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

JENS WÄCHTLER
BACHELOR THESIS

DESIGN AND IMPLEMENTATION OF PARAMETERIZABLE DATA IMPORT FOR THE JVALUE ODS

Submitted on 29 October 2020

Supervisors:
Prof. Dr. Dirk Riehle, M.B.A.,
Georg Schwarz, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität 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.

Erlangen, 29 October 2020

License

Erlangen, 29 October 2020

Abstract

Governments have recognized that the publication of open data is of great economic and social value. Collecting and using this data is challenging because it is not always available in an easy to process format. Minimizing these challenges is the task of the JValue Open Data Service (ODS), a system that makes data consumption easy. Yet the location of a resource and the time of a data import is statically defined.

This thesis presents a concept how the ODS can be extended by *parameterizable datasources* and how the data import can be triggered manually. This addresses the challenge of rapidly changing data on the Internet and adapts the ODS in order to deal with the emerging problems. With *parameterizable datasources* it is viable to dynamically describe the location of resources. The possibility for manual data imports ensures that data is only retrieved when it is really needed. The design decisions and the implementation of these functionalities for the ODS are covered in this thesis.

Contents

Acronyms

API Application Programming Interface

ODS JValue Open Data Service

REST Representational State Transfer

UI User Interface

URI Uniform Resource Identifier

HTTP Hypertext Transfer Protocol

FTP File Transfer Protocol

ID Identifier

ETL extract, transform, load

AMQP Advanced Message Queuing Protocol

JSON JavaScript Object Notation

URL Uniform Resource Locator

XML Extensible Markup Language

1 Introduction

Data has become increasingly important as an economic factor. It is also referred to as the oil of the information era [Eco17]. Tech companies like Facebook or Google have specialized in the collection of data, especially user data. These are rarely made available to other parties and if they are, it is usually for a fee.

This is in contrast to the concept of open data. Open data is the idea that some data should be freely available to everyone to use, and share. The value of open data in the European Union is estimated at 184 billion in the year 2019 [HK20]. In addition to the market economy view, open data is a vast collection of knowledge for society.

The major problem with open data is that the raw information of different sources is in heterogeneous form. In order to address this, the ODS was developed. It was created in order to provide public data in a uniform, processed form and to better utilize the potential of open data.

The ODS collects this data according to a predefined schedule, processes it, stores it and makes it available to a user. This is called a *pipeline. Datasources* serve as the basis for a *pipeline.* These describe the location of the data and their meta information.

Users of the ODS in its current state face two problems: First of all, when creating a *datasource,* a user specifies the timeliness and quantity of the data using the preset time slots. It can happen that the external data does not change between the periods and is unnecessarily collected, or newer data has already been made available than what was fetched. If the time periods are chosen too large, the timeliness decreases, if they are too small, the amount of data increases. A user of the ODS thus determines the up-to-dateness of the data already when creating a *datasource,* without having any influence on the original external source. The ODS lacks the capability to guarantee that the latest data is received.
Second, some open data is made available via an Application Programming Interface (API), which requires parameters to be passed in order to be able to use them. These can be entities that are located around a fixed radius of a geographical coordinate. When defining a *datasource* for the ODS, this is done via a fixed

Uniform Resource Identifier (URI). This means that the parameters are defined by the user when creating the *datasource*. If entities within a different coordinate are to be requested from the above example, a user must describe a new data source, although they only differ in the coordinate.

This thesis introduces a concept and implementation to provide these features for the ODS. It shows how *datasources* can be requested and processed with the highest possible currency. Additionally, a concept is implemented how to make parameter-based data sources accessible.

2 Requirements: Support parameterizable data imports

In the course Advanced Methods of Software Engineering (AMSE) of the Open Source Research Group at the Friedrich-Alexander Universität Erlangen-Nürnberg students were testing and evaluating the ODS with self-chosen group projects. In their projects, they were able to make use of the capabilities that the ODS offers them. However, they also reached the limits of the current feature set. The most important remark was that they would like to have a way to send query parameters with their request to the ODS. They also wanted API requests to be generated not only periodically but also by an individual request to the ODS.

Those valuable insights from the actual use of the ODS, as well as brainstorming with the active developers were used to generate the upcoming hierarchical list of requirements. The respective major features have been divided into subrequirements. To realize them completely, all subitems must be implemented.

2.1 Configuration of parameterizable pipelines

Conceptually, a *pipeline* in the ODS consists of a *datasource,* a *transformation,* data storage and zero or more notifications.

In a *datasource* URIs are used to specify the location from where data should be fetched. The endpoint of a Representational State Transfer (REST) interface usually consists of fixed and variable components. In addition, a URI can contain variable query parameters. At the moment it is only possible to define a *datasource* with predefined parameters as a *pipeline* within the ODS. The user should be able to define the variable components as parameters.

2.1.1 API design to define parameters

The system shall provide an endpoint to create *parameterizable datasources*. For this purpose, a suitable model shall be provided.

Allow naming of parameters

It shall be possible to pass names for the parameters. These shall serve for identification and assignment of potential parameters.

Allow default values of parameters

When creating a *parameterizable datasource* it shall be allowed to set default values for the parameters. These are to be used for the localization of a resource unless otherwise specified during execution.

2.1.2 UI component for seamless configuration

Within the user interface it should be possible to create and configure *parameterizable datasources*. This should be kept as intuitive as possible for the user.

2.2 Execution of parameterizable data sources

Currently, the ODS uses an internal scheduler to trigger a *pipeline* periodically. The user has the option to define the period between two executions for each individual *pipeline*. However, it is not possible for a user to manually trigger an execution of the *pipeline* outside the defined times. This would be necessary if the most current data of a data source is needed. To achieve this goal, a function shall be provided that enables the manual triggering of a pipeline. This functionality should be available via the User Interface (UI) as well as by a request to the ODS. The following tasks must be implemented for this purpose.

2.2.1 Trigger mechanism

To initiate a data import there shall be a mechanism to notify the system. For this purpose, interfaces must be defined.

API

The system shall offer an API to trigger a manual data import. For this purpose, a suitable interface has to be modeled.

UI

Within the UI a user should be able to initiate a data import for a created *datasource.*

Allow defining parameters

The user should be able to fill open parameters at the time of a manual data import request. The system shall provide an interface for purpose.

2.2.2 Should execute data source

The system shall fill the open parameters with those set by the user in a data import request. Then a data import is to be initiated. In the case of a valid request, the user receives the requested data.

Take default parameters if not defined

When a data import request is made to the system, it should be executed. If there are open parameters for the requested *datasource* and no parameters are passed in the request, the default parameters should be used.

2.2.3 Should execute pipelines

As it is possible to trigger a data import manually, it should be ensured that transformations are applicable to it. For this, the system must be informed about a manual data import. Once the system receives a request for a pipeline for a *parameterizable datasource* by a user it should execute it. The request should be checked for validity. For valid requests the data source should be queried. If a transformation has been defined for this *pipeline,* it is applied to the received data.

2.2.4 Should return the resulting data to requesting actor

After the manual request for a *pipeline* has been made, the *pipeline* is executed. The system then sends the data back to the requester as a response.

2.3 Still support non-parameterizable data sources

The modifications to the system should not change the previous way of using the ODS, but only expand it. In particular, the function of a *parameterizable datasource* is not intended to replace the existing periodic pipeline queries from data sources.

3 Fundamentals

This chapter provides an overview of the basics for this thesis. A brief overview of the technologies that were required to expand the ODS to support parameterizable data imports is given. The first part of the chapter deals with how to access resources in the network. Then, the technologies on which the ODS is based are discussed, in order to go into the ODS in more detail at the end.

3.1 Uniform Resource Identifier

As a basis for the ODS it is important to be able to request resources. The ODS makes use of public data that is accessible via the Internet. In order to make this data usable through the ODS, it is necessary to localize the data. For this purpose, a description language is used to identify a resource. URIs are used to identify these resources. They belong to the Internet standard and are documented in [BFM05]. The following specification defines URIs.

> "A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource. This specification defines the generic URI syntax and a process for resolving URI references that might be in relative form, along with guidelines and security considerations for the use of URIs on the Internet. The URI syntax defines a grammar that is a superset of all valid URIs, allowing an implementation to parse the common components of a URI reference without knowing the scheme-specific requirements of every possible identifier. This specification does not define a generative grammar for URIs; that task is performed by the individual specifications of each URI scheme." [BFM05]

A URI is first of all an abstract construct, but in the form of a Uniform Resource Locator (URL) it forms one of the foundations for the World Wide Web. A URL is a specialized form of the URI. First, we take a closer look at the structure of a URI.

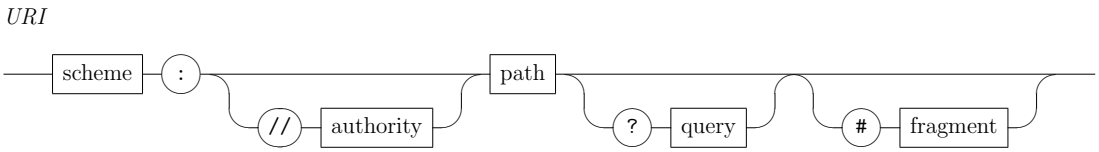


Figure 3.1: URI syntax diagram [BFM05]

Syntax and Grammar

A URI consists of a limited set of characters from US-ASCII. By default the letters of the Latin Alphabet, numbers and the special characters - . _ and ~ are allowed. There are also characters with certain tasks. Some of them are used to delimit the URI, others to encode characters that are not allowed. There is a pool of possible delimiters, which you can allow, but do not have to. These are ":", "/", "?", "#", "[", "]", "@", "!", "$", "&", "'", "(", ")", "*", "+", ",", ";" and "=". For encoding, use "%" followed by two hexadecimal numbers. This results in characters that are still available and must not appear in a URI. These include "|" "{" "}" "<" ">", "^", and "\".

A URI consists of five parts: Scheme, authority, path, query and fragment. The structure is shown in figure 3.1. For a valid URI only scheme and path are required.

The scheme defines a context. This can be a protocol like Hypertext Transfer Protocol (HTTP) or an International Standard Book Number (ISBN) to identify books. To differentiate to the next part of the URI a colon follows after the scheme. A scheme always starts with a letter. Examples include http, https, ftp or mailto [BFM05].

The hierarchical path follows the scheme, which is composed of the optional authority and the path.

3.2 Microservices

Originally, the architecture of the ODS was a monolith, a coherent unit. Since then it has been transformed into a microservice architecture. For a better understanding of how the ODS works, it is important to know what microservices accomplish.

3.2.1 Characteristics of Microservices

The idea of microservices is to divide a large application into many small functionalities, each one as a separate standalone service. They interact with each other using a communication protocol over the network through fixed interfaces

[New15]. This leads to a reduction in coupling. Microservices can be deployed independently. Changes to one microservice can be released into production independently of changes to other microservices [Wol16].

Typically, when a monolithic application is running at full capacity, the solution is to start an additional instance, even though only individual functions within it represent the bottleneck. The microservice architecture allows to run an application not only on one single machine but on many different ones in their own processes. As a result, microservices are not restricted to a technology like the programming language. It is possible that each service is implemented in a different programming language. The fact that you can run several instances of a service in parallel increases the scalability. If a service has an increased computing demand, new instances can be started and if the load drops, they can be shut down again. It is easy to replace the implementation of individual services without having to go deep into the architecture, because of well-defined interfaces. This makes it easier to add, extend or change functionality without the need to understand the complete implementation of an application architecture. Furthermore, it is possible to provide new functionalities as a part of the application without replacing and providing the complete system. Therefore, the maintainability is increased. In addition, individual microservices can also be reused for other applications, as they are ideally only responsible for single functionalities. Individual services should be manageable, preferably handled by a single team. This takes up the idea of agile software development, where the goal is to make the development process more flexible and leaner [Wol16].

In addition to the many advantages of microservices, there are also challenges in development. The introduction of microservices creates more complexity within the system. It is important to note that network connections are not reliable. Delays or complete failures can occur, which must be taken into account during development. Because messages are exchanged over the network, it is necessary to serialize and deserialize them, resulting in a higher computing load than exchanging data directly within the process space. Due to the variable number of instances of services, there must be possibilities to localize them in the system. In addition, not only a single system or replicas of it must be monitored, but a large number of small services [Wol16].

3.2.2 Technologies enabling Microservices

One of the most important technologies for microservices is virtualization. In the case of microservices, this means pretending to have its own operating system that runs exclusively on one computer. In reality, the application shares the actual hardware with many others. Container virtualization is usually used for microservices. The virtualization happens on the userspace level, so they also share the kernel. This makes the provision of new instances easy, since no new

virtual machines have to be started [Wol18].

There are different ways to communicate over the network and exchange data or events. You can communicate synchronously or asynchronously. In synchronous communication, the system waits for a response from the other party. In more specific terms this means that as soon as a request is made to a service, the caller waits until the request is processed, and a response is returned. In the microservice environment HTTP is mostly used to make requests to APIs. These days REST is mostly used for the design of web interfaces [SR20].

With asynchronous communication, the caller makes a request and does not wait for a response. This is later sent to him as a message via a message broker or, in the event-driven approach, via the channel on which the response is published [MFP06].

3.3 JValue Open Data Service

The ODS originally had a monolithic architecture. Over time, more and more parts of the original monolithic architecture were migrated to a microservice architecture. This means that the ODS has changed from a single autonomous unit to an application consisting of many small services. The monolith has since broken open, but the ODS is still a changing system and is constantly being improved.

This chapter provides an overview of how the ODS fetches data from a data source, transforms it, manages it and makes it available to users. It also describes the various microservices and how they build the architecture of the ODS. Furthermore, it is described how changes in the architecture influence design decisions.

3.3.1 Concept of the ODS

The ODS makes it as easy as possible for a user to retrieve, modify and store data for further use, according to the principle of extract, transform, load (ETL). ETL originates from the concept of data warehouse. In this process, data from different sources are transferred into a system. Therefore, it is irrelevant whether the data is available in homogeneous or heterogeneous form. In the transformation step, the data is transformed into a structure that allows easy access and storage. Load refers to the transfer into a target database for later access [Den+16]. Since open data is not available in a standardized form, such processes are indispensable for its meaningful use. As a result, the ODS must provide ETL and it must be made as easy as possible for a user to use the system. For this purpose, the ODS provides several microservices that work together to fulfill the task. Section 3.3.2

goes into more detail on the individual components. There are services for fulfilling the core tasks, including the *Datasource-Service*, *Transformation-Service* and *Query-Service.* These services are responsible for retrieving data from external sources, modifying it and then persistently storing it for later use. Due to the microservice architecture, additional components are required to ensure a workflow that runs smoothly. These include a load balancer, scheduler and message broker. A web interface and a service for notification of changes are available for ease of use.

3.3.2 Microservices of the ODS

The ODS is a system in ongoing development and as such new functionalities are constantly being added or the existing system is being revised. While working on this thesis a lot has changed in the system, affecting the different microservices. The attempt was made to provide the most current possible view of the system. In addition, recent and upcoming changes are mentioned. It must be particularly emphasized that the communication within the system has changed from a request response model via REST to an event-driven model. In addition, the architectural change has moved away from a workflow orchestration to a workflow choreography. Away from the scheduler as the central control element within the architecture to the principle that each service is independently responsible for its interactions within the system. This has resulted in a major change within the system.

The ODS consists of a number of microservices that together form the system. These are described in the following.

Datasource-Service

The *Datasource-Service* is responsible for the extract process. This was formerly known as the *Adapter-Service* but has been renamed to better describe its task. It is responsible for the administration of *datasources* and for the collection of the data they describe. *Datasources* describe the locality of data, their metadata and the times of collection. The configuration is done by a REST interface and offers the ability to create, change or delete *datasources*. Furthermore, the *Datasource-Service* is responsible for the collection of the data that a *datasource* describes. Whenever new data is available, this is announced within the system. The communication with the other microservices takes place using event-driven communication through the *Message-Broker.*

Pipeline-Service

The *Pipeline-Service* is responsible for the transformation process according to ETL. This service offers a REST interface to configure transformations on *data-*

sources and to apply them when new data is available. This information is made known to it via the *Message-Broker.* For the transformations JavaScript code is used, which is executed in an isolated environment. After a transformation has been successful, the result is made known to the system via the *Message-Broker.*

Query-Service

The *Query-Service* is responsible for providing the data. This includes storing the data persistently. In addition, it makes it available to a user via a REST interface. As soon as new transformed data for a datasource are announced via the *Message-Broker,* they are stored in a *Postgres*[1] database.

Notification-Service

The *Notification-Service* ensures that a user is informed about new transformed data via external services. The information about new data is made public via the *Message-Broker.* The configuration is again done via a REST interface. Currently, configurations for *Firebase*[2] or *Slack*[3] are offered. It is also possible to implement own webhooks.

Scheduler

The *Scheduler* is responsible for informing the *Datasource-Service* when a *datasource* is due to be collected. Before the changeover to an event-driven approach to communication took place, the *Scheduler* had a far-reaching coordination task. It was necessary to use constant polling to ask the *Datasource-Service* whether there were any new changes to the *datasources.* In addition, the *Scheduler* initiated when the *datasources* were to be collected and then informed the other components in the system. The only thing that is left of the orchestration is informing when *datasources* should be collected. In the previous version the *Scheduler* was a potential bottleneck in the system. This has been solved with the event-driven approach. Furthermore, the complexity of the *Scheduler* has been reduced.

Message-Broker

The Advanced Message Queuing Protocol (AMQP) Message-Broker provides the infrastructure for the internal communication between the various microservices. As mentioned in chapter 2, the microservice architecture requires some form of orchestration. With the introduction of the *Message-Broker,* this task was taken

[1] https://www.postgresql.org/
[2] https://firebase.google.com/
[3] https://slack.com/

out of the scheduler. As *Message-Broker RabbitMQ*[4] is used. For communication with *RabbitMQ* AMQP is utilized. AMQP is an open network protocol for application-level communication [Vin06]. The *Message-Broker* provides channels on which services can publish data. Other services can subscribe to these channels and are informed about new data. They can then decide independently whether and how they react to it.

Load Balancer and Reverse Proxy

Traefik[5] is used as a load balancer and reverse proxy. As described in Chapter 3, a major advantage of the microservice architecture is its dynamic scalability. In order to take advantage of this, it is necessary that requests to the services are distributed evenly. For this purpose, *traefik* serves as an additional layer for communication with the services by a user. Each type of service is provided for the outside world under a location and distributed to the individual instances of these microservices.

Web-Client

The above-mentioned components of the ODS are to be assigned to the server side. The communication with the ODS is performed via REST interfaces, which would be very difficult for a human user to use. In order to achieve greater acceptance for the ODS, there is a web client that provides a user interface. *Datasources* can be created and modified via this. Transformations to *datasources* can be configured in the form of *pipelines*. In addition, the data assigned to a *pipeline* can be viewed.

3.3.3 ODS Workflow

In order to select work packages for the implementation of the requirements, it is important to be aware of how the processes within the ODS were at the beginning of this thesis. In addition, with the change to an event driven approach to communication, a major change was imminent. Since *parameterizable datasources* are an important functionality, desired by users of the ODS, it was important to integrate them into the system as quickly as possible. Extensive planning and foresight was therefore required as to how the division and sequence of changes would be integrated into the system. First it is shown how the processes within the ODS were at the beginning of the work. Then what changes had been assumed that influenced the design decisions. Finally, which decisions result from it.

[4] https://www.rabbitmq.com/
[5] https://traefik.io/

Workflow at beginning

At the beginning of the thesis, the internal communication of the ODS ran via a request response model. Figure 3.2 shows the components involved in the ODS.

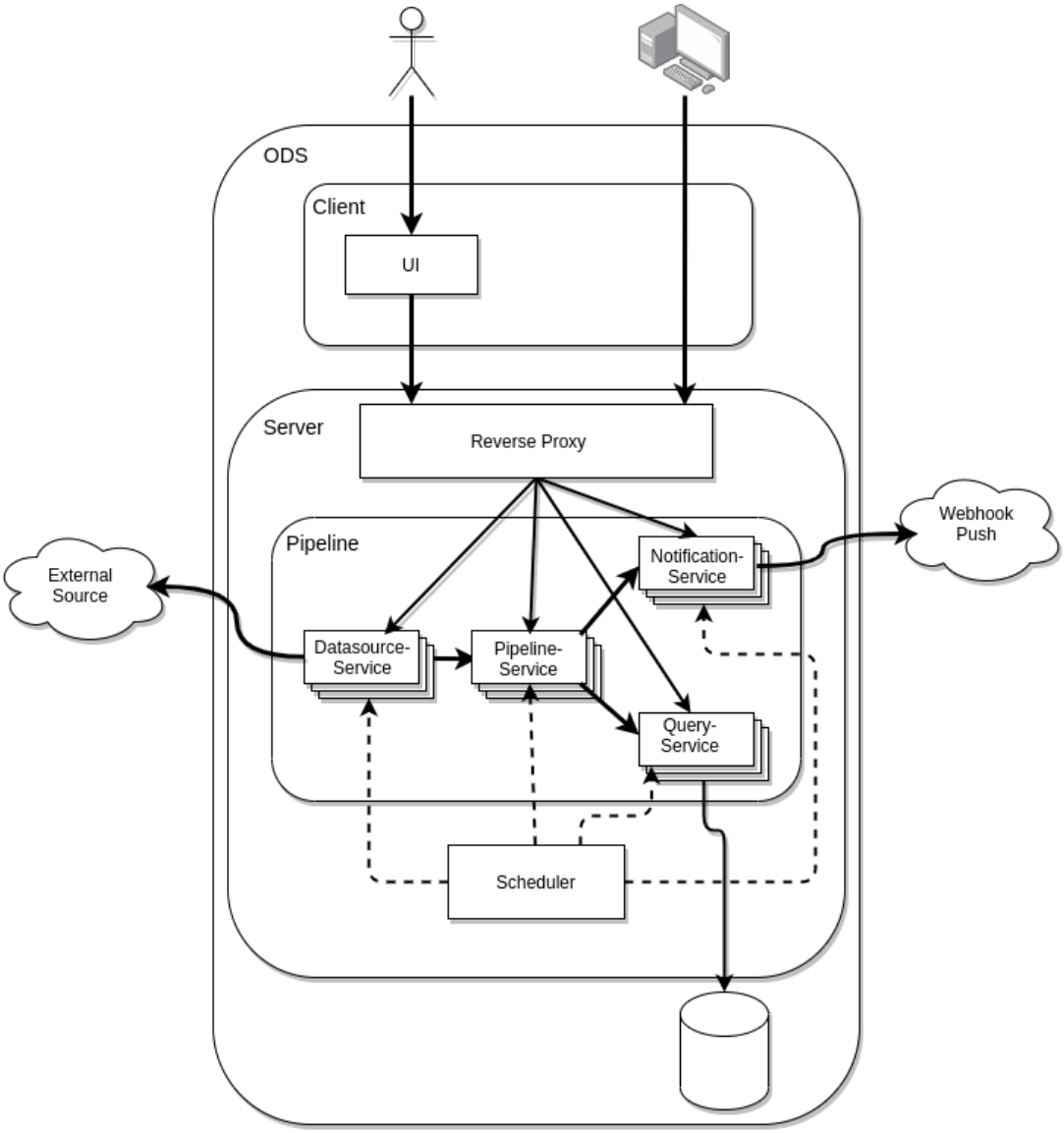


Figure 3.2: Architecture of ODS at beginning of thesis

The interaction with the ODS takes place via a REST interface. It is either used by a human user via the UI or addressed directly via HTTP. A *pipeline* describes the process of fetching data from the system, applying one or more transformations to it and then saving it. Therefore, a *datasource* is first defined at the *Datasource-Service.* It is determined where the data is located, and which protocol is used. In addition, metadata is added to a *datasource,* such as the license

model of the data. When creating a *datasource,* it is determined in which period the data should be fetched. The model of a *datasource* is sent to the *Datasource-Service* as a JavaScript Object Notation (JSON) structure via a HTTP POST request.

The *Scheduler* polls the *Datasource-Service* to obtain information about the latest changes to *datasources.* This gives the *Scheduler* the information when a *datasource* is to be fetched. The *Scheduler* handles the task of orchestration within the system. It triggers the collection with a request to the *Datasource-Service* and waits for a response about the location of the data.

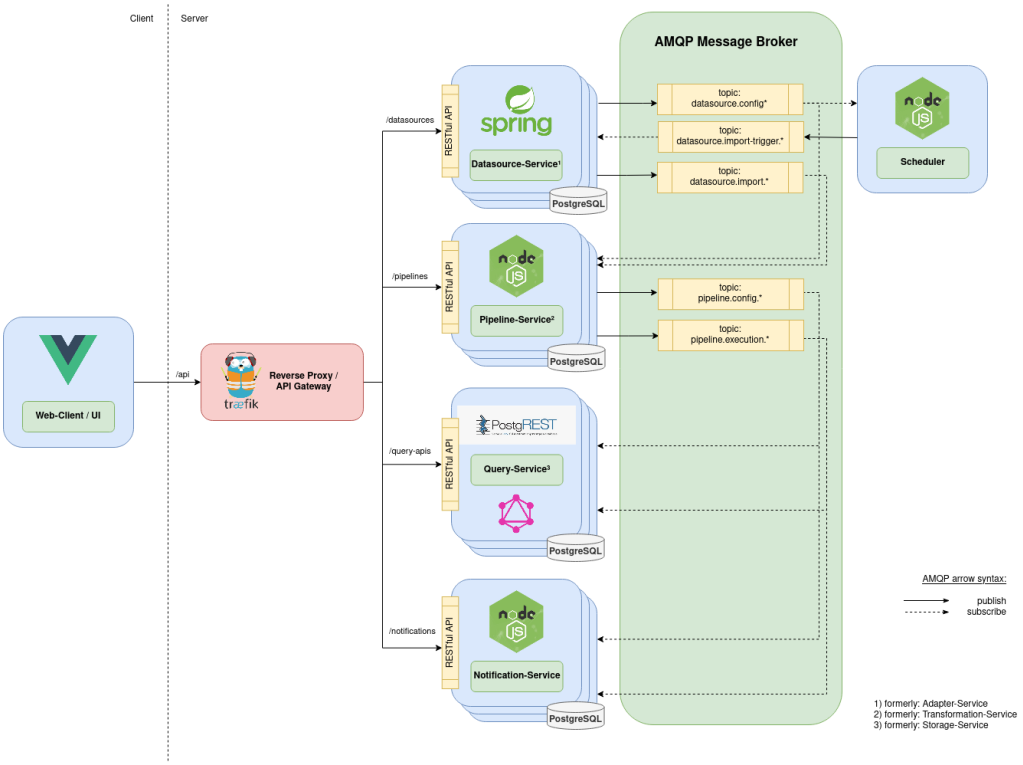
In the *Pipeline-Service,* a user can define transformations on *datasources.* The *Scheduler* informs the *Pipeline-Service* about the new data of the *datasource.* The *Pipeline-Service* then applies all defined transformations and responds to the *Scheduler.*

The *Scheduler* instructs the *Query-Service* to store the data. It also informs the *Notification-Service* about the transformed data. If a user has defined notifications about this, these are sent out.

Workflow changes

The change to an event-driven architecture for internal communication changes the processes within the system. Mainly tasks will be removed from the scheduler. Figure 3.3 shows the process after the switch to RabbitMQ as the *Message-Broker.* Internal communication no longer takes place via the request response model. The publish subscribe model is used for communication with the various components. The exchange of messages takes place via channels that have a topic identifier. Services can publish on these channels and others can subscribe to them. When new messages are available, they are informed.

The *Datasource-Service,* which is configured by the user via a REST interface, is still responsible for the administration of *datasources.* It no longer propagates changes to *datasources* directly to the *Scheduler* but via the *Message-Broker.* This removes the coupling between the components. The *Scheduler* continues to manage the times for collecting data sources and notifies the *Datasource-Service* of this via a *Message-broker.* Information about the successful collection is no longer passed on to the *Scheduler* but made known via the *Message-Broker.* This enables interested components in the system to react to it. In the case of the ODS, this is the *Pipeline-Service,* which executes the transformations on the retrieved *datasources* and makes the result known via *Message-Broker.* The subscribed *Query-Service* and *Notification-Service* react to this and save the data in storage for use by the user and notifies the user if required.



Reference:
`https://github.com/jvalue/open-data-service/blob/master/doc/service_arch.png`

Figure 3.3: Architecture after switching to RabbitMq

4 Architecture and Design

This chapter addresses design decisions that form the basis for the implementation. First, it is evaluated how parameterizable data sources can be modelled. Subsequently, it is argued how the endpoints of the system can be designed to trigger a manual data import. Finally, it is discussed how transformations can be applied to manual data imports.

4.1 Modeling Parameterizable Datasource

This section describes how *parameterizable datasources* can be transferred to the ODS. It was considered how to integrate *parameterizable datasources* into the system in addition to the existing non-parameterizable *datasources.* In addition, how the concept of a *datasource* can be extended to *parameterizable datasources.*

For both considerations, it is important to be familiar with the creation of traditional *datasources.*

The first step is to consider what information a user must define for an existing *datasource.* This is usually done in the *Web-Client.* The UI guides a user through the necessary steps. The sequence is:

1. Assignment of a name
2. Location and representation of the data
   (a) Specifying the protocol
   (b) Specifying the URL
   (c) Specifying the encoding
   (d) Determination of data format
3. Adding Metadata
   (a) Description of a datasource
   (b) The author

(c) The license

4. Defining the point in time and frequency of a data import

The name makes it easier for a user to differentiate between the various *datasources.* For a meaningful use of *parameterizable datasources*, names should also be assigned to them. The metadata fields should be retained.

In order to investigate how a parameterizable data source could be modeled, it was decided to create a prototype for the UI based on the creation of the already existing data sources.

For the prototype, the UI for creating *datasources* was copied to make adjustments. It is obvious here that a *parameterizable datasource* must still have a name and it must be possible to assign metadata. These can be adopted in this way. Adjustments are necessary for the localization of the data. In addition, it is not necessary to fetch *parameterizable datasources* periodically, because they should be addressed manually. For this reason, the modelling of *parameterizable datasouces* focusses on the localization of the data.

First an attempt was made to model how parameters can be defined within a URI. For this purpose, a regular expression was used, which recognizes potential parameters in the structure of a URI. Variable parts are located both in the path, which is used for REST, and in the query part of a URI. For the parameters, elements are dynamically displayed in the UI, which allows a user to configure them. Figure 4.1 shows the UI after entering a URI and the parameter elements generated from it. The user can specify whether the parameter is a parameter he wants to configure or is a fixed part of the URI. In addition, it is possible to define which identifier is given to a parameter and which value is to be assigned to it.

By designing the prototype, experience was gained that lead to two different design considerations. These are shown in the following.

The localization of the data is done via the URI of a data source. To fulfill the requirements of chapter 2, this URI should have variable parts that can be named. As shown in the prototype mentioned above, this can be realized by a regular expression, which potentially filters out variables. A parameter must be seen as a separate data structure as part of a URI. A parameter needs an identifier and a default value to fulfill the requirements of section 2.1.1. A URI needs to be modelled in a more complex way then it is done in the existing *datasources*, where they are represented by a simple string. It consists of a sequence of static parts combined with dynamically filled gaps. Thus, parameterizable *datasources* differ from existing ones and have to be considered separately.

The UI prototype has revealed weaknesses in the first approach to modeling a *parameterizable datasource.* By specifying a regular expression for the separation

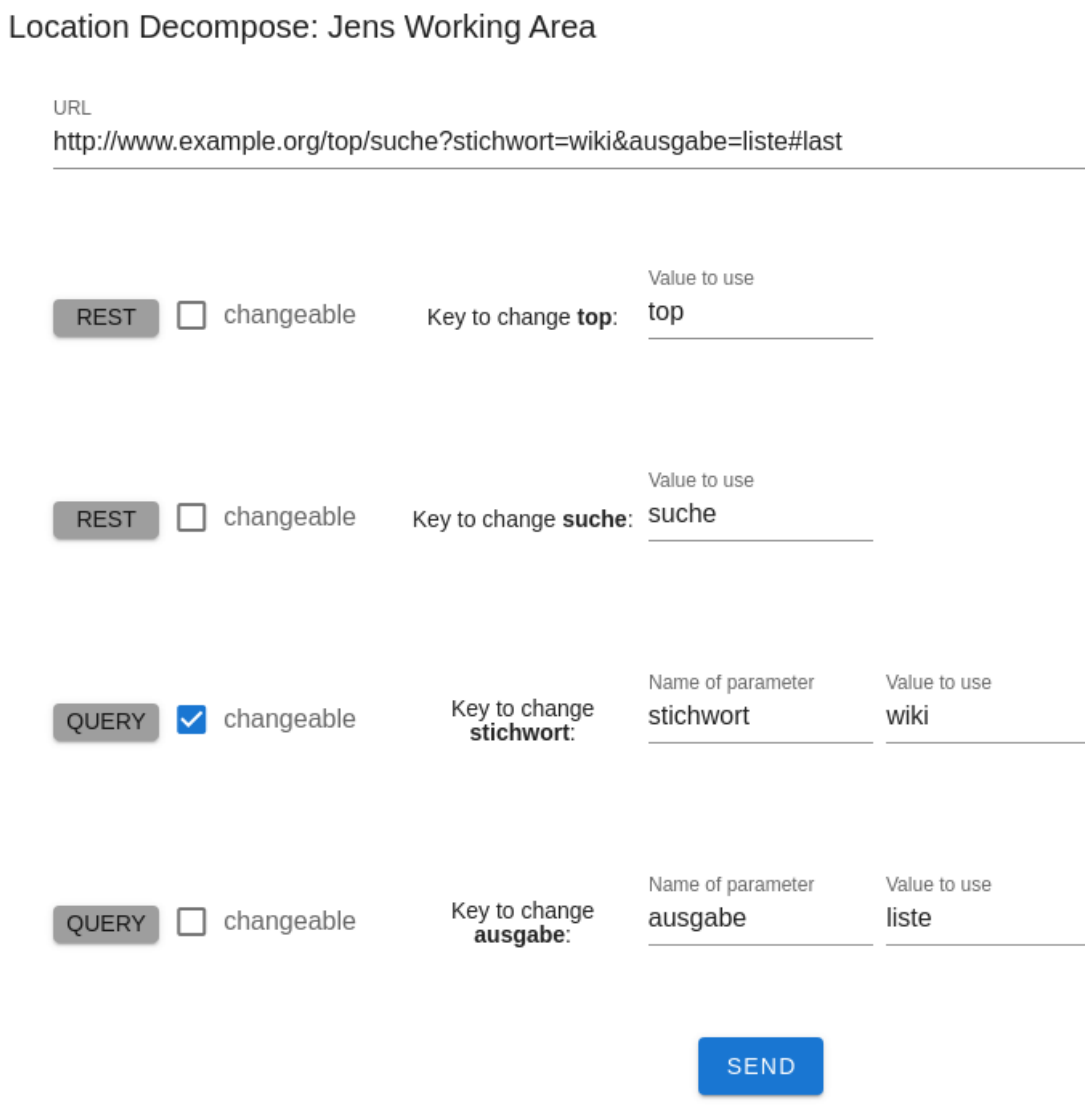


Figure 4.1: UI prototype for extracting parameters from a URL

of parameters, the UI takes away configuration possibilities from the user. To address these, it was examined how URIs are structured. As described in section 3.1 a URI has defined syntax and grammar rules. The idea is to encode free parameters directly into the URI. To make this possible, strings are required that cannot be part of a URI. These include the curly brackets. The beginning of a parameter can be defined with an open curly bracket and the end of a parameter with a closed curly bracket. The character string between the curly braces is used as an identifier. Then it is sufficient to replace the areas within a URI with the desired string. This makes it even possible to replace the authority within a URI. A possible use for this would be to react on address changes of servers without having to create a new *datasource.* This would meet the requirement that open parameters can be defined within a *datasource* without restricting which part of a URI should be variable. By encoding parameters in a URI, it is possible that existing *datasources* can be expanded into *parameterizable datasources.*

Depending on whether *parameterizable datasources* are considered as an independent concept or as part of the existing *datasources,* different changes to the ODS system are necessary. These changes will be presented below.

4.1.1 Independent design for parameterizable datasources

The first consideration is how *parameterizable datasources* can be implemented for the ODS in addition to the existing *datasources.* Since this is a new functionality, a new microservice should be introduced. This service is responsible for the administration of the *parameterizable datasources* and their configuration. In addition, it is responsible for data import and thus also for filling parameters with values. If data from these *datasources* are available, the system must be informed. This can be done as a new topic via the *Message-Broker.* Other services that are interested can subscribe to this topic.

For the configuration of the *parameterizable datasources* by the user new REST endpoints must be defined. These need to have the same functionality as the *Datasource-Service.*

4.1.2 Datasources as a general concept

As described above, there is the possibility to consider *datasources* as a general concept by encoding the parameters. This requires changes to the existing *datasources.* Parameters must be made known by braces or other not allowed characters. Default parameters must be defined during the configuration of a *datasource. Datasources* can then be created using the endpoints already existing in the system. The *Datasource-Service* is responsible for this. The validation takes place within this service. For the implementation of *parameterizable datasources,* the concept must be made known to the service. This means that,

together with values for the parameters, a *datasource* can be created that describes a resource that can be requested via the network. This task has to be fulfilled by the service.

4.1.3 Design decision

Both changes were necessary to meet the requirements 2.1.1 and 2.2.2.

The concept from section 4.1.1 has disadvantages. A separate service for parameterizable *datasources* would increase the maintenance effort. It would also have to be built from scratch. This creates an additional component within the distributed system that can fail. In addition, other services have to be modified, which contradicts the domain-driven design that only one service has to be touched to implement a new feature.

It was decided for the design from section 4.1.2 to consider *datasources* as a general concept. A big advantage here is that the remaining services of the ODS work directly with the data import from *parameterizable datasources,* as they appear externally as a structure that is already known. Since it is already possible to apply transformations in the form of *pipelines* to *datasources,* this is also possible automatically for *parameterizable datasources.*

4.2 API Design for manual data import

Depending on the decision from section 4.1 it has to be considered how a data import can be initiated. Since it was decided not to differentiate between existing *datasources* and *parameterizable datasources,* the behavior must be the same for both. A REST endpoint has to be provided because a user from outside should trigger it. It has to be examined how parameters are to be passed. Thereby it has to be considered how values are assigned to the parameters and passed to the system. Two different possibilities are shown below.

4.2.1 Parameters as query strings

The first consideration is to provide a HTTP GET endpoint on the system, passing parameters via the query string of the URI. With the query string it is possible to describe key-value pairs. If a request is sent by a user, a data import should be initiated. If parameters are defined on a *datasource,* the query string should be evaluated. For this purpose, the *Datasource-Service* must provide the endpoint. When a query reaches the service, the service must evaluate it and then trigger the data import. On success, the system is informed, and the user is answered with the data.

4.2.2 Parameter transfer as JSON object

Another modelling would be via a HTTP POST endpoint. For parameter passing the body of the http message is utilized. The *Datasource-Service* must evaluate the data and use it to fill the free parameters of a *datasource.* Afterwards the data import should be initiated and if successful, the user should receive the data.

As in the previous section, the endpoint must be provided by the system. The *Datasource-Service* is responsible for this. If a data import was successful, it is announced within the system.

4.2.3 Design decision

From a functional point of view, both designs would work. With both it is possible to initiate the data import manually and to specify the free parameters for parameterizable data sources. An API is defined for a trigger mechanism and it is possible for a user to fill parameters with values during a manual data import. This means that both points of the requirements from sections 2.2.1 and 2.2.2 are fully met. Additionally, it is possible to execute the *pipelines* defined on *datasources* as required in section 2.2.3.

The query string has disadvantages. If values are being passed, it has to be taken into account that they are encoded correctly. An example is the equals sign, because it serves as a separator of the key-value pairs. This restriction does not exist if the mapping is done as an object over the body. In addition, the functionality would not match the GET semantics. It requires that the state of the system should not change [Fie+99]. This happens through the data import, as the transformed data is saved by the *Query-Service.* Therefore, the endpoint should be implemented as described in section 4.2.2.

4.3 Enabling transformation for manual data import

The concept of integrating the *parameterizable datasources* into the existing ones makes it possible to use the existing services of the ODS, consequently also the *Pipeline-Service.* It applies the defined transformation as soon as a data import has been announced. To meet the requirements of section 2.2.4, the system needs to be adjusted. In order to receive transformed data, the *Query-Service* must be contacted as soon as the *pipeline* has been successfully executed. This additional request should be avoided for a user. A mechanism is to be introduced to initiate a data import and receive the transformed data as response. For this purpose, a REST endpoint must be introduced.

In the following, two different ways to achieve this are shown.

4.3.1 Integration into the Pipeline-Service

The *Pipeline-Service* is already responsible for the transformations and has all data a user potentially requests. This makes it possible to integrate the functionality there. The service must provide an additional REST endpoint that allows manual data import for the targeted *pipeline.* A user should be able to specify values for parameters based on the endpoint described in section 2.1.1. A manual data import is to be initiated on the *Datasource-Service.* The connection must be maintained until the data arrives. With the data received in response, the selected transformation can be executed. After successful transformation, the data is sent back to the user as a response.

4.3.2 Stand-alone microservice

Another possibility for the implementation is to outsource the task to a new independent microservice. As above, this would have to provide a REST interface for user interaction. The endpoint should refer to a *pipeline.* Since the newly introduced microservice has no information on which *datasource* the *pipeline* was defined on, this must be requested from the *Pipeline-Service.* A manual data import can be triggered with the identifier of the *datasource.* The manual data import automatically informs the *Pipeline-Service* and applies the transformation to the data. The *Pipeline-Service* already publishes this via the *Message-Broker* for the *Storage-Service.* The new microservice can also subscribe to this. As soon as it is informed about the successful transformation, it can extract the data and send it back to the user as a response. The user session must be maintained from the request to the final response.

4.3.3 Design decision

The implementation effort for the first design proposal is lower because only one additional endpoint needs to be introduced and the actual logic already exists in the *Pipeline-Service.* A new service introduces an additional component into the system, which may fail and further complicates the architecture.

One problem is the maintenance of a user session. Keeping the session alive adds additional load on the *Pipeline-Service* and is not one of its core tasks. Currently, the *Pipeline-Service* does not communicate with the *Datasource-Service.* Furthermore the *Pipeline-Service* does not communicate with the *Datasource-Service* at the moment. This is not known to the *Pipeline-Service.* The *Pipeline-Service* is decoupled from the *Datasource-Service.* This restriction would have to be softened.

A new microservice would circumvent this. Its principal task is to maintain a user session and forward the requests to the corresponding services. This ensures that the data source service only makes new data imports known to the system via the *Message-Broker* and the *pipeline* service still does not have to communicate with it. For these reasons it was decided to implement the design from section 4.3.2.

5 Implementation

This chapter describes the implementation process of the design decisions made in chapter 4. The goal is to meet the requirements from chapter 2. New functions are implemented in the smallest possible independent sub-steps in such a way that they can be easily integrated into the system. The integration was done in the github repository of the ODS[1]. Development was performed on a separate fork.

First, *parameterizable datasources* are integrated into the system, then the functionality of manual data import. Next, a new microservice is integrated into the system, which provides transformed data for manual data imports to a user. Additionally, the new functionalities are documented, and tests are written. For easy configuration by users, the new functions are also provided within the UI.

5.1 Integrating Parameterizable Datasources

The system shall be extended by *parameterizable datasources*. After the considerations in section 4.1.3 it was decided to extend the existing *datasources* with this functionality. The *Datasource-Service* is responsible for the management of *datasources* within the system, therefore the implementation must be performed within the code base of this microservice.

5.1.1 Adapt Datasource Model to support parameters

In order to make *parameterizable datasources* valid without passed runtime parameters, it was necessary to adapt the model of a datasource so that it includes default values for the free parameters during creation. The model used for configuring *datasources* is shown in fig. 5.1. In this context, the question arose whether default values should be modelled by a separate attribute or be by extending one of the existing attributes. It could be argued that a `DatasourceTrigger` describes the execution of a request, making it a potential class for the extension

[1] `https://github.com/jvalue/open-data-service`

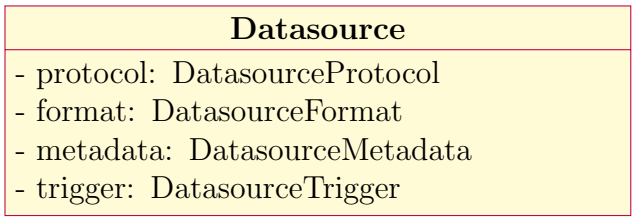


Figure 5.1: Datasource Class

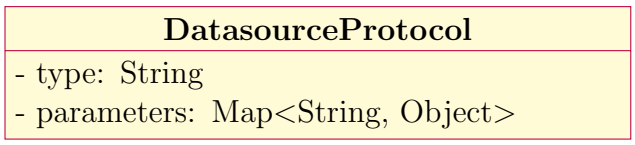


Figure 5.2: DatasourceProtocol Class

of the default parameter functionality. Another possibility is to locate it within the protocol attribute. In this field the protocol of the URI is defined, but also the URI itself is specified. Since default parameters are necessary to complete the URI, it was decided to store them directly in the `DatasourceProtocol`. The structure of a DatasourceProtocol is described in fig. 5.2. It is kept very abstract for flexibility reasons. Any object can be stored under a string key. This is necessary because a URI has different syntax for different protocols, as described in section 3.1.

Within the parameter map the default parameters should be stored. For this purpose, a new model, the *Runtime-Parameters* was created. When performing a HTTP POST request on the the trigger endpoint of a *datasource,* the runtime parameters object is used to specify the open parameters of a *parametrized datasource*. It is also intended to be used to define the default values. For these changes a map as data structure was the obvious choice. To avoid that future model changes lead to code modifications in many places, a separate class for runtime parameters was created.

Since strong typing is eliminated in the protocol, validation must be done manually. This happens within the *Import* class. To support a new protocol, it is necessary to create a new importer, which inherits from an abstract superclass `Importer`. There are currently the generic `Importer` class and the `HttpImporter` class for the HTTP protocol. For the implementation of the default parameters it was necessary to make changes to these two classes.

The parameter list of an importer defines the string keys and the corresponding object class for a protocol. This checks if they match when adding a *datasource* to the system. It would reject the protocol validation if pairs are missing or not defined. In the case of an original *datasource,* it was necessary to soften this re-

striction, since these have no default parameters. Therefore potential parameters were introduced. These can be missing during validation but are accepted as valid parameters if they are necessary.

For the `HttpImporter` potential parameters are defined under the string key `defaultParameters`. It is necessary to override the newly introduced method `getRequiredParameters()` to make this distinction to the required `location` and encoding `parameters`. The `RuntimeParameters` class is used to describe the object to be stored for the key `defaultParameter`.

After these adjustments it is possible to create *datasources* with default parameters. Tests were written for all additional functionalities. Furthermore, the documentation of the ODS was extended by the additional changes.

5.1.2 Build UI Component for Datasource configuration

The UI component was integrated into the productive system at a later stage but was already written during the implementation of the *datasources.* A developer branch was used for this purpose. This made it possible to test the configuration of the backend, but a user was not yet offered a feature that he could not use. The real benefit of *parameterizable datasources* comes into play as soon as parameters can be specified at the time of data import.

Within the web client the component responsible for the configuration of *datsources* was extended. Creation and modification of a *datasource* behave the same way for a user of the UI and is further summarized as configuration of a *datasource.*

In order to expand the configuration to include *parameterizable datasources*, default parameters must be definable. Therefore, parameter components have to be provided within the UI, which allows to name a parameter and assign a value to it. Two text fields per parameter are displayed. For this purpose, the UI was extended by such a component. As soon as a part of the URI is marked as variable by curly brackets, a parameter component appears for this part within the UI. The area enclosed by the brackets is used as a standard identifier and default value for the parameter. A user can still change both independently of each other. The remaining creation process behaves as before. The changes are then transferred to the system via the REST interfaces of the *Datasource-Service.*

5.2 Trigger Endpoint

This section shows the process for enabling manual data import. For this purpose, the considerations from section 4.2.3 are implemented. These are implemented iteratively in small pull requests to simplify the review process. First, the end

point was integrated into the system. Then the functionality of the data import of existing data sources was added. Finally, the support of parameters was added.

5.2.1 Trigger Endpoint Implementation

To provide an additional REST endpoint it is necessary to define it in the *Datasource-Service.* Its REST endpoints are defined by annotations in the `DatasourceEndpoint.java` as shown in listing 5.1. For trigger endpoints, this is done with `@PostMapping("/{id}/trigger")` above the method signature. Where `{id}` is a variable value provided by the caller and interpreted by the *Datasource-Service* as the ID of a *datasource.* This triggers the method `getData()` for each POST access for the respective *datasource.* The data transferred via POST in the body can be missing or must be *Runtime-Parameters.* These are discussed above in more detail in section 5.1.1. The framework tests whether the call is valid. In case of an error it sends back a error response. The `DatasourceManager` class contains the logic that is used by the REST endpoints. The `trigger` method is added to the class in the first step as an empty method, which does not offer any functionality yet. This makes it possible to test the communication with the endpoint.

```
1  @PostMapping("/{id}/trigger")
2  public DataBlob.MetaData getData(@PathVariable Long id,
3                                   @Valid @RequestBody (required =
       false) RuntimeParameters runtimeParameters) {
4    try {
5      return datasourceManager.trigger(id, runtimeParameters);
6    } catch (IllegalArgumentException e) {
7      throw new ResponseStatusException(HttpStatus.NOT_FOUND, "No
       valid Datasource for id "+ id);
8    } catch (InterruptedException e) {
9        throw new ResponseStatusException(HttpStatus.
       INTERNAL_SERVER_ERROR);
10   }
11
```

Listing 5.1: Trigger Endpoint

After the communication works, the functionality for `trigger()` is added. The implementation is described in the next section.

5.2.2 Manual data import

For the conversion of the data import the trigger() method must be implemented. This is shown in listing 5.2. The data import is done with an `Adapter` via executeJob(). This selects the appropriate interpreter of the data, such as JSON or Extensible Markup Language (XML). To fetch data, a *datasource* must be

converted to an `Adapter`. An `AdapterConfig` is required to create an adapter. This is an abstraction of the *datasource* and describes the locality and representation of the data. The `AdapterConfig` contains all the information needed to trigger a data import.

```
public DataBlob.MetaData trigger(Long id, RuntimeParameters
    runtimeParameters) throws InterruptedException {
     AdapterConfig adapterConfig = getParametrizedAdapterConfig(id
    , runtimeParameters);
     try {
         Adapter adapter = adapterFactory.getAdapter(adapterConfig
    );
         DataBlob executionResult = adapter.executeJob(
    adapterConfig);
         DatasourceImportedEvent importedEvent = new
    DatasourceImportedEvent(id, executionResult.getData());
         publishAmqp(RabbitConfiguration.AMQP_IMPORT_SUCCESS_TOPIC
    , importedEvent);
         return executionResult.getMetaData();
     } catch (Exception e) {
         ImportFailedEvent failedEvent = new ImportFailedEvent(id,
     e.getMessage());
         publishAmqp(RabbitConfiguration.AMQP_IMPORT_FAILED_TOPIC,
     failedEvent);
         if(e instanceof IllegalArgumentException) {
             System.err.println("Data Import request failed.
    Malformed Request: " + e.getMessage());
             throw e;
         } else {
             System.err.println("Exception in the Adapter: " + e.
    getMessage());
             throw e;
         }
     }
}

```

Listing 5.2: Trigger Implementation

Via `publishAmqp()` an event is published on the Message-Broker in case of success or failure. If the process is successful, it contains the imported data and the identifier of the *datasource*. The *Pipeline-Service* can react to this if a transformation has been specified for the *datasource*.

When a request is made to the *Datasource-Service* via the REST endpoint described in section 5.2.1, the data import for the corresponding Datasource is initiated. To retrieve the *parameterizable datasources*, it is necessary to resolve the URI when converting a datasource to an `AdapterConfig`. This is described in the next section.

5.2.3 Support of parameters

The data import is supposed to work also with *datasources* that contain parameters. The function from listing 5.2 serves as a basis. To get the necessary `AdapterConfig` for the data import the method `getParametrizedAdapterConfig` has already been introduced. This method gets the corresponding *datasource* for an identifier from the database and passes the runtime parameters to it. In case of *datasources* without parameters or if the default parameters are used, the value can be empty. This method currently only converts *datasources* without parameters. It considers a URI as it is, including the curly brackets. The data import of a *parameterizable datasource* would fail with an error message in this case. The conversion of a *datasource* to an `AdapterConfig` is now adjusted.

The URI field of the protocol is filled with the parameters. For this purpose, the default parameters are copied. Afterwards their values are updated by the passed parameters. This makes it possible that parts of the URI are overwritten by the values of the default parameters and others by the passed parameters. The values that are specified by the user via the trigger endpoint have a higher priority.

For the replacement process the URI string is searched for the parameter identifiers surrounded by the curly brackets. These are replaced by the values of the parameters from the resulting map.

If this is successful, a corresponding `AdapterConfig` is returned with which the data import can be triggered. This allows to specify values of a *parameterizable datasource* at the time of the data import.

5.3 Microservice for transformed data

The next step is to fulfill the requirements of section 2.2.3. For this purpose, a new microservice is introduced. This section describes the process of integrating a new service into the ODS. Then it is described how this is extended by the functionality to provide transformed data of a manual data import.

5.3.1 Provide skeleton for service

After consultation with the active developers of the ODS it was decided to write the microservice in TypeScript. This is due to the great experience of the developers with this programming language. In addition, the latest services have been developed in TypeScript and offer a good introduction to how a service can be provided within the ODS.

First of all, a TypeScript project must be created. This is named *pipeline-trigger*.

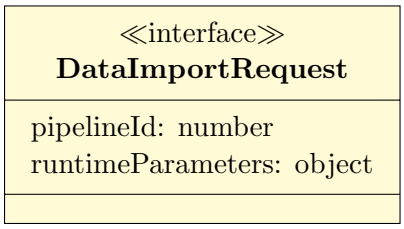


Figure 5.3: DataImportRequest Model

This is accomplished by the JavaScript Runtime Node.js[2]. The initialization of the project is done by the commandline tool npm, which is delivered with Node.js. This tool is also used to install dependencies and offers commands to start the server.

Services for the ODS must run in their own Docker container[3]. Therefore, it is necessary to describe how the service should be built and started. A Docker file is used for this. For the new service such a Docker file was created. This allows the service to be executed in virtualization.

For the configuration of an application, which is composed of several Docker containers, a docker-compose file is used. This file describes how the new service fits into the system. Such a file is available for the execution of the ODS. This file was modified for the new *Pipeline-Trigger-Service*. Necessary environment variables were set, dependencies were defined and metadata for the reverse proxy was set.

After these steps the new service can be started as part of the ODS. It does not yet provide any functionality but can be tested against the other services.

5.3.2 Introducing the REST endpoint

Next, the corresponding endpoints were added to the service. Each service within the ODS should have an endpoint for the API version. This was provided. Under the base endpoint a user can request if the service is available.

For the actual functionality of the service a REST endpoint was provided. For a request to this endpoint a HTTP POST on `baseurl/dataImport` is used. The body of the message must contain a `DataImportRequest`. This consists of the `pipelineId` and `runtimeParameters` as shown in fig. 5.3.

[2] `https://nodejs.org/`
[3] `https://www.docker.com/`

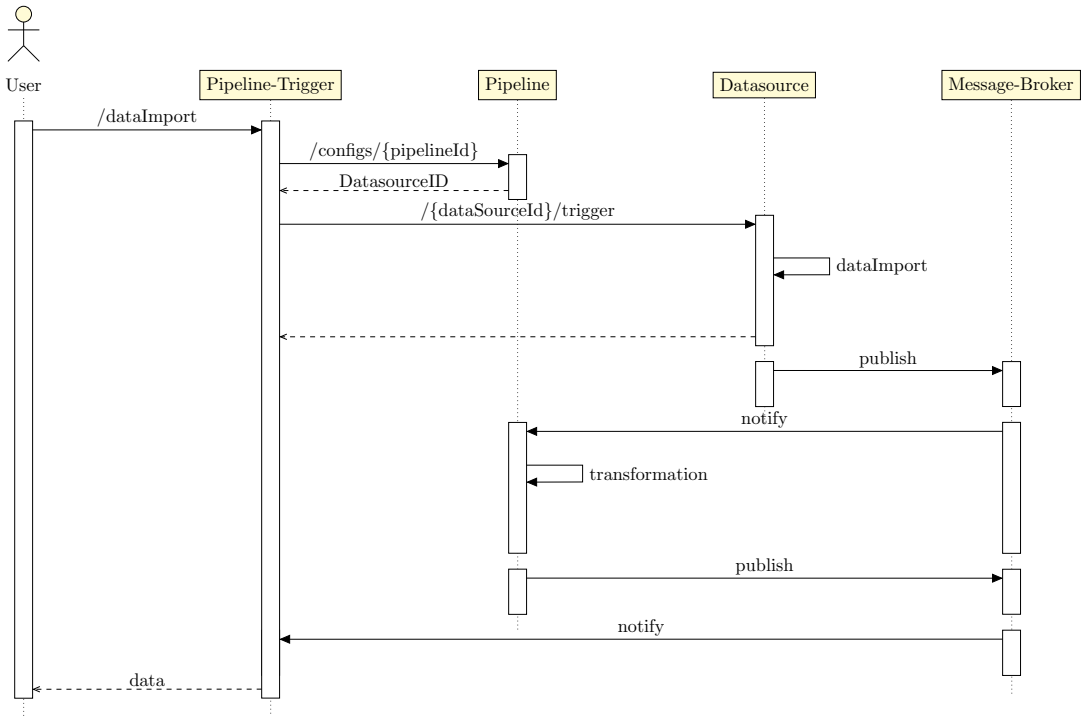


Figure 5.4: Sequence Diagram of a DataImport

5.3.3 Enable parameterizable pipeline

Now that the necessary infrastructure for a new service has been set up, the actual functionality is provided. Figure 5.4 shows a sequence diagram of the data import process. The `PipelineId` is extracted from a request to the service. The `DatasourceId` for which the transformation was created is required for the request to the *Datasource-Service.* Therefore, a request to the *Pipeline-Service* must be made to query it. The `DatasourceId` can be extracted from the response. Then a request is made to the *datasource* trigger endpoint with the `runtimeParameters`. The *Datasource-Service* publishes the successful data import. The *Pipeline-Service* responds to this and applies the transformation. The data is then published as an event via the *Message-Broker.* The *Pipeline-Trigger-Service* waits for this event. It extracts the data from the event and returns it to the user in response.

If the data import takes too long, an error message is sent back to the user. This can happen if there are network failures within the system.

6 Evaluation

In this chapter the requirements defined in chapter 2 are checked for their fulfillment. It is examined whether the requirements are fully, partially or not fulfilled. Furthermore, possible weaknesses of the implementation are discussed and how these can be addressed.

6.1 Configuration of parameterizable data sources

Datasources have been extended by the possibility to mark parts of the URI as variable. This was achieved by encoding parameters using curly brackets. The string enclosed by the braces made it possible to assign names to these parameters. *datasources* have been expanded to include default parameters which, together with the URI string marked as a parameter, can localize a resource. The configuration is possible via a REST interface as well as via the UI. As a result, the two sub-items from sections 2.1.1 and 2.1.2 this requirement were met.

Encoding within a URI gives the user of the ODS the greatest possible freedom in defining *parameterizable datasources*. Errors would occur if protocols that allow curly brackets were to exist in the future. This happens with protocols that do not follow the specification of a URI. For such protocols, albeit improbable, such a case would have to be intercepted. This would be possible with a different encoding of parameters specific to such a protocol. The requirements from section 2.1 are met.

6.2 Execution of parameterizable data sources

An interface was added to the system that enables a user to initiate a data import. The user can assign values for defined parameters at the time of the request. When importing data, these values are used to complete the URI. On success, the response contains the retrieved data.

This functionality is used within the UI when creating pipelines. When defining a transformation, it can be tested directly on the actual data.

In addition, a user can also use this interface for manual data import of *data-sources* without parameters. This was not explicitly required, but since data sources have been extended by the concept of *parameterizable datasources*, the functionality is also available for them.

The requirements of sections 2.2.1 and 2.2.2 are fulfilled.

It was required that transformations are applicable to a manual data import. This is possible by publishing an event via the *Message-Broker.* Additionally, it was required that one can initiate a data import with following transformation and receive the transformed data as response. This was achieved by an additional microservice, the *Pipeline-Trigger-Service.* Through an interface requests can be made. Values for parameters can be sent along. A data import is triggered, the selected transformation is executed and sent back to the user as response.

A weakening in consistency is that a query cannot be assigned to the data with certainty. This is due to the distributed architecture of the ODS. One possibility to address this issue would be to give the requests unique identifiers that are passed through the system. This would require the adaptation of all components that are currently involved in the data import process. This could not be done in the context of the thesis. As a result, the service lays the foundation for meeting the requirements of section 2.2.3 but still offers potential for improvement.

Therefore requirement 2.2 is partially fulfilled.

6.3 Still support non-parameterizable data sources

The existing functionality of the ODS has been extended without changing it. Although parameterizable and non-parameterizable data sources share the same configuration process, they can be used as before the changes. The normal periodic pipelines are not affected. Therefore the requirement of section 2.3 is fulfilled.

7 Conclusion

The JValue Open Data Service is a service that was launched to make the consumption of open data reliable, easy and safe. The constantly advancing system picks up on suggestions for improvement from its users. Many users wished to be able to initiate data imports not only according to a fixed schedule but also manually. Others required parts of *datasource* URI to be specified dynamically when querying the data.

This thesis addressed this feedback. Its goal was to enable parameterizable data sources for the ODS. Furthermore, a mechanism should be integrated to manually trigger a data import and dynamically assign values to the open parameters. In chapter 4 the design decisions were discussed. Parameters can be provided by encoding within a URI. For the manual data import an API was defined. It was determined that a user can request transformed data from a manual data import by a newly introduced microservice. Their implementation was then shown in chapter 5.

The artifacts produced during this thesis extends the existing ODS functionality in three aspects: First, *parameterizable datasources* for the system were enabled. Second, a user can trigger a manual data import and specify parameters for a *parameterizable datasource.* Finally, transformed data resulting from a manual data import can be received as a synchronous response to the trigger request. This is enabled by a new microservice.

Due to the distributed architecture of the system, an exact assignment of requests and data cannot be guaranteed. To avoid this, the system must be extended by a one-to-one mapping of requests and data. This will be one of the next steps to improve the JValue Open Data Service.

References

[BFM05] Tim Berners-Lee, Roy T. Fielding and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. http://www.rfc-editor.org/rfc/rfc3986.txt. RFC Editor, Jan. 2005. URL: http://www.rfc-editor.org/rfc/rfc3986.txt.

[Den+16] Michael J. Denney et al. 'Validating the extract, transform, load process used to populate a large clinical research database'. In: *Int. J. Medical Informatics* 94 (2016), pp. 271–274. DOI: 10.1016/j.ijmedinf.2016.07.009. URL: https://doi.org/10.1016/j.ijmedinf.2016.07.009.

[Eco17] The Economist. *Regulating the internet giants - The world's most valuable resource is no longer oil, but data*. (Accessed on 10/20/2020). May 2017.

[Fie+99] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. http://www.rfc-editor.org/rfc/rfc2616.txt. RFC Editor, June 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt.

[HK20] Esther Huyer and Laura van Klippenberg. *The Economic Impact of Open Data Opportunities for value creation in Europe*. Tech. rep. European Commission, 2020. DOI: 10.2830/63132. URL: https://www.europeandataportal.eu/sites/default/files/the-economic-impact-of-open-data.pdf.

[MFP06] Gero Mühl, Ludger Fiege and Peter Pietzuch. *Distributed Event-Based Systems* -. 1st ed. Berlin Heidelberg: Springer Science Business Media, 2006. ISBN: 978-3-540-32653-3.

[New15] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[SR20] Georg Schwarz and Dirk Riehle. 'What Microservices Can Learn From Enterprise Information Integration'. In: Jan. 2020. DOI: 10.24251/HICSS.2020.678.

[Vin06] S. Vinoski. 'Advanced Message Queuing Protocol'. In: *IEEE Internet Computing* 10.6 (Nov. 2006), pp. 87–89. ISSN: 1941-0131. DOI: 10.1109/MIC.2006.116.

[Wol16] Eberhard Wolff. *Microservices - Flexible Software Architecture*. Amsterdam: Pearson Education, 2016. ISBN: 978-0-134-65040-1.

[Wol18] Eberhard Wolff. *Microservices - A Practical Guide*. CreateSpace Independent Publishing Platform, 2018. ISBN: 978-1-717-07590-1.