Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät
Department Informatik

Kai Malte von Rönne

BACHELOR THESIS

GraphQL-based generic and domain specific query interfaces for the JValue ODS

Submitted on 08.10.2020

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.

Professur für Open-Source-Software

Department Informatik, Technische Fakultät

Friedrich-Alexander University Erlangen-Nürnberg

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.

_____

Nürnberg, 8.10.2020

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other that the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. Furthermore, this thesis has not been submitted to any other auditing authority in the same or similar form and has not been accepted by them as part of an examination paper.

_____

Nürnberg, 8.10.2020

License

_____

Nürnberg, 8.10.2020

Abstract

The JValue Open Data Service (ODS) is an open source application, founded and developed by the professorship of Open Source Software of the Friedrich Alexander Universität Erlangen Nürnberg (FAU). The application aims to become the go-to place for developers and data scientists around the world when building applications by using Open Data. The JValue ODS' business model consists of collecting data from various different Open Data APIs, apply processing and cleansing and offering the improved version of the data back to the user. At the time of writing the thesis, the ODS only offers a rudimentary auto generated REST Interface for accessing the resulting data which does not fulfill the various requirements of its targeted user base. Therefore, this thesis aims to offer a solution on how a responsive and clean user API could be implemented using GraphQL as query language. As the ODS itself is a collection of many different docker native micro-services, the relevant functionality for providing the APIs is developed using the same core principles.

In order to grant the user the ability to fetch the raw data, each creation of a new pipeline, which configures the result of the attached data endpoint and a customizable transformation script, can be further configured to generate a read only GraphQL endpoint on the basis of the underlying PostgreSQL schema. This API is capable of filtering and combining all the fields of the collected data. To give the users, who are more knowledgeable in the domain they are working in, the possibility to add business logic to this data a node.js template project is provided. After implementing the desired functionality and hosting it on their preferred hosting solution, the users can register their custom endpoint through the JValue ODS Web Interface which are then validated and merged with the existing schema.

Content

List of abbreviations

AGPLv3 ..... GNU Affero General Public License

API ..... Application Programming Interface

FAU ..... Friedrich-Alexander Universität

HTTP ..... Hypertext Transfer Protocol

JMA ..... Japan Meteorological Agency

JSON ..... JavaScript Object Notation

Massachusetts Institute of Technology ..... MIT

ODS ..... Open Data Service

REST ..... Representational State Transfer

URL ..... Uniform Resource Locator

User Interface ..... UI

WSV Wasserstraße- und Schiffahrtsverwaltung des Bundes (Federal Waterways and Shipping Administration)

1 Introduction

The modern world is data driven. This statement is now more relevant than ever before as every major industry is in the process of incorporating more technology into their daily business workflows. The digitally captured key indicators which are generated in every step of a supply chain help large companies, for example in the automotive industry, pinpoint some missed resource optimization and help fixing this issue. In another part of the world, the data generated by the sensors of the JMA in Japan helps the country to keep its residents safe by being able to track seismological shifts and raise alert in case of an impending (Japan Meteorological Agency, n.d.). Developers around the world have access to this data and can use it to write their own applications to further increase security and awareness. This is just one of countless examples in which data drastically changes the current way of living.

In recent years, the concept of open data has started to gain more popularity and large amounts of supporters. Between 2018 and 2030 the European Commission has projected in their Impact Assessment Support Study for the Revision of the Public Sector Information that the total direct economic value of Public Sector Information (PSI) will increase from 52 billion Euros to 194 billion euros earthquake (Data Policy and Innovation Unit G. 1, 2020)**.** **"**Open Data can be freely used, modified, and shared by anyone for any purpose and is marked as open by being provided under a conformal license", claims Riehle (2019). These characteristics allow people who are interested to use the vast amount of generated data of different fields and economic disciplines to get their best personal use out of it. The data can be used just out of curiosity but also as a foundation to build new, innovative and life changing applications.

The JValue ODS, developed by members of the Open Source professorship under the guidance of Prof. Dr. Dirk Riehle, aims to be an enabler for other people interested in open data in this context by providing a software platform that connects to various different streams of open data and makes it accessible from one place.

So far the JValue ODS is still missing an intuitive and a future proof way of making this transformed data accessible via a fitting API concept, which can be extended by its users and community. This thesis aims to give the ODS the basic function of such an API by implementing this feature using Facebook's GraphQL[1] specification which focuses on efficient and logical data retrieval.

[1] https://graphql.org/

2 Requirements

2.1 Motivation

As the community around the ODS aims to "[…] establish the ODS as the go-to place for using Open Data (Schwarz, et al., 2020)", it does not only need a way to make the transformed data available to its users, but also a deliberate future proof concept and a concrete reference implementation on how such a claim can be achieved. Next to the functionality a software offers, a major issue on whether a software is adopted by the targeted user-group heavily relies on the way the user can interact with the system (PICCIONI, FURIA, & MEYER, n.d.).

The following requirements for implementing such a system have been collected by firstly talking to current developers of the system to make sure the proposed concept fits into the targeted vision of the project. Secondly, interviews have been conducted with several selected users of the JValue ODS to take their feedback from regularly working with the system from the consumer viewpoint into account.

2.2 Functional requirements

Malan and Bredemeyer define functional requirements as "[…] functional requirements capture the intended behavior of the system. This behavior may be expressed as services, tasks or functions the system is required to perform" (MALAN & BREDEMEYER, 1999). In this case the functional requirements specify the scope of the GraphQL-API feature of the ODS and the main focus points.

2.2.1 Automatic generation of basic API

The solution should be able to automatically generate a working API Endpoint that exposes the raw data collected from a JValue ODS Pipeline to the user. This endpoint should be the first contact point of users with the data before they continue using, altering, and interpreting it.

2.2.2 Navigating between snapshots of the data set

The API should give the user the possibility to jump from one snapshot to the previous or following snapshot. This requirement should emphasize the sequencing nature of Open Data and enable users to easily compare snapshots of data sets at different points in time.

2.2.3 Interface to other components

The API should get their data directly from the storage databases. No intermediate data layer is planned as of now.

2.2.4 Query language

The query language for fetching the data is supposed to be GraphQL. Its features provide the user with a complete and understandable description of the data in the API (GraphQL Foundation, n.d.).

2.2.5 Extensibility through the JValue ODS community

The concept should give the community of the JValue ODS the possibility to extend the provided API with existing API endpoints on top of the existing data and share them with others. This should enable users to enrich the raw data with customized business logic.

2.2.6 Reasonable time to set up API

The solution should be able to set up the API in a reasonable amount of time. The API should be available to be queried within ten seconds after a new pipeline has been created or the feature has been enabled on an existing pipeline.

2.2.7 User interface

Users should be able to configure the API generation and extension workflow via the JValue ODS user interface. It should be available to choose whether the public API feature is turned on or off and to configure the extensions of the public API with user provided functionality.

2.3 Non-functional requirements

2.3.1 Adaptability

The solution should be adaptable and applicable to changes in the future. This applies to the used programming language, libraries, and standards in particular.

2.3.2 Licensing

The licenses of all used components should not conflict with the AGPLv3 license of the JValue ODS. Only external components licensed under the MIT or Apache License 2.0 should be used. These licenses are considered as very liberal. They are written permissively and allow the users to use components distributed under these licenses to use them for any use-case (MIT, n.d.) (Apache Software Foundation, 2020).

2.3.3 Deployment

All parts of the solution that are included in the final code base of the JValue ODS should be easily deployable in a controlled environment. As most of the micro services of the JValue ODS use docker for this task, the solution should be able to use docker as well, unless there is a sound reasoning for not doing so.

2.4 Stakeholder

Developing software in such a wide field of use, for example in the JValue ODS with many types of stakeholders, often has potential of conflicts of interest. Different stakeholders have different priorities as they usually focus on different characteristics of the software. Key stakeholders have been identified to reduce this complexity following

2.4.1 Developers

Developers of the JValue ODS are responsible for building and maintaining the software in the GitHub repository[2]. They are responsible for keeping their code base clean and organized. Developers should have a strong interest for the factual implementation and for the technology in use.

2.4.2 Users

The users of the JValue ODS are mainly other developers who use ODS as a basis to build their own applications. Since the data provided by the JValue ODS often is an integral part of their products, they focus on the reliability of the system, available features and performance.

2.5 Evaluation

The described functional and non-functional requirements in the chapters 2.2 and 2.3 are the basis against which the final result is being evaluated. Furthermore, this thesis also documents the progress of implementing the feature for the JValue ODS production code base. All points discussed in this thesis can be found in the linked GitHub pull request[3]. The final decision on whether this feature will be integrated in the productive code base of the JValue ODS belongs to the developing community around the project.

[2] https://github.com/jvalue/open-data-service
[3] https://github.com/jvalue/open-data-service/pull/189

3 API design and GraphQL as a query language

3.1 Characteristics of a good API

The API defines of how other programmers can interact with your software. Therefore, the importance for the success of any project that relies on other developers to integrate their software in the product's ecosystem cannot be understated. A good API can certainly make the difference on whether other developers choose to accept to work with your system or rather search for an alternative (BLANCHETTE, 2008, pg. 7-13).

Over the history of software development some high-level key characteristics have evolved that are alike with most good APIs. These characteristics have been collected and written down by Joshua Bloch, principal software engineer at Google Inc. (BLOCH, n.d.).

3.1.1 Easy to learn and memorize

Quantifying what is considered 'easy to learn' is difficult but there are some patterns that should be featured. Jasmin Blanchette describes this characteristic as follows: "An easy-to-learn API features consistent naming conventions and patterns, economy of concepts, and predictability. It uses the same name for the same concept, and different names for different concepts" (BLANCHETTE, 2008).

Furthermore, the API should be consistent within itself and repeat the same patterns in its different parts so the user can reapply the same concepts. The semantics should be coherent and simple. The API should require as little boilerplate code as possible to get started.

3.1.2 Hard to misuse

The API should guide the user into the right direction by default and encourage the developer on using standard programming best practices. It should reduce implicit side effects that the user is not aware off and should not be strict order dependent when there is no need.

3.1.3 Easy to extend

Most APIs mature with time and functionality either increases or at the very least changes. A good API design should consider the life cycle of the application in the design process and take measure to ensure that it can be adapted with as little effort as possible. Changes that require the users to alter their code should be minimized. This is how users avoid having to maintain an array of different API versions.

3.1.4 Complete

Ideally the API should be complete in the sense that it can fulfill all possible use cases of its users. As this is often not possible to quantify in advance, a good API gives the user the

possibility to extend and customize the API. “Completeness is also something that can appear over time, by incrementally adding functionality to existing APIs”, referring back to the extendibility of a good API (BLANCHETTE, 2008).

3.1.5 Appropriate to audience

As every good software keeps in mind the audience it caters to, the same applies to a well-designed API. Recognizing the users, their main goals and motivation can help the process of API design tremendously. In the end it is most often the users that decide whether the API is good or bad and suited for the specific use case.

3.2 GraphQL as a query language

GraphQL is a specification that has been developed by Facebook and open sourced to the public in the year 2015. Currently GraphQL is maintained by the GraphQL foundation, a neutral foundation founded by global technology and application development companies. This foundation encourages contributions, stewardship, and shared investments from a broad group in vendor-neutral events, documentation, tools, and support for GraphQL (GraphQL Foundation, n.d.). The release included the current specification and a reference implementation. Developed out of a need for a querying interface that can handle the massive amount of API calls of their infrastructure, developers of Facebook quickly realized the big potential of GraphQL. It can be a well-suited alternative for REST APIs, especially for use cases that require a vast amount of data fetching. GraphQL does this by allowing the users to directly define which data they exactly need in only one call and therefore reducing the total amount of data transferred by reducing the overhead of each API call. The nature of a Graph hereby allows GraphQL to follow relationships between objects. In contrast a REST service would need multiple roundtrips while sending more data than what might be needed by the caller (GraphQL Foundation, n.d.).

3.2.1 Basic principles of GraphQL

To understand how GraphQL is an improvement to existing API standards, such as REST, one must understand how the set-up of GraphQL and how the query language and type system work. Firstly, a GraphQL service is created by defining the individual types and fields for the existing data structures. Then functions, named resolvers in GraphQL lingo, are implemented that return the defined fields on each type (GraphQL Foundation, n.d.).

In this basic example two types called Query and Planet are created. The field Planet is assigned to the Query. The type Planet itself has the properties ID, name, radius, orbital period and volume with the corresponding data types.

```
type Query {
  planet: Planet
}


type Planet {
  id: ID
  name: String
  radius: float
  orbital_period: float
  volume: float
}
```

To allow users to interact with these types, resolvers for the individual fields on the types must be implemented.

```
  function Query_planet (request) {
        return db.get(request.param.planetName)
  }

  function Planet_name(planet) {
        return planet.name
  }

  function Planet_radius(planet) {
        return planet.circ / (2 * Constants.pi)
  }
```

When this server is up and running, users can send queries against the URL of the server and ask for data. Queries are just plain JSON strings that reference the different type and fields and can thereby specify what exact data is needed (GraphQL Foundation, n.d.).

An example for the service described above could look like this:

```
  {
        Saturn {
              name
              radius
        }
  }
```

The returning result is also a JSON string, containing all the data that has been queried for:

```
{
    “Saturn”: {
            name: “Saturn”
            radius: 58.232
    }
}
```

The response omits all the other fields of the type Planet, such as the ID, the orbital period, and the volume of the Planet. This is the core mechanic that allows a decluttered and smooth user experience by only serving data to the user that has been explicitly asked for.

3.3 Core features

To understand in what other ways GraphQL differs from existing webservice solutions, chapter 3.3 describes the details of the most important features of GraphQL.

3.3.1 All data needed with one call

GraphQL typically provides one endpoint to consolidate all the data to get rid of the need to combine the data fetched from individual endpoints. The main goal is to reduce the amount of overhead that increases with the number of requests and serve the data quickly and responsively (BEZUGLA, 2019).

3.3.2 No over- and under-fetching problems

As the user specifies the fields on the types he wants to receive, they are only served exactly the data that they will need for their use-case. This reduces the workload on the transfer medium and increases the efficiency of each call (BEZUGLA, 2019).

3.3.3 Validation and type checking by design

The inherent type system of GraphQL allows the user to offload the type checking to GraphQL. This system specifies the availability of types, the properties on each type and the relationship between the different objects (BEZUGLA, 2019).

3.3.4 In-built API maturing and versioning

In contrast to many other systems, that rely on providing different versions of their API to not break systems built on a previous APIs version, GraphQL solves this issue by deprecating APIs on a field level. This means that new fields can be easily added, and older fields slowly fade out without the need for users to immediately adapt to the changes.

Fields that have been replaced with other field will not interfere with any clients as they have not specified the new field in their existing queries. Therefore, they will not feel the change on the application level (BEZUGLA, 2019).

3.3.5 Detailed error messages

GraphQL gives users an easy default way to handle errors by providing a detailed error message in the returning payload so that users do not need to manually check for the returning status code. However, as GraphQL does not enforce a specific standard; users can also choose to build their own application-specific error code system if they wish to do so (BEZUGLA, 2019).

3.4 GraphQL compared to REST

As REST is currently in fact the default paradigm on how webservices are to be written and consumed table 1 compares the key features between GraphQL and REST.

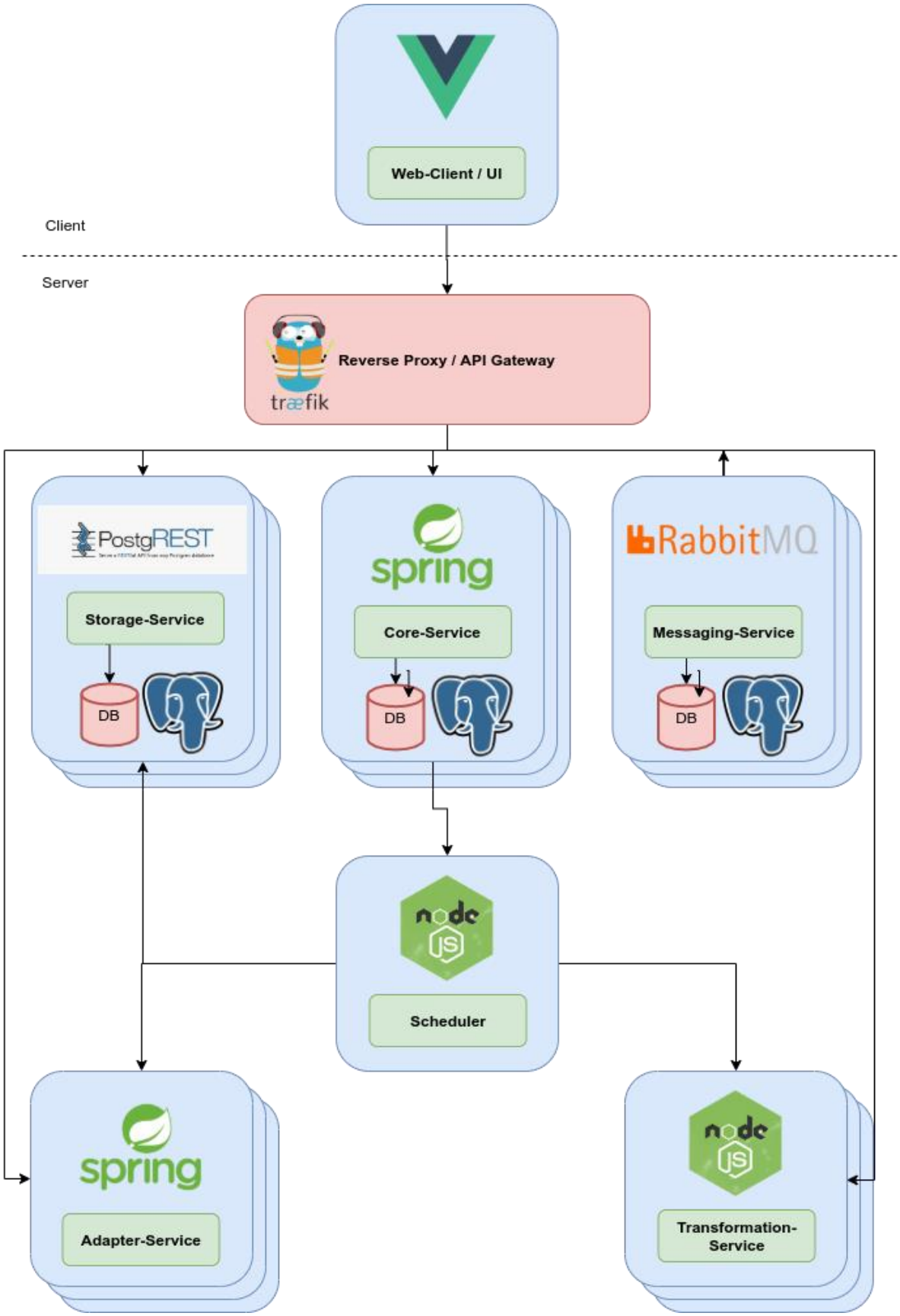
Table 1 - Comparison between REST and GraphQL. Von Rönne (2020)

	GraphQL	REST
Architecture	client-driven	server-driven
Data fetching	specific data with one API call	fixed data with multiple calls
Stateless	yes	yes
Caching	no	yes
Operations	query, mutation, subscription	CRUD
File uploading	no	yes

The core difference between GraphQL and REST is that REST API is an architectural concept for network-based software. GraphQL on the other hand is a comprehensible query language, a specification and a set of tools that operate over a single endpoint using HTTP (Russell, 2019). The main advantage of REST is that one can always be sure to receive the complete data set with each call. If data of two objects is required, the client has to make two separate calls. This concept might be often ineffective but is inherently simplistic and easily understandable. GraphQL, however, was created with performance and flexibility in mind. This performance increase leads to a steeper learning curve but rewards the user with a faster development workflow later.

4 Architecture and design

Figure 1 - Adapted overview of JValue ODS architecture. Schwarz et al. (2020)

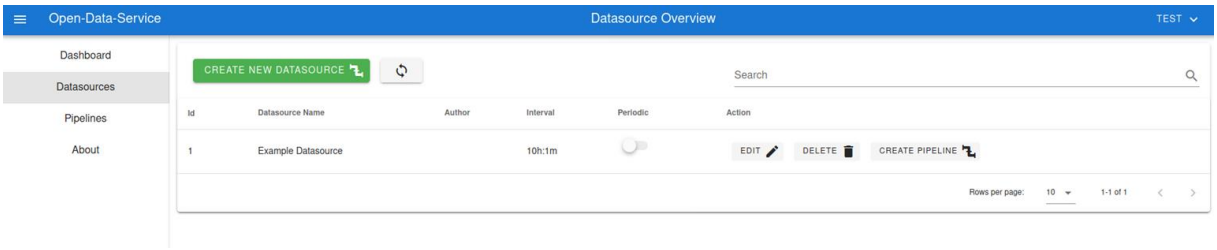


As the thesis aims to implement a viable solution on how GraphQL API could be fast and used easily by the JValue ODS, the design and architecture of the project must be analyzed. The JValue ODS is built on a micro service oriented architectural style with individually defined interfaces. They can be used to communicate between each other during runtime. Furthermore, each micro service is built on a cloud-native architecture in order to allow for individual scaling. During the creation process of this thesis the communication between the individual components was transformed from REST communication to a message broker supported by RabbitMQ (SCHWARZ, et al., 2020).

Web-Client / UI

This service serves the customer the user interface. Built with VueJS[4], a popular Javascript user interface framework, this web application allows the user to add new sources and pipelines or configure existing sources to their needs. The main entry point for a new user is the declaration of a new data source. The data source configuration allows the user to configure what open API is being called and what its characteristics are. For calling the API, the JValue ODS needs to take several meta information about the API such as the used protocol, encoding, format and the URL of the endpoint.

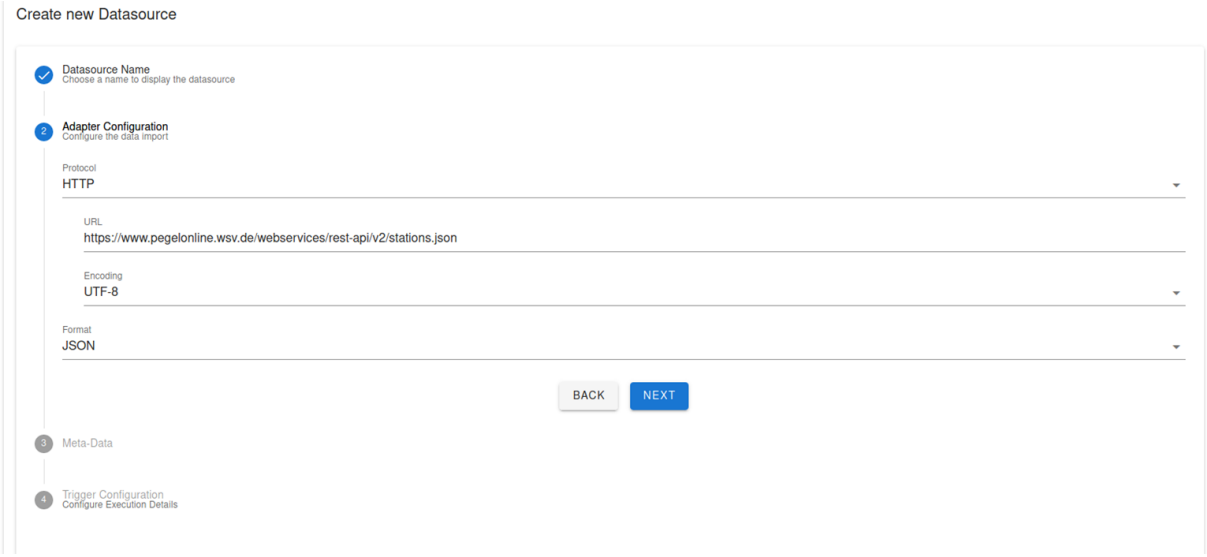
Figure 2 - User interface of the JValue ODS. Von Rönne (2020)



In addition, the user can also configure when and how often the data is being collected and store additional meta data about the pipeline such as a description, name of the author and license.

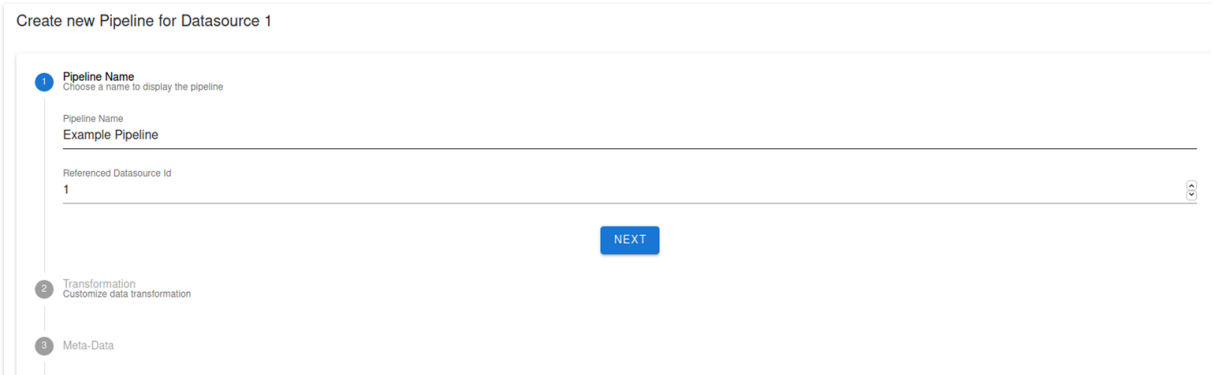
[4] https://vuejs.org/

Figure 3 - User interface: Create new data source form. Von Rönne (2020)



After the data source has been created, the user can then choose to create a pipeline. A pipeline consists of a connected data source and a transformation function written in Javascript that can be used to alter the plain data returned by the data source, e.g. by adding additional fields. Each pipeline has like a data source connected meta field storing the description, author, and license.

Figure 4 - User interface: Create new pipeline form. Von Rönne (2020)



Scheduler

This service is responsible for the orchestration of the pipeline's execution. Written in nodeJS[5], this component takes messages from the UI on creation of data sources and orchestrates the execution of the different data sources with the help of cron[6] jobs and an internal map of jobs that are to be executed. When a job is ready to be executed, the scheduler component propagates

[5] https://nodejs.org/en/
[6] Time-based job scheduler in Unix-like operating systems

this to the adapter service via a rest call.

Adapter-Service

This service handles the connection to the attached APIs by the user. The service is written in Java on top of the commonly known Spring framework. The service is being triggered by the scheduler. Once triggered, it utilizes the configuration provided by the user to call the endpoint and interprets the results. On successful import execution, the data is then stored as a DataBlob in a PostgreSQL[7] database.

Transformation-Service

Written in nodeJS, this service handles the execution of a pipeline with its specified execution transformation function. Using the data blob stored by the adapter service, the component tries to apply the user supplied Javascript function to alter the given data. After successful execution the new data blob is forwarded to consecutive services.

Notification-Service

This service is responsible for handling notifications of the software by accepting various configurations, accessible from the user interface. As of now support for three different types of notification have been added. Firebase, Slack and generic web hook notifications. Whenever an event that fulfills the requirements of a notification configuration has been received, this service sends out the proper notification to the specified subscriber.

Storage-Service

The storage of the data and configuration of the JValue ODS is done via PostgreSQL databases. The storage service is responsible for storing the data of the executed pipelines and providing the current REST API that can be used to access the data automatically. In addition, this service handles the changes to its database via Liquibase[8], a common tool for database version control.

[7] https://www.postgresql.org/
[8] https://www.liquibase.org/

Reverse-Proxy

The JValue ODS uses Traefik[9] as a reverse proxy solution to handle requests to the application. In short, Traefik is configured to work as a sort of API Gateway that hides the complexity of the backend system by providing API forwarding requests to the right place. Each container can then be further configured to be able to access incoming or outgoing traffic to the network (traefiklabs, n.d.).

Deployment

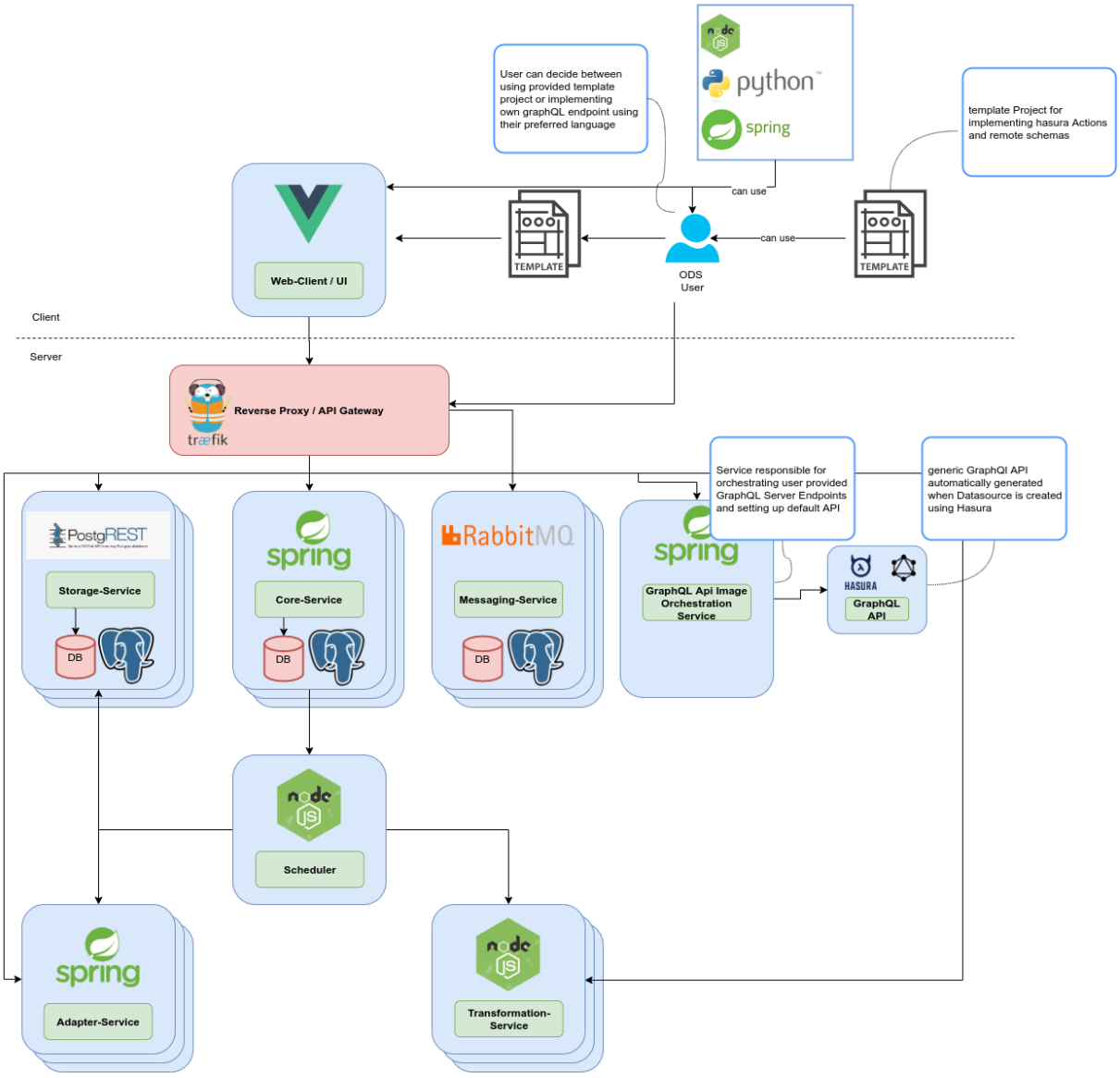
The ODS is designed as a cloud-native application. Each service is dockerized and individually deployable. The final reconciliation is done via docker-compose. Hosting can be done by any platform that is able to run docker and is able to provide the minimal hardware requirements of each component. Currently no publicly addressable instance is being deployed.

[9] https://docs.traefik.io/

4.1 Proposed solution

4.1.1 Hasura GraphQL engine

Figure 5 - Adapted overview of JValue ODS architecture. Schwarz et al. (2020)



The proposed solution consists of three key areas: the API orchestration service, the running API service and the template project provided to users to add additional business logic to the API. All these components are built within the Hasura Open Source GraphQL Engine[10] ecosystem, a project which accelerates serving production ready GraphQL APIs.

[10] https://hasura.io/opensource/

4.1.2 Hasura GraphQL engine

Hasura GraphQL engine is an open source project developed by the company Hasura which provides developers with a tool set to set up production ready GraphQL APIs on top of their PostgreSQL databases. The project has been released to the public on August 6th, 2018 under the Apache license v2 and has since gathered a thriving community on GitHub with nearly 19.000 users starring the repository. Since its release the software has been widely adapted in the software industry and has even found its way into the code base of Fortune500 companies, the largest 500 companies in the United States according to the renowned magazine Fortune (Hasura, n.d.). On the 8th of September 2020 Hasura announced the closing of their Series B financing with $25 million US dollars, led by the company Lightspeed with participation of renowned investors such as Vertex, Nexus, SAP.io and Strive. This investment round was closed only six months after the $9.9M Series A round and is an indicator on how valuable the services provided by Hasura are to the tech sector. With announcing the investment, Hasura also announced that their services have been extended to work with MySQL databases and therefore tapping up another big sector database (LARDINOIS, 2020).

The engine provides several advantages and fledged out features that enables the JValue ODS to quickly transform its existing REST API to GraphQL; a highly complex and time enduring task that would otherwise capture a major part of the available human resources.

Following features have been identified as the most relevant for the JValue ODS.

Automatic API generation

Hasura can automatically generate a production ready API based on a PostgreSQL database instance. This feature is key for the ODS as the gathered data from the different APIs can have an arbitrary structure, with the only mutuality being that the data is later stored inside a jsonb array field in a PostgreSQL database by the storage service (Hasura, n.d.).

Figure 6 - Storage database schema. Von Rönne (2020)

Column	Type	Comment
id	bigint	
data	jsonb	
timestamp	timestamp *NULL*	
pipelineId	character varying *NULL*	

Provision of CRUD Queries

Hasura automatically generates all necessary queries to access each field of the database. Therefore, it is immediately possible to filter for all existing field once the API is set up and running. This greatly reduces the amount of work that has to be done after adding new pipelines to the JValue ODS and will enable the software to quickly spin up an API after a new pipeline has been created (Hasura, n.d.).

Dynamic access control

Hasura allows for a dynamic access control that can be integrated with the existing access control of the JValue ODS. As currently the primary use case of the JValue ODS only specifies a read only API, this requirement can be fulfilled by providing role-based schema that restricts the access to the API (Hasura, n.d.).

Off-site business logic

Hasura offers two ways in which additional logic can be inserted into the auto generated schema. Firstly, new logic can be added through 'Hasura Actions'. Here the custom queries and mutations are defined by the user and subsequently backed by an already existing REST API. The REST API does not have to run in the same infrastructure context as the remaining services of the JValue ODS but could be hosted on the users' private hardware or even as a server-less function.

The second way in which the GraphQL API can be expanded is by merging Remote GraphQL schemas together. In this variant the user could write a complete standalone GraphQL API that uses the generic GraphQL API to access and work with the existing data and then merge the added functionality into the default API provided by the ODS (Hasura, n.d.).

Scalability

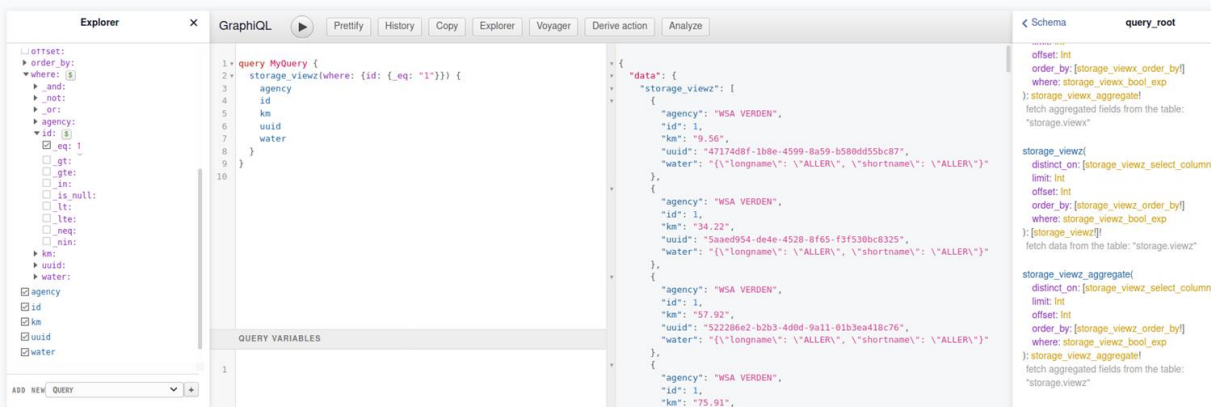
Hasura is by default designed to only use as many resources as needed while still being able to quickly scale as the number of requests increases. As the Hasura API is deployed in a containerized environment, automatic scaling is done as easily as spawning new instances of the container based on key performance indicators such as requests per second. In summer 2019 the Hasura team verified these statements by testing their system with up to one million rows updated per second on a PostgreSQL database (GraphQL Foundation, n.d.). At this number of requests the PostgreSQL load was measured at 28% load with peak number of PostgreSQL connections at around 850. These values are way above any traffic that the ODS has been seen until today and most likely will not see for the foreseeable future and is therefore more than

sufficient for the ODS (Hasura, 2019).

Onboard explorer

Hasura ships with the GraphiQL[11] online explorer tool which is used to explore the different parts of the API. The explorer lets one examine all queries and types and supports building queries by providing documentation, auto completion and an interface to build a query out of existing components. This UI could be forwarded from the JValue user interface to enable easy exploration of the available data.

Figure 7 - Interface of GraphiQL. Von Rönne (2020)



4.1.3 Alternatives to Hasura GraphQL engine

Two alternatives to Hasura were considered but in the end were not pursued for various reasons. One possible solution alternative is the PostGraphile[12] library, developed within the toolset of Graphile. This library follows a similar approach to the Hasura GraphQL engine. The library takes existing postgres tables and uses them to automatically create a working GraphQL API. The current set of functions is also very similar, extensions are being handled via a plugin system (GitHub, n.d.). In the end the main arguments for not using Postgraphile are the more complex initial setup and less active community. Especially the activity of the community with new commits and active contributors fueled the decision to implement the feature using the Hasura GraphQL engine.

The second approach consists of building the component that handles the API creation internally. Benefits are of course the ownership of the code base and therefore the ability to create and use code that has been directly designed for the JValue ODS. An early concept consisted of a number single docker containers that hosted a self-written, automated GraphQL

[11] https://github.com/graphql/graphiql
[12] https://www.graphile.org/postgraphile/

API that would have been able to be expanded by community supplied docker images. However, the amount of development work needed and missing knowledge in the developing team along with security concerns quickly eliminated this approach.

4.1.4 API Orchestration Services

The API Orchestration Service is the main piece of code written for this thesis. As seen in Illustration 5 the service is the mediator between the Hasura API and the other micro-services. Its main responsibility consists of two tasks. Firstly, setting up the default configuration of Hasura, so that all existing pipelines get served with a default API that allows all users to gain reading access to the data stored by the JValue ODS.

The other responsibility is handling the configuration of the GraphQL engine during runtime. This means the service must allow for modifications of the GraphQL API which could consist of adding or deleting pipelines or extending the existing schema with additional remote endpoints.

As the internal communication of the ODS is being rewritten as of writing this thesis, the orchestration service exposes this functionality through a Rest interface, and in addition is being able to handle incoming messages through the RabbitMQ[13]. The GraphQL-engine primarily provides a GraphQL configuration API which will be used to handle the setup and necessary adjustments.

There are two primary use-cases for users of the ODS in which they want to interact with the GraphQL API. Firstly, they simply want to consume the API and use the provided data for their own use and projects. Secondly, they might want to inject additional business logic to the default API.

The proposed solution fulfills this request by providing the user the possibility to do this by utilizing the Hasura remote schema feature. The user must build their own GraphQL server, host it on their own hardware and then registers the service via the JValue ODS UI.

4.1.5 Template remote schema project

To use the remote schema feature of Hasura, users are required to have a basic understanding on how GraphQL works to be able to implement their own server. To simplify this process for the users of the JValue ODS, the template remote schema project is designed as a template that already includes most of the boilerplate code required for setting up a basic GraphQL Web service.

[13] https://www.rabbitmq.com/

The project is being kept intentionally lean, just the very basic required functions are implemented. These consist of consuming the existing GraphQL API, altering the data and exposing a new endpoint that can be used to register the service via the UI. All settings that can be generalized are being preconfigured, and an easy to understand short documentation is being provided to guide users along the way.

5 Implementation

5.1 Integration of GraphQL engine

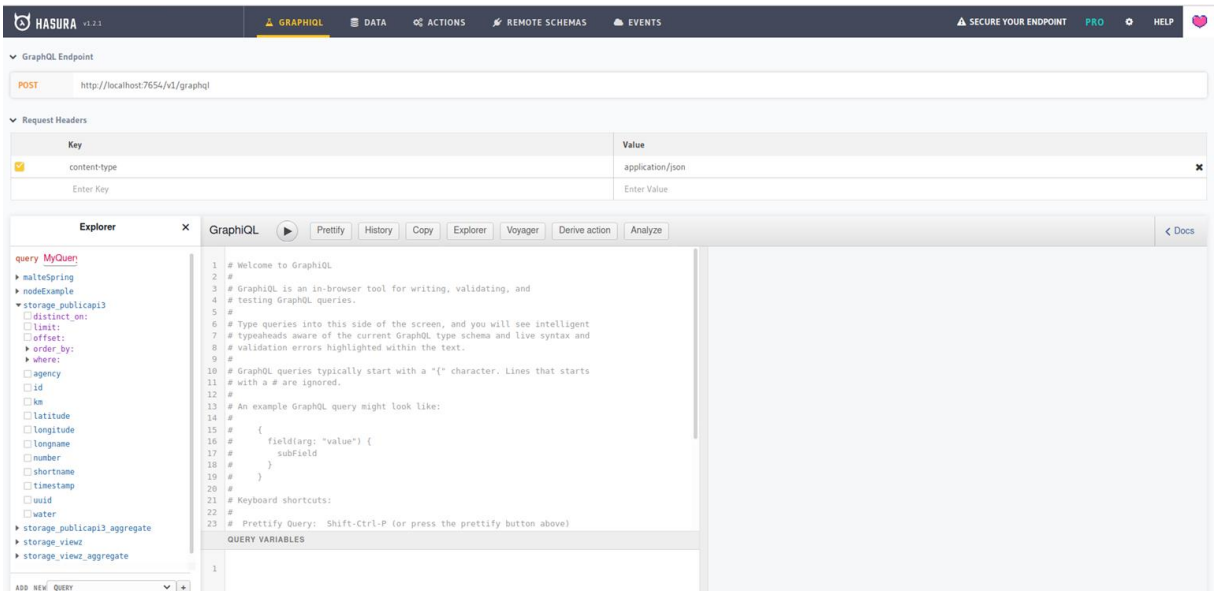
Hasura offers three different ways on how their service can be integrated into an existing project landscape: using their own Hasura hosted cloud, using a deployment on Heroku or using native docker. As the JValue ODS is setup of multiple dockerized microservices, it makes sense to continue this pattern by integrating the GraphQL-engine in the docker-compose file.

```
graphql-engine:
  image: hasura/graphql-engine:v1.2.1
  ports:
    - "7654:8080"
  depends_on:
    - "storage"
  restart: always
  environment:
    HASURA_GRAPHQL_DATABASE_URL: postgres://ods_admin:ods_pw@storage-db:5432/ods
    HASURA_GRAPHQL_ENABLE_CONSOLE: "true"
```

To set up a default Hasura instance, the chosen image and the configuration of the environment in which Hasura is supposed to run must be provided. The setup mainly consists of the network configuration and providing the settings to the PostgresSQL storage database.

Additionally, there are some other parameters to customize the instance, e.g. enabling the web console or defining secret for security.

Figure 8 - Interface of the Hasura web console. Von Rönne (2020)



After this step and running the application it is already possible to access the Hasura console. Hasura takes by default every table it has access to, through the provided credentials and creates a default GraphQL API for it. This API can be explored through the GraphiQL tool in the interface.

Using the default example of creating a pipeline that connects to the ‘Pegelstände’ Online API of the WSV, it can be seen how Hasura converts the existing PostgreSQL database schema into an API. Each individual field gets taken and transformed into a select-able and parameterize-able identifier which can be used to build queries to the API.

This proves a challenge for the usage of Hasura with the existing JValue ODS as the schema currently condenses all the data provided by one execution of the pipeline as a JSON string into the data field. Hasura is therefore only able to allow filtering on the metadata fields of the pipeline, namely the ID, license, origin, pipeline-ID and the timestamp.

However this is mostly data that the user is not particularly interested in, the user wants to be able to filter through the data that was returned by the ‘Pegelstände’ API and filter for different properties in there, e.g. which station currently reports a water level above a certain threshold.

This issue and further configurations therefore must be handled during runtime by the API Orchestration Service.

5.2 API Orchestration Service

5.2.1 Container

```
#-------------------------------------------------------#
# First stage: image to build and test Java application #
#-------------------------------------------------------#
FROM adoptopenjdk/openjdk13-openj9:alpine-slim as builder

WORKDIR /builder
COPY *.gradle /builder/
COPY src /builder/src
COPY gradle /builder/gradle
COPY gradlew /builder/gradlew

# Build project and run unit tests
RUN ./gradlew assemble
RUN ./gradlew test

#-------------------------------------------------------#
# Second stage: image to run Java application           #
#-------------------------------------------------------#
FROM adoptopenjdk/openjdk13-openj9:alpine-slim

RUN mkdir /app
WORKDIR /app

ENV SPRING_PROFILES_ACTIVE prod

# Pull the dist files from the builder container
COPY --from=builder /builder/build/libs/* app.jar

# Run app and expose port to other containers
EXPOSE 6969
ENTRYPOINT ["java", "-jar", "app.jar"]
```

The container is mostly setup the same way as the other services of the JValue ODS that run in a java environment. The common approach is it to divide the Dockerfile into two stages. The first stage is concerned with building the application whereas the second stage starts the final application. For both stages the alpine-slim image has been used as base image with the JDK version 13. Firstly, all the needed files are copied into the /builder directory. Afterwards

Gradle[14] is started with the assemble command to build the application.

In the second stage an /app directory is created in which all the artifact *app.jar* and the libraries from the builder are put into. Lastly the app is started with the command *java -jar app.jar* and an exposed port.

To integrate this service into the micro service structure of the ODS now, it must be configured through the docker-compose.yml file. The code snippet for that looks as follows:

```
api-configuration:
  image: ${DOCKER_REGISTRY}/api-orchestration
  ports:
    - 8090:8090
  build:
    context: ./api-configuration/
  depends_on:
    - graphql-engine
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.to-apiorchestration.rule=PathPrefix(`/api/apiorchestration`)"
    - "traefik.http.routers.to-apiorchestration.middlewares=apiorchestration-stripprefix@docker"
    - "traefik.http.middlewares.apiorchestration-stripprefix.stripprefix.prefixes=/api/apiorchestration"
    - "traefik.http.services.apiorchestration.loadbalancer.server.port=8090"
```

In short, this code fragment instructs docker-compose to build the previously described image, export port 8090 and configure various Traefik rules that describe how this service is positioned in the network. Additionally, all needed environment variables are being passed to the service through here.

5.2.2 PostgresSQL schema issue

There are multiple ways on how the problem of the insufficient database schema of the JValue ODS can be addressed. Firstly, it would be possible to alter the whole schema application wide. However, this has serious implications on all other parts of the application that in some way interact or depend on the current schema. This change would need lots of analyzing and communication with the current project team to settle on a suitable solution.

Another more pragmatic solution, and the way it was finally implemented, was is to create a view on top of the existing table that unwraps the jsonb data field of the individual entries and create individual columns and rows. PostgreSQL provides additional functions to plain SQL

[14] https://gradle.org/

that can handle data stored in jsonb datatype fields. These functions allow us to loop through the data field and unwrap it into different columns for the keys and individual rows for the stored objects.

The java class *SQLFactory* of the API Orchestration Service handles most of this workload. It exposes a method called *createSchemaQueryWithUnpackedFields* which takes the name of table and the name of the to be created view and returns a SQL Command as a String. Internally it queries the first entry of the table and extracts all of the keys stored in the data field on the top level. These keys are then used to build the resulting SQL Command which consists of three different parts.

Firstly, the header is generated.

```
private static final String SQL_SCHEMA_CREATION_HEADER =
  "CREATE OR REPLACE VIEW \"storage\".%s AS " +
    "SELECT id, \"timestamp\" ";
```

This part is responsible for creating the view with the provided *viewName* as parameter.

The second part is responsible for creating a field based on the previously extracted keys. The final java script loops through the keys and adds this SQL snippet for each individual key.

```
private static final String SQL_SCHEMA_CREATION_FIELD = "x.datapoint ->> '%s'::text AS %s ";
```

The last part describes from where the data is to be picked to make it available to the previous script.

```
private static final String SQL_SCHEMA_CREATION_REST =
  "FROM (WITH a AS " +
    "(SELECT \"%s\".id, \"%s\".timestamp, jsonb_array_elements(\"%s\".data) AS datapoint " +
    "FROM storage.\"%s\")" +
    "SELECT a.id, a.timestamp,a.datapoint FROM a) x;";
```

After these steps, a query is being returned that can be run against the PostgreSQL Instance to create a new view for the specified table.

This is the resulting view when run against the table containing the ‘Pegelstände’.

Figure 9 - Example entry of unraveled storage database entry after transformation. Von Rönne (2020)

	id	timestamp	km	uuid	water	agency	number	latitude	longname	longitude	shortname
1	1	2020-05-11 13:40:00	9.56	47174d8f-1b8e-4	{"longname"	WSA VERDEN	48900237	52.90406541008721	EITZE	9.27676943537587	EITZE

Note that individual fields for each key of the data have been created. A column now does not represent the data returned by the execution of a pipeline but instead a specific station with the associated properties. This also implies that the primary key is no longer just the *ID*, but the combination of the *ID* and *UUID* of the data.

5.2.3 The REST API

The REST API is implemented with the framework of spring boot. To keep it consist with the other rest Service of the JValue ODS, the same libraries with the same version have been used whenever possible.

The API is integrated the same way as the other services in the application by referencing it in the *docker-compose.yml* file.

Following minimal set of endpoints have been identified and implemented.

```
@PostMapping("/deleteAPIForTable")
HttpStatus deleteAPIForTable(@RequestParam(name = "tableName", required = true) String tableName);

@PostMapping("/createDefaultApiForTable")
HttpStatus createDefaultApiForTable(@RequestParam(name = "tableName", required = true) String
tableName);

@PostMapping("/removeDefaultAPI")
HttpStatus removeDefaultAPI(@RequestParam(name = "tableName", required = true) String tableName);

@PostMapping("/initDefaultGraphAPI")
HttpStatus initDefaultGraphAPI(@RequestParam(name = "tableName", required = true) String[] tableNames);

@PostMapping("/addRemoteSchema")
HttpStatus addRemoteEndpoint(@RequestParam(name = "schemaName", required = true) String schemaName,
@RequestParam(name = "url", required = true) String url);

@PostMapping("/removeRemoteSchema")
HttpStatus removeRemoteSchema(@RequestParam(name = "schemaName", required = true) String
schemaName);
```

These endpoints provide the following functionality: creating and deleting default APIs for a given pipeline, initializing the default API on first start-up and registering a remote schema by a given URL.

5.2.4 Integration of RabbitMQ

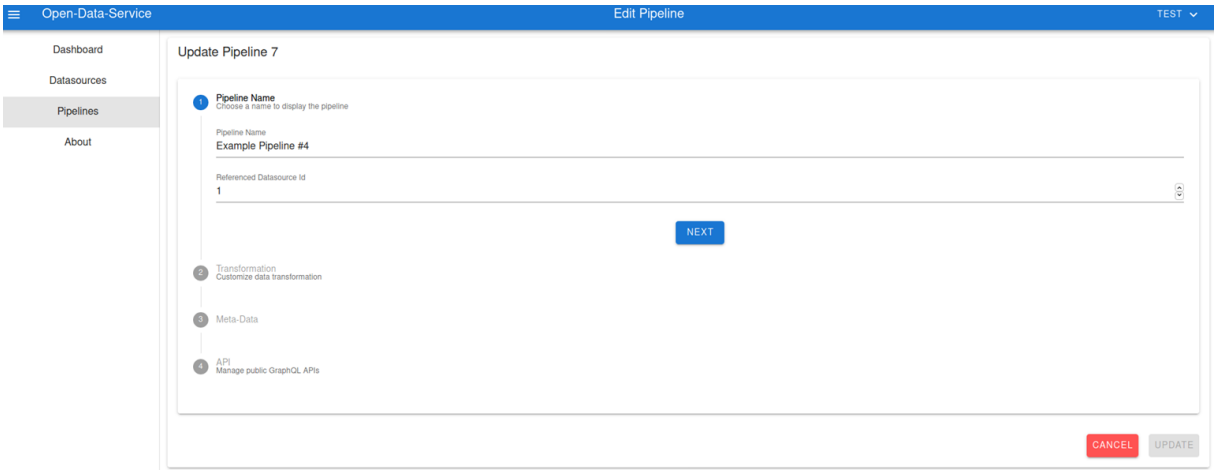
For the integration of RabbitMQ the provided libraries of Spring spring-amqp and spring-rabbit have been used. The RabbitConfiguration class takes the environment variables provided in the docker-compose.yml file to enable to *RabbitMQConsumer* class to listen to the proper events exposed by the storage-mq service. These events include the necessary API configuration in its payload which is then parsed into the *PipelineData* model class. After analyzing the received data, the proper methods of the *ApiConfigurationService* are being executed (Müller 2020).

5.3 Necessary changes to other components

5.3.1 User interface

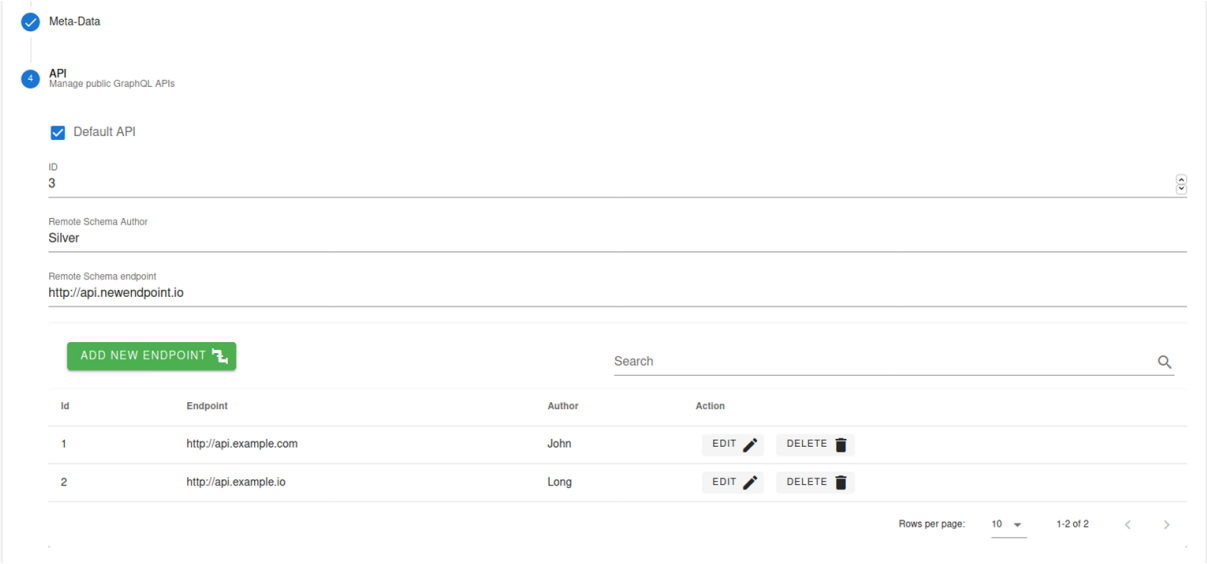
The user interface must be extended to give the users the control over the API associated to their pipelines. The main features that need to be configured from the UI are whether a default API is generated or not and furthermore the different remote Schemas that are being attached to it.

Figure 10 - JValue ODS UI: Updated pipeline creation/edit form. Von Rönne (2020)



The configuration for this part of the pipeline is done in a separate step in the pipeline creation/edit workflow.

Figure 11 - JValue ODS UI: Configure API of pipeline. Von Rönne (2020)



Firstly, the user can choose whether he wants to have the GraphQL API created or not by selecting or deselecting the checkbox ‘Default API’. Next, the user is confronted with an input Form in which he can attach remote Endpoints with the endpoint URL and a name of the Author responsible for this endpoint. The button ‘Add new Endpoint’ adds the endpoint to the configuration of the pipeline and displays it in the table below. Illustration 11 shows how the user can delete an endpoint or change the configuration by editing it.

5.3.2 Database schema

During the development of this feature there has been a focus on keeping all changes as backwards compatible as possible. However, for the persistence of the API configuration it was necessary to alter the PostgreSQL schema for the PipelineConfiguration table. In line with the changes to the User Interface two new fields have been added.

Figure 12 - New Pipelineconfigs table schema. Von Rönne (2020)

Table: PipelineConfigs

Select data **Show structure** Alter table New item

Column	Type	Comment
id	bigint	
datasourceId	bigint	
func	character varying	
author	character varying *NULL*	
displayName	character varying	
license	character varying *NULL*	
description	character varying *NULL*	
createdAt	timestamp *NULL*	
defaultAPI	boolean *NULL*	
remoteSchemata	jsonb *NULL*	

The *defaultAPI* field of data type boolean saves whether this pipeline should publish a default GraphQL pipeline or not.

The *remoteSchemata* jsonb field saves the configuration of the various remote Schemata in a JSON array of objects. The JSON object consists of the *fields' 'id', 'endpoint'* and *'author'* which persist the configuration done by the user in the UI.

5.4 Template project

The template project has one defined goal that it aims to accomplish, namely give users that are unfamiliar with GraphQL a hands-on example on how they can implement and run their own server to enrich the GraphQL API with their own customary logic. To accomplish this task, a tech stack has been chosen that caters to a big audience and is considered beginner friendly.

5.4.1 Programming language

The server is setup with node.js in the version 8.10. Node is a JavaScript runtime that is built on Chromes V8 JavaScript engine. It is widely used and has and offers the GraphQL libraries of Apollo[15] with excellent integration with GraphQL.

[15] https://www.apollographql.com/

5.4.2 Functionality

To keep the project as simple as possible and only focus on its main purpose, its sole functionality is in reading data from an existing GraphQL API, enriching this data with an additional field and exposing this endpoint via an URL which then can be taken to bind this service to the ODS.

5.4.3 Querying data

As the Apollo library is used, the main point of interest is the Apollo client. After initializing this client with the library pointed to the GraphQL endpoint of the ODS, querying data can be started:

```
const data = client.query({
  query: gql`
    query MyQuery {
      storage_viewz(limit: 1) {
        id
        agency
      }
    }
  `,
})
```

In this example the user queries for the object ‘storage_viewz’ and ask for the first object and require that the response should include the id and agency of the object.

5.4.4 Exposing endpoint

Following the introduction introduced in chapter 3.2, the required types of the new API are defined:

```
const typeDefs = gql`

 # Example enriched data type
 type ImprovedData {
  id: String
  agency: String
  newCustomField: String
 }

 # Example Query type
 type Query {
  betterData: ImprovedData
 }
`;
```

The type improved data type gets an additional field *newCustomField* of type string. This type is then added to a query called *betterData* which returns the type *ImprovedData*.

Next it must be defined what happens internally when *betterData* is being queried for by defining the resolvers:

```
// mapping of how the example type betterData is being resolved
const resolvers = {
 Query: {
  betterData: () => collectBetterData(),
 },
};
```

In this example the request is being passed on to the function *collectBetterData()*. This function queries the existing JValue GraphQL API and returns the same data with the additional field *newCustomField* with value ‘custom’.

```
// example function that firstly collects data from the server and then
// enriches the data with custom business logic
function collectBetterData() {
  const data = client.query({
    query: gql`
      query MyQuery {
        storage_viewz(limit: 1) {
          id
          agency
        }
      }
    `,
  })
  .then(result => {
    console.log(result.data.storage_viewz[0].id)
    return {
      id: result.data.storage_viewz[0].id,
      agency: result.data.storage_viewz[0].agency,
      newCustomField: "custom1"
    }
  });
  return data;
}
```

6 Evaluation

In the beginning of the thesis several different functional and non-functional requirements have been established that can be used to evaluate the final result against. Analyzing each requirement on whether it has been successfully implemented and if not what the reasons are behind the failure.

6.1 Functional requirements

6.1.1 Automatic generation of basic API

This requirement has been fulfilled. A user is able to decide by selecting a checkbox on creation or editing of the pipeline whether a generic API should be automatically generated or not.

6.1.2 Navigating between snapshots of the data set

This requirement has been mostly fulfilled. The final generated API allows filtering for different data snapshots with their specific timestamps. However, when formulating the requirement, the optimal solution would have included separate field in the data to their successor and predecessor. In hindsight this was owed to the fact of imagining some sort of REST interface in which an implementation as of now would have not been possible.

6.1.3 Interface to other components

This requirement has been fulfilled, there is no intermediate layer between service and the storage database. The service exposes a REST interface along with the integration of RabbitMQ so developers of other components can choose how they want to interact with the API-configuration service.

6.1.4 Query language

This requirement has been fulfilled. GraphQL is the main query language that is being used to acquire the data gathered by the pipeline.

6.1.5 Extensibility through the JValue ODS community

This requirement has been mostly fulfilled. A template project that can extent the schema of the GraphQL database is available for the users of the ODS. However, to use this solution the user needs to be technology versed and familiar with node.js.

The current solution only works for offsite hosted GraphQL servers, the users must host their extension themselves. This limits users that are not able to host their own infrastructure, be it for knowledge-based reasons or economic ones.

6.1.6 Reasonable time to set up API

This requirement has been fulfilled. Using some benchmark testing the median time measured

between creating the pipeline and the ability to query it takes about 4.5 seconds. Note that this also includes all the other steps the system has to take when a new pipeline creation has been triggered like querying the first data set, applying transformations and storing the data and configuration in the databases.

6.1.7 User interface

This requirement has been fulfilled. The user has the possibility to choose whether the API is available or not through a checkbox in the edit pipeline workflow. Furthermore, the user is able to add or remove certain entries of the remote API configuration table which maps the external schema extensions to the API.

6.2 Non-functional requirements

6.2.1 Adaptability

This requirement has been mostly fulfilled. All code that is being deployed with the JValue ODS it written in Java utilizing the Spring framework which is heavily relied on by the JValue ODS. The GraphQL-engine is open sourced and will be continually improved on by the flourishing community or even the team of the JValue ODS themselves if the need arises in the future.

6.2.2 Licensing

This requirement has been fulfilled. All licenses of the suggested solution are compatible with the open source standards set by the JValue ODS.

6.2.3 Deployment

This requirement has been fulfilled. The API-orchestration service is deployed with docker and included in the docker-compose files of the main application. The Hasura GraphQL-engine is also deployed in a docker container and included in the docker-compose files.

7 Further steps and conclusion

The pull request associated with this thesis contain all the previously described and discussed features and is ready to be integrated into the master branch of the JValue ODS project. However, as it is the case with most new features, there is still a lot of work that could be done to further improve the quality of the GraphQL components. One major aspect that has been mostly left out of this thesis but is of high importance and should be analyzed in the coming future, is the security concept resolving around the APIs. As of now all the APIs are created in a public domain and can therefore be accessed by any individual who has access to the network that is hosting the JValue ODS. Hasura GraphQL-engine has a huge array of features that enable the provider of the service to secure their endpoints, be it via integration of Keycloak[16] or even using the existing role concept of the database.

Another area that requires additional work in the feature is the way in which the data stored by the adapter service is turned into a PostgreSQL schema that is used as the basis of the GraphQL API. However, this will require a complete look at the way the ODS works in transforming data from external endpoints in a suitable format that can be used by the JValue ODS and its components.

In retrospect the main difficulties with this thesis have been the large amount of knowledge that is required to understand all the different aspect of the JValue ODS and the final integration. Such an integral new feature as an automatically created and deployed API required changes in many different areas of the project and therefore the need to familiarize oneself with a wide array of different technologies such as deployment with Docker and docker-compose, Spring, vueJS, Traefik, PostgreSQL, GraphQL, Apollo, nodeJS, GraphQL-engine and more. This prior learning process required a large amount of the time used for writing this thesis that in the end is not being reflected by the final result. Nevertheless, have most of the defined acceptance criteria been fulfilled and a working system is ready to be integrated into the JValue ODS which can be improved on in future changes.

[16] https://www.keycloak.org/

8 References

Apache Software Foundation. (January 2020). *apache.org*. Retrieved on 31. July 2020 from https://www.apache.org/licenses/LICENSE-2.0

BEZUGLA, K. (19. April 2019). *dzone.com.* Retrieved on 7. July 2020 from https://dzone.com/articles/graphql-core-features-architecture-pros-and-cons

BLANCHETTE, J. (2008). *The Little Manual of API Design.* Espoo: Trolltech. Retrieved on 6. September 2020 from https://people.mpi-inf.mpg.de/~jblanche/api-design.pdf

BLOCH, J. (n.d.). How to Design a Good API and Why it Matters. Retrieved on 23. August 2020 from https://pdfs.semanticscholar.org/8d66/ded1587ffc89b860d2217ff6543aa2da0592.pdf

Data Policy and Innovation Unit G. 1. (22. March 2020). *europa.eu*. Retrieved on 31. July 2020 from https://ec.europa.eu/digital-single-market/en/open-data

GitHub. (n.d.). *github.com*. Retrieved on 15. August 2020 from https://github.com/graphile/postgraphile

GraphQL Foundation. (n.d.). *graphql.org.* Retrieved on 13. September 2020 from https://graphql.org/

GraphQL Foundation. (n.d.). *graphql.org*. Retrieved on 27. September 2020 from https://graphql.org/learn/

Hasura. (17. July 2019). *hasura.io*. Retrieved on 9. September 2020 from https://hasura.io/blog/1-million-active-graphql-subscriptions/

Hasura. (n.d.). *hasura.io.* Retrieved on 27. July 2020 from https://hasura.io/user-stories/

Hasura. (n.d.). *hasura.io*. Retrieved on 9. September 2020 from https://hasura.io/docs/1.0/graphql/core/

Japan Meteorological Agency. (n.d.). *data.jma.go.jp*. Retrieved on 24. July 2020 from https://www.data.jma.go.jp/svd/eqev/data/bulletin/catalog/notes_e.html

LARDINOIS, F. (8. September 2020). *techcrunch.com*. Retrieved on 22. September 2020 from https://techcrunch.com/2020/09/08/hasura-raises-25-million-series-b-and-adds-mysql-support-to-its-graphql-service/?guccounter=1

MALAN, R., & BREDEMEYER, D. (7. July 1999). *psu.edu.* Retrieved on 14. August 2020 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.436.4773&rep=rep1&type=pdf

MIT. (n.d.). *opensource.org.* Retrieved on 29. August 2020 from https://opensource.org/licenses/MIT

PICCIONI, M., FURIA, C. A., & MEYER, B. (n.d.). An Empirical Study of API Usability. Zürich, Switzerland. Retrieved on 14. August 2020 from http://se.ethz.ch/~meyer/publications/empirical/API_usability.pdf

RIEHLE, D. (September 2019). Enabling Open Innovation through Open Data with the JValue Open Data Service. Penang, Malaysia. Retrieved on 16. July 2020 from https://dirkriehle.com/wp-content/uploads/2019/09/Enabling-Open-Innovation.pdf

RUSSELL, D. (21. November 2019). *rubrik.com.* Retrieved on 6. August 2020 from https://www.rubrik.com/blog/graphql-vs-rest-apis/

SCHWARZ, G., ZINNEN, M., User 9dt, BAUER, A., MARTINEZ, H., & CASADO QUIJADA, A. (3. August 2020). *github.com*. Retrieved on 4. September 2020 from https://github.com/jvalue/open-data-service/blob/master/README.md

traefiklabs. (n.d.). *traefic.io.* Retrieved on 29. September 2020 from https://doc.traefik.io/traefik/

9 List of figures

10 References of figures

Figure 1

SCHWARZ, G., ZINNEN, M., User 9dt, BAUER, A., MARTINEZ, H., & CASADO QUIJADA, A. (3. August 2020). *github.com*. Retrieved on 4. September 2020 from https://github.com/jvalue/open-data-service/blob/master/README.md

Figure 5

SCHWARZ, G., ZINNEN, M., User 9dt, BAUER, A., MARTINEZ, H., & CASADO QUIJADA, A. (3. August 2020). *github.com*. Retrieved on 4. September 2020 from https://github.com/jvalue/open-data-service/blob/master/README.md

Figures 2 - 4; 6 - 12

Von Rönne, K. M. (September 2020). Self-created illustrations. Nürnberg, Germany.