pdfs/sg248275.pdf>
- Wilson, D. C. (2021). *Express*. Verfügbar 30. August 2021 unter <https://www.npmjs.com/package/express>
- Yergeau, T. B. J. P. C. M. E. M. F. (November 2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). <https://www.w3.org/TR/REC-xml/>