Hierarchical Open Data Source Import for the JValue ODS

MASTER THESIS

Fischer Benjamin

Submitted on 29 July 2021

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

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Versicherung

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

Erlangen, 29 July 2021

License

Erlangen, 29 July 2021

Abstract

Open Data has become more popular in the last few years due to its value to society. Governments, institutions, companies or individuals can make use of Open Data and add to economic growth or extract new knowledge from publicly available data. The Open Data Service (ODS) is a software developed by the Professorship of Open Source that aims to simplify the consumption of Open Data and make it more reliable.

The goal of this thesis is to extend the functionality of the ODS by the support of hierarchically structured data sources, in particular, File Transfer Protocol (FTP) based data sources. Due to the simplicity and reliability of the FTP, it is an appropriate solution for providing Open Data. This thesis aims to enable the user to explore and configure FTP data sources by developing a new microservice with a proof-of-concept user interface. As a result, consuming Open Data from FTP data sources is simplified and becomes more flexible.

Contents

Acronyms

ODS Open Data Service

FTP File Transfer Protocol

API Application Programming Interface

REST Representational State Transfer

HDS Hierarchical Datasource Service

URL Uniform Resource Locator

CRUD Create Read Update Delete

RPC Remote Procedure Call

HTTP Hypertext Transfer Protocol

SPA Single Page Application

IP Internet Protocol

NAT Network Address Translation

SSH Secure Shell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

IPC Inter-process communication

SOAP Simple Objects Access Protocol

HATEOAS Hypermedia As The Engine Of Application State

1 Introduction

The amount of digital data created increased significantly in the last years, driven by the digital transformation. Due to new sensors, IoT devices, and the rising awareness about the value of data in general, a growing number and variety of data is generated each day. For example, the International Data Corporation estimated that the amount of digital data would increase rapidly in the following years, reaching up to 163 zettabytes by 2025 (Reinsel et al., 2017).

Dealing with this sheer amount of data introduces additional problems regarding storing, providing, accessing, and processing this data. The difficulty of these challenges also depends on the data type, primarily if the data is structured or unstructured, and whether access is restricted or not. Furthermore, the raw data itself is not useful unless it is processed and the encoded information is extracted. Consequently, automated processes have to be developed in order to make the value of the underlying data accessible.

As a developer who, for example, wants to build a new weather application, the process of retrieving the required data can be a tedious task. First, the required data might not be completely available at a single source but could be split between different servers. Second, the data might be available in different formats and could be incomplete sometimes. In addition, it might be required to periodically retrieve the data, e.g., each hour and persist it in a separate database. Consequently, much effort is spent on retrieving the underlying data instead of working on the actual application itself.

Facing these challenges, the JValue ODS is developed by the Professorship for Open Source Software at the Friedrich-Alexander University Erlangen-Nürnberg. The ODS aims to simplify consuming data sources and thereby focuses on Open Data, which is data that "[...] can be freely used, modified, and shared by anyone for any purpose" (Open Knowledge Foundation, n.d.). In more detail, the ODS periodically retrieves, processes, and persists this data from various data sources and provides this data to third-party applications. As a result, the time and effort spent on the overall process of making the desired data available are reduced. Developers then can focus on extracting information from the data by creating new applications or improving already existing software.

This thesis aims to extend the existing functionality of the ODS by hierarchically structured data sources, in particular FTP data sources. Hence, a new variety of data sources will be supported by the ODS, and developers could benefit from the advantages of the ODS when working with data that is available via FTP servers.

In the next chapter, it is outlined which specific problems have to be solved in order to support FTP data sources for the ODS, followed by a summary of the essential technical fundamentals for the context of this thesis. Chapter 4 lists the single objectives that were derived from the previous problem specification. Afterwards, the conceptual solution design is described and potential solution approaches are explained. In chapter 6, the concrete implementation of the new functionality is discussed in detail and demonstrated in chapter 7. Finally, the implementation is evaluated concerning the objectives, and a short outlook is given.

2 Problem Identification

This chapter will explain what specific problems arise when trying to support FTP data sources for the ODS. The underlying goal of this approach is to extend the accessibility of Open Data sources by the ODS due to the positive influence of Open Data. Therefore, it is important to understand the specific characteristics of Open Data. Similar to the Open Knowledge Foundation, the European Commission also emphasizes the value of Open Data in their definition:

> "Open data is data that anyone can access, use and share. Governments, businesses and individuals can use open data to bring about social, economic and environmental benefits." (European Commission, n.d.)

This definition already shows the possible gains which Open Data can provide. Due to that, many Open Data initiatives have been created in order to meet this goal. For example, the number of datasets published by the European Data Portal has more than doubled from May 2016 to August 2019 (Publications Office of the European Union, 2020). Unfortunately, this published data is often barely documented, lacks machine readability, or uses data formats that require proprietary software for further processing (Braunschweig et al., 2012). As a result, the data is often hard to use and thus, can not unfold its true value.

Improving this situation, in particular providing more effortless ways to access and work with Open Data, is a crucial aspect the ODS focuses on. In order to reliably consume a new data source via the ODS, a *pipeline* for this data source has to be configured. This pipeline specifies the data source configuration, its content type, additional metadata, and a (periodic) trigger that defines when the data should be retrieved. According to such a configuration, the ODS will (periodically) fetch the data from the data source and persist it. First designed as a monolith, the ODS and its components were transformed into a microservice-based software architecture (Schwarz, 2019). Over time, the ODS was subject to many engineering theses that focused on improving the software's functionality. However, at the time of writing this thesis, the ODS only supports data sources that are accessible via single Hypertext Transfer Protocol (HTTP) endpoints, leaving many data sources using other protocols like the FTP uncovered.

Contrary to the general conception, the FTP is still widely used, especially for openly accessible data. Due to its simplicity and usage over decades, the FTP poses a reliable and viable solution to provide Open Data. Unfortunately, accessing FTP data sources is fundamentally different from retrieving data from a single HTTP endpoint. Thus, a new microservice shall be implemented that fits into the existing ODS infrastructure. In this context, the following problems arise, which have to be solved in the scope of this thesis:

- How to model hierarchical data sources and FTP data sources in particular?
- How to handle (parallel) FTP connections efficiently?
- How to enable accessing archived files without major effort for the user?
- How to explore data sources manually/automatically?
- How to generate powerful data source configurations for all major use cases?
- How to integrate these configurations into the ODS pipeline mechanism?
- How to achieve scalability of this microservice?

As a result, the user should be able to explore and configure a FTP data source using this microservice. Once the user has defined a list of files that contain the desired data, these files should be downloadable by the ODS, similar to the already existing mechanism for HTTP endpoints, making FTP data sources consumable by the ODS.

3 Fundamentals

This chapter will provide the background knowledge which is mandatory to understand the proposed solution design and its concrete implementation. Due to the engineering focus of this thesis, this chapter is kept as concise as possible.

3.1 File Transfer Protocol (FTP)

The specification of the FTP was released as an RFC standard in 1985. The following excerpt is the first paragraph of its introduction.

> "The objectives of FTP are 1) to promote sharing of files (computer programs and/or data), 2) to encourage indirect or implicit (via programs) use of remote computers, 3) to shield a user from variations in file storage systems among hosts, and 4) to transfer data reliably and efficiently. FTP, though usable directly by a user at a terminal, is designed mainly for use by programs." (Postel & Reynolds, 1985)

This is an accurate summary of the purpose of the standardization of this protocol in 1985. Since then, various extensions have been published, such as introducing new optional commands for authentication (Lunt, 1997) or adding support of encrypted file transfer (Housley & Yee, 2000). Nevertheless, the basic functioning of the protocol is unchanged since its first publication. The upcoming two sections will provide a short overview of how the protocol in general works and its technical limitations.

3.1.1 Communication between the Client and the Server

The FTP defines a standardized communication between a *client* and *server* for file sharing purposes. The communication is implemented using two separate connections, a *control* and a *data* connection. The control connection is for sending/receiving FTP commands, whereas the data connection is used to transfer the actual data, like the content of a file or a directory listing. The conceptual design of the FTP is shown in figure 3.1.

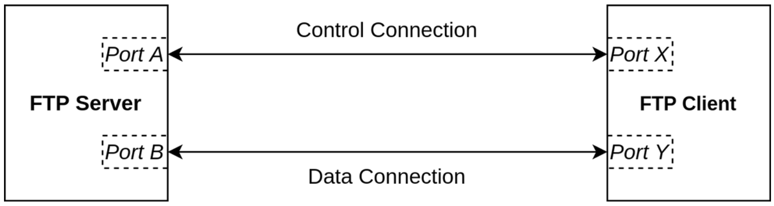

Figure 3.1: Overview about the FTP

At first, the client initiates a control connection from its port *X* to port *A* (*21* by default) of the server, which is maintained during the communication. Then, the data connection is usually initiated by the server from port *B* (*20* by default) to port *Y* of the client which was signaled by the client upon establishing the control connection. This mode is also commonly referred to as the *active mode*, whereas the standardization describes this as the *active state* of the data transfer process (Postel & Reynolds, 1985).

In contrast, the client can also signal to the server by the PASV command that the client should initiate the data connection instead of the server. As a response, the server sends its Internet Protocol (IP) address and port number *B* to which the client can connect to establish the data connection. This mode is also commonly referred to as the *passive mode* (Postel & Reynolds, 1985).

For each data transfer, e.g., listing a directory or downloading a file, a new data connection is established. This introduces additional overhead, because an additional Transmission Control Protocol (TCP) connection must be initiated. After the data was transferred or the transfer was aborted, the data connection is closed again, usually by the server (Postel & Reynolds, 1985).

3.1.2 Technical Limitations

When the FTP was standardized in 1985, it provided a new way of sharing files between multiple host systems. However, due to the consistent change in technology, the FTP now contains some drawbacks that might disqualify it for modern applications. Some of these issues were addressed by making use of other protocols like Secure Shell (SSH) or Secure Sockets Layer (SSL) and thus do not provide a flexible solution (Xia et al., 2010).

The usage of two separate connections is not only problematic regarding a secure communication channel, but also with respect to network or routing issues. At the time when the FTP was standardized, more complex network setups, including (reverse) proxies, firewalls, or Network Address Translation (NAT), were not as frequently used as nowadays. These setups complicate establishing connections between the client and the server, especially in the active mode when the server initiates the data connection. This connection attempt might be blocked by a firewall that is protecting the client. In addition, the IP address sent by the server might be its internal IP in a private network, which is hidden behind a NAT. When the client tries to connect to this IP address with the given port, a connection can not be established (Gleason, 2005).

Furthermore, the number of active connections to the FTP server is often limited by the FTP server itself. This limit of concurrent connections might depend on the specific FTP server and its configurations. FTP servers often restrict access to a certain number of connections per IP address (or range) and a total maximum number of connections. For example, the popular *pure-ftpd*[1] FTP server restricts the maximum number of users to 50 and the maximum number of clients with the same IP address to 8 by default[2]. This is especially problematic when multiple connections should be used, for example, for parallel file downloads.

3.1.3 Advantages for Open Data

Although the FTP has some technical drawbacks, it is still widely used nowadays. For certain cases, the FTP still provides a suitable solution due to its stability and simplicity. Especially Open Data sources can profit from its advantages and therefore often use it to provide the data. Table 1 contains some exemplary Open Data sources which make use of the FTP.

Most importantly, Open Data sources do not require encrypted communication between the client and the server or any secure authentication mechanism due to the nature of the data. This data is supposed to be publicly available, so it should neither be restricted in access nor contain confidential data that has to be protected. Furthermore, the FTP is a straightforward solution for providing file-based data via a FTP server. This is even more relevant when the provided data is already contained in files. In contrast to other APIs that might be based on a whole technology stack (database, middleware, etc.), FTP based Open Data sources only require a comparatively simple to setup and maintain FTP server, making a dedicated directory tree accessible for clients. This can reduce the overhead of developing and maintaining an Open Data source tremendously, especially when the provider's resources are limited. Because of that, many public institutions, authorities, or software publishers still use the FTP.

[1] https://github.com/jedisct1/pure-ftpd/
[2] https://raw.githubusercontent.com/jedisct1/pure-ftpd/master/pure-ftpd.conf.in

3.2 Microservices

Over the last years, software development has become more complex due to the rapidly growing technology change, and the way software is deployed. Nowadays, cloud computing services like Amazon Web Services or Microsoft Azure provide an easy to use, flexible, and often cheap way to deploy software. Due to that, the way software is designed and developed has changed as well. (NGINX Inc., 2016)

Coming from a monolithic architectural style where all logic is bundled in a single software artifact that usually runs as a single process, the trend has changed to a finer granular, so-called microservice architecture. Martin Fowler and James Lewis describe microservices in the following way:

> "In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API." (Lewis & Fowler, 2014)

Similar to this, Newman defines microservices as "[...] small, autonomous services that work together" (Newman, 2015, p. 2). He further describes *loose coupling* and *high cohesion* as a key concept of the microservice architectural style, which states that similar functionality should be bundled into the same service, whereas communication (coupling) between services should be reduced to a minimum (Newman, 2015, p. 30). This way, various benefits like resilience, technology heterogeneity, or ease of deployment can be achieved by this architectural style (Newman, 2015, chap. 1).

Figure 3.2 shows the conceptual difference between a monolith and its corresponding architecture as microservices and was derived from figure 4 (Lewis & Fowler, 2014). As a result, communication between the single services is only possible over the network. Thus, the importance of well-designed and concise APIs is increasing, summarized by Lewis et al. as "smart endpoints and dumb pipes" (Lewis & Fowler, 2014).

3.2.1 Scaling

A fundamental difference between the monolithic and microservice-based approach is the ability to scale and how data is stored. Whereas both approaches can benefit from vertical scaling, horizontal scaling is realized differently. A monolith can only be duplicated as a whole, even though only a particular component of it would require increased system resources. This scaling can be performed more precisely with microservices, leading to more efficient utilization of the available resources, since only the service which requires additional system resources can

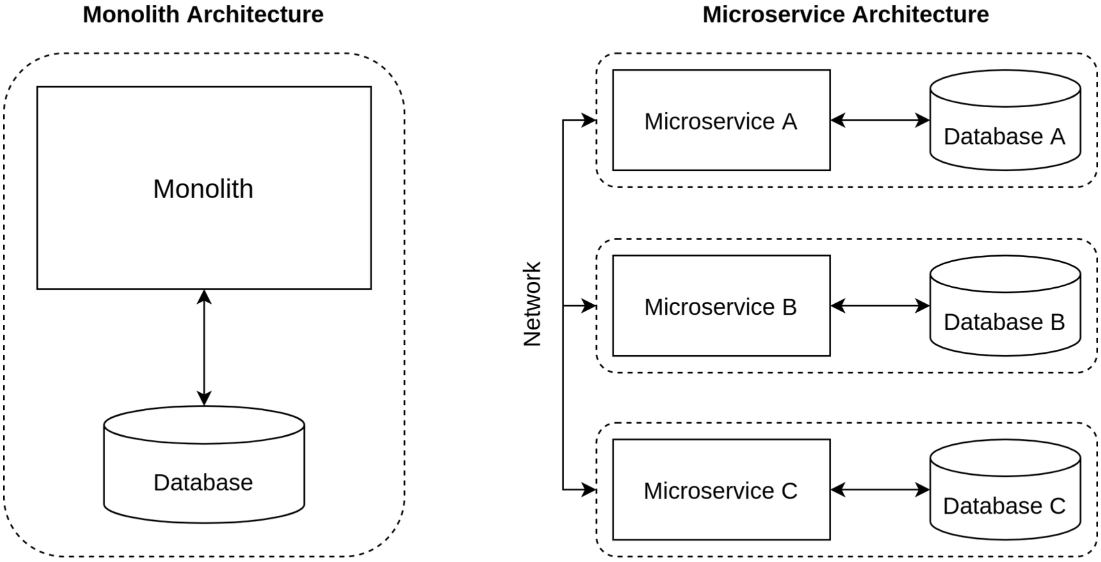

Figure 3.2: Monolith and Microservice architecture

be replicated (Lewis & Fowler, 2014).

Furthermore, microservices are different regarding persisting data. Whereas a monolith often uses a single database, each microservice is supposed to store its own data. As Newman describes, this helps to hide specific implementation details from the stable public interface and reduces coupling between the services. Ultimately, sharing databases violates the concept of *loose coupling* and *high cohesion* and complicates changing implementations of the corresponding microservices (Newman, 2015, pp. 41-42).

3.3 Architectural Styles of Application Programming Interfaces (API)

As described in the previous section, splitting monolithic architectures into a set of independent microservices shifts the communication from Inter-process communication (IPC) to the network. As a consequence, the importance of well-designed and concise APIs is increasing. When describing APIs, terms like Simple Objects Access Protocol (SOAP), Representational State Transfer (REST), Remote Procedure Call (RPC), or GraphQL are often used to specify the architectural style of the corresponding API. This section will give a short overview of the REST architectural style by comparing it to the SOAP approach, which was mainly used before the introduction of REST. Finally, the advantages of REST compared to SOAP regarding microservices are highlighted. Other architectural styles like RPC and GraphQL are omitted in this comparison.

3.3.1 Simple Object Access Protocol (SOAP)

The SOAP specification first became a World Wide Web Consortium recommendation in the year 2003 with version 1.2. In its latest specification, SOAP is described in the following way:

> "SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols." (Lafon et al., 2007)

The description already emphasizes two essential aspects of SOAP, namely the tight coupling to XML technologies and the independence of the underlying protocol. Furthermore, the SOAP itself is designed to be independent of the underlying platform or operating system, since it only relies on XML. The messages sent using a SOAP API consist of the overall envelope, a header, and a body. The body can contain an optional fault that provides additional information about errors and error handling. XML technologies are then used to reliably validate, parse and process the messages. (Lafon et al., 2007)

Messages are sent from the SOAP sender to the ultimate SOAP receiver via optional SOAP intermediaries. Those intermediaries can process the message (headers) and forward the message to the ultimate SOAP receiver, extending the original communication between a single client and server. Hirsch et al. describe a practical use case for these intermediary nodes as corporate security gateways used for encryption/authentication across corporate boundaries, which eventually increases the security of the communication between those parties (Hirsch et al., 2007, chap. 3.2.1.3).

In summary, the advantages of the SOAP are its platform and protocol independence and its standardized way of communication using XML messages. This makes the SOAP still reliable solution for many enterprise or corporate solutions, e.g., financial services. Its major disadvantages are the tight coupling to XML and the large message size due to the XML structure. Furthermore, the strict XML schema definition of the message decreases flexibility and adaption when developing SOAP based APIs (Mumbaikar, Padiya et al., 2013).

Both the lack of flexibility and the significant overhead when transferring data make SOAP an unfavorable solution for the communication between microservices which heavily depend on these characteristics.

3.3.2 Representational State Transfer (REST)

Contrary to the SOAP architectural style, REST is a more flexible architectural style that is based on the REST principles. In his dissertation from 2000, Roy

T. Fielding introduced the REST architectural style and defined the six REST principles. The following list is a short summary of the principles stated in section 5.1 of the dissertation (Fielding, 2000):

1. **Client-Server**
   The communication takes place between a client and a server, separating the user interface from the backend.

2. **Stateless**
   The communication between the client and the server must be stateless. The client is responsible for storing the session state.

3. **Caching**
   Responses from the server must be implicitly/explicitly labeled as cacheable or non-cacheable.

4. **Uniform interface**
   Implementations are decoupled from services they provide. Data is transferred in a standardized form and is not adjusted to the specific needs of an application.

5. **Layered system**
   Enabling hierarchical layers and restricting knowledge only to a single layer.

6. **Code on Demand (optional)**
   Extend the client functionality by downloading and executing code on demand.

In contrast to SOAP, these principles define constraints an API should apply to instead of a standardized protocol. Once an API applies to these constraints (to a certain degree), it is referred to as a REST or RESTful API. The REST architectural style does not require using the HTTP as the application layer protocol, but since it was designed concerning it, many RESTful APIs make use of it. Furthermore, REST does not restrict the media type of the content (JSON, XML, etc.). A fundamental concept of the REST architectural style is that endpoints provide access to resources instead of specific methods or procedures which is encapsulated in the fourth REST principle (Fielding, 2000). Due to this, REST is often described as noun-centric, whereas RPC/SOAP is mostly verb-centric.

For example, a RESTful API might provide an endpoint /users for modeling the resource *users*. This resource can be accessed or modified via the standard HTTP verbs, e.g., via GET /users for listing all users (or GET /users/{userId} for a single user) or POST /users for adding a new user. In the latter case, the actual user data would be contained in the request body. In contrast, a RPC API would instead provide multiple endpoints such as getUsers and addUser to provide this functionality.

Fielding furthermore specifies the fourth REST principle by four additional constraints that are substantial for a uniform interface (Fielding, 2000, p. 82):

- Identification of resources
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia As The Engine Of Application State (HATEOAS)

The last one, HATEOAS, states that a client using a RESTful API should not need any additional knowledge about the API itself but should be driven via hypermedia (e.g., links). This enables the client to dynamically interact with the API and rely on the relations provided by the server. In a blog post, Fielding mentioned that this constraint is often misunderstood or ignored by developers when labeling an API as RESTful (Fielding, 2008).

However, the REST architectural style provides a convenient framework when developing APIs for a microservice architecture due to the aspects mentioned above. Especially the provided flexibility, support of scaling through the layered system and statelessness make RESTful APIs a suitable choice for it, since this matches the requirements that a microservice architecture should fulfill.

4 Objectives

This chapter lists the objectives that were established for this thesis. The objectives are referenced using the combined section and listing number, e.g. *4.1.1* for the first objective of section 4.1.

4.1 Exploration of FTP Data Sources

1. The modelling of data sources shall generalize the structure of hierarchical Open Data sources while providing a mechanism to annotate data source nodes with specific properties without loss of generality in order to create a universal abstraction that can easily be extended for specific types of data sources.
2. The software shall include a mechanism to create an intuitive configuration that stores information about relevant data source nodes and their specification (update intervals, request parameters, etc.) in order to use the results from the exploration process for the existing ODS pipeline infrastructure.
3. The software shall provide a RESTful API that enables third-party applications to use the functionality provided by the Hierarchical Datasource Service (HDS).
4. The software shall fulfill the above-mentioned objectives for FTP data sources, fit into the existing microservice environment, and apply to common programming and documentation guidelines in order to simplify expanding and collaborative work.

4.2 Support of Archive Inspection

1. The software shall support the extraction of .zip archives on the server-side. The archives shall be extracted on the server, so the client does not have to install additional software or download the archives.

2. The content of the extracted directory shall be handled as the content of a regular directory that is directly accessible via the FTP data source. It shall be possible to *download* files that are located in an archive and *export* them periodically later on via the ODS pipeline.

4.3 Intuitive User Interface

1. The user interface shall be web-based, responsive, and focus on the design of desktop devices in order to provide the best user experience for the common use cases.

2. The user interface shall use VueJS as the JavaScript framework and Bootstrap as a styling framework and apply to common programming and documentation guidelines in order to achieve code maintainability and expandability.

3. The user interface shall mirror the sequential workflow of adding a data source, exploring it, and selecting relevant nodes for the export to corresponding pages/screens with back and forth navigation in order to be self-explanatory and intuitive to use.

4. The user interface shall allow the user to explore the data source in a file browser-like manner with the opportunity to show additional information and view the content of a data source node on demand in order to provide an easy and revealing exploration of the data source.

5. The user interface shall support the selection of multiple files and directories (recursively) for the export into the ODS pipeline in order to be practicable for applications that require the data provided by multiple data source nodes.

5 Solution Design

5.1 Model of Hierarchical Data Sources

The fundamental concept of hierarchical data sources is their inherent hierarchy. This hierarchy can be interpreted as an arbitrary tree structure, which enables structured traversal and exploration of the data source. Thus, the data source can be abstracted using basic graph theory in which a graph $G = (V, E)$ is defined as a tuple of nodes and edges. Using this abstraction, FTP data sources publish a file system in which the files and directories are nodes of the graph and edges between nodes define the hierarchical structure. Furthermore, files are always leaf nodes, whereas directories always will have children nodes and thus are further traversable unless they are empty. It is important to note that this approach is not restricted to FTP data sources or file systems in general. A HTTP data source which follows the RESTful design approach models the hierarchy of described entities by their Uniform Resource Locator (URL). For example, a RESTful data source provides the list of users at /users and information about a specific user *John* at /users/john. Similar to FTP data sources, this hierarchy can be abstracted using an arbitrary tree structure.

In contrast to the raw graph theory in which the nodes are often unique identifiers such as v_i, the nodes of hierarchical data sources have additional data tied to them. This data depends on the actual type of the data source and the node itself. An endpoint of a RESTful API, for example, has a dedicated HTTP method tied to it (GET, POST, etc.), whereas a file has a certain size. On the other hand, both nodes are identified within the data source by their URL/path and are also leaf or no leaf nodes within the data source.

5.1.1 Definition of a Data Source Node

As described above, the fundamental concept of modelling hierarchical data sources is the definition of their components (nodes) and the inherent hierarchy, which is introduced by their relation to each other (edges). Due to the hierarchical structure of the data source, each node has a parent node and a set of children nodes. Exceptions are the root node for which the parent is undefined and leaf nodes with an empty set of children nodes. Furthermore, each node has a name and is uniquely identifiable by a URL.

At this point, these properties would allow modelling the hierarchical data source sufficiently for traversal, but without any additional functionality such as displaying specific properties or downloading a file from a FTP data source. In order to achieve this, each node also has a set of properties and a set of actions that can be performed on it as shown in figure 5.1.

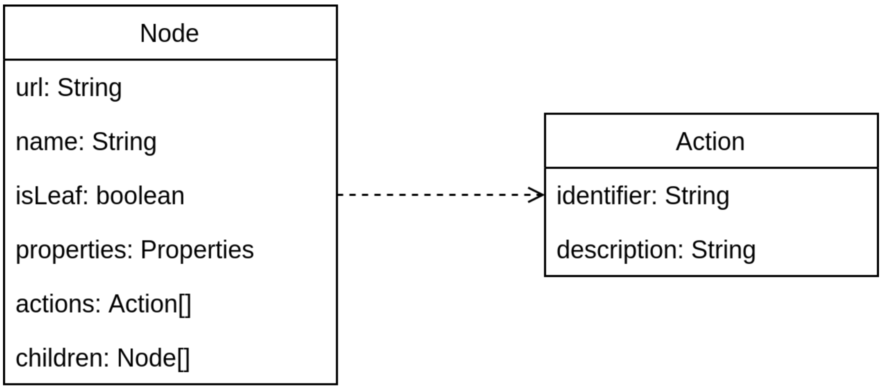

Figure 5.1: Definition of a data source node

The isLeaf attribute indicates if the node is a leaf node and was added for simplicity. It is important to note that no dedicated parent attribute for bidirectional tree traversal exists. This is due to the fact that traversals start at the root node and thus, the parent node was already known before accessing the node itself. In addition, the URL of the parent node can be retrieved from the URL of the node.

The properties store additional information about the node itself, whereas actions are basically plain objects that define an action for a node. These actions are implemented separately, and each node stores the identifiers of the actions that are applicable for it. This design decision is explained in the upcoming section.

5.1.2 Extendability for Other Types of Data Sources

A significant issue of this conceptual modelling is that it should cover as many use cases as possible. Thus, it must be general enough not to restrict specific use cases and adaptable enough to fit as many different situations as possible. The previous section described the general attributes of every hierarchical structure and its specific properties and actions. Both of these attributes are used to store additional information dependent on the type of the data source. For example, a file of a FTP data source can have a *download* action that downloads this specific file from the data source, whereas a HTTP endpoint of a RESTful API can have a *request* action which sends a request to the specific endpoint. The same applies to the properties attribute, which can store the file size of a file or the HTTP request method for an RESTful endpoint. With this concept, new types of data sources can be supported by adding the properties and implementing the applicable actions for this type of data source. All conceptual functionality, such as data source exploration and accessing individual nodes, is independent of this and is not required to be modified.

5.1.3 FTP Specific Properties

This section will describe the FTP-specific properties due to the FTP-focused scope of this thesis (see objective 4.1.4). The nodes of a FTP data source are similar to those of a hierarchical file system, namely files and directories. In addition, there are also symbolic links that can reference other files or directories. A special case of a regular file within this thesis is an archive that must be handled separately. Compressed files, such as .gz files that are compressed using *gzip*, are also viewed as an archive since the actual file content is not directly accessible. These different kinds of nodes also have different properties as shown in figure 5.2. Regular files (including archives) have a file size, a modification time, and an extension, whereas symbolic links have a destination they link to. Directories do not have additional properties.

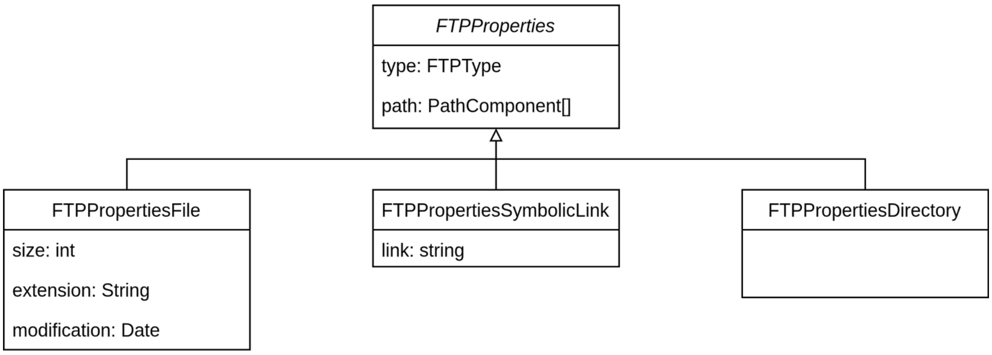

Figure 5.2: Definition of the FTP specific properties (1)

Additional to the url attribute of the node itself, the FTP properties store an extra path attribute that is a list of path components of the URL (figure 5.3). For regular files and directories, this path consists of simply one entry for which the path is equal to the URL and the type to the node type. However, a problem arises when a file is hidden inside of an archive. In this case, a single URL is not sufficient to encode that this file is not directly accessible at the data source, and further steps must be taken in order to access this file (see section 6.1.5).

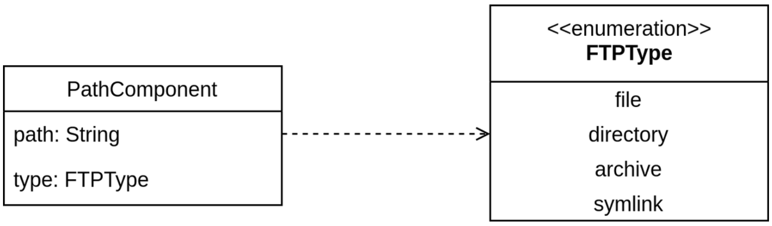

Figure 5.3: Definition of the FTP specific properties (2)

An example of a file in an archive is given in figure 5.4 below.

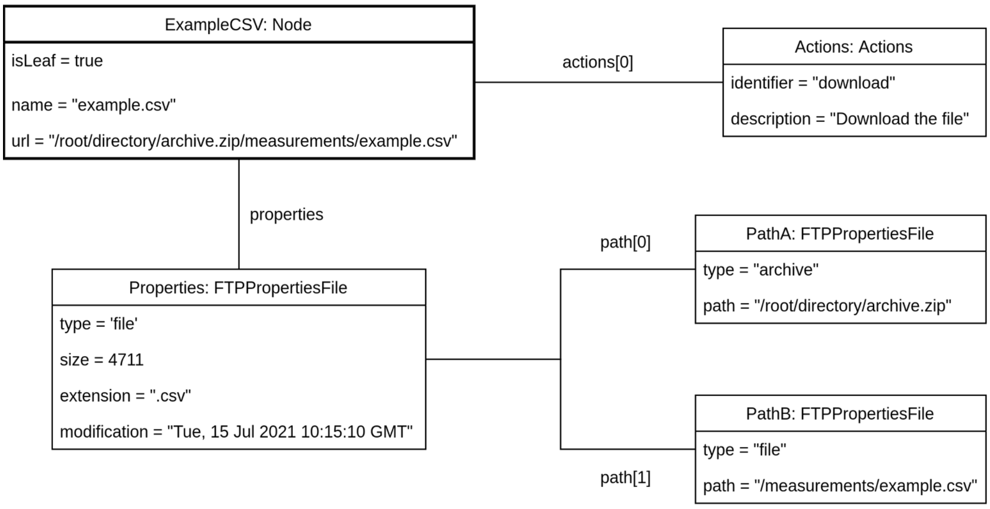

Figure 5.4: Example of a file inside an archive

5.2 Inspection of Archives

A major problem this thesis deals with is simplifying the process to retrieve data from (compressed) archives. Archives are a practical way to group files and provide all the contained files in a single download compared to regular directories, where each file has to be downloaded individually. With compression, the actual size of the archive can be crucially reduced compared to providing the raw files. Compression algorithms primarily perform well for similar data like measurement data, which is especially useful for many Open Data when this kind of data is provided. Thus, the usage of archives reduces network traffic and the required disk space.

Unfortunately, these advantages come with additional drawbacks. First of all, the structure of the archive is not remotely viewable anymore. Similar to common file browsers, the FTP does not provide the functionality to inspect (compressed) archives remotely. While decompressing an archive on the local file system is relatively fast, a remote archive first has to be downloaded, which might take some time depending on the size of its content and the network connection. This is especially laborious when the user wants to browse through the content of the archive quickly and is only interested in specific files based on their name. Furthermore, it is an additional overhead for the user to download and extract the archive, in particular when extra software is required to extract the archive. The user might also use a mobile device, which does not necessarily provide software to, for example, extract a *gzipped* .tar archive.

In most cases, an archive stores a set of files or a directory tree with a recursive directory structure. Unfortunately, data sources like opendata.dwd.de also expose archives that contain archives themselves or store compressed files. Compared to the example in figure 5.4, the path attribute of a file in a recursively structured archive would consist of multiple path components of the type *archive* instead of a single one. In this case, extracting the root archive does not enable the user to inspect the files of interest. Instead, all other archives must be extracted recursively in order to gain access to the contained files.

5.3 Caching and Scalability

Section 3.2 gave a short overview of how the microservice architecture can be beneficial for tailored horizontal scaling of services. Whereas different services should be decoupled from each other and should not use the same data, instances of the same service might benefit from shared access or caching of the data for increased performance.

In the context of the HDS, the term *cached data* is interchangeably used with extracted archives. Since archives can not be inspected on the remote data sources, they have to be downloaded, extracted (locally), and stored. The extracted data is not required to be persisted permanently (unlike user credentials, for example) since it is only used for performance improvements. Instead of downloading and extracting the same archive each time it is requested to be inspected, performance can be increased by only doing this once and storing the extracted archive for further requests. Once it changes, the cached entry can be replaced (see section 6.3.3). Sharing this extracted archive between multiple HDS instances can lead to further performance improvements since additional downloading/extracting overhead is skipped.

There are various solutions for sharing the content of extracted archives between multiple HDS instances. First of all, it would be imaginable that - after an archive was extracted - its content is inserted into a database which is shared across HDS instances, as shown in figure 5.5. The dotted lines indicate the boundaries of the host a HDS instance is running on.

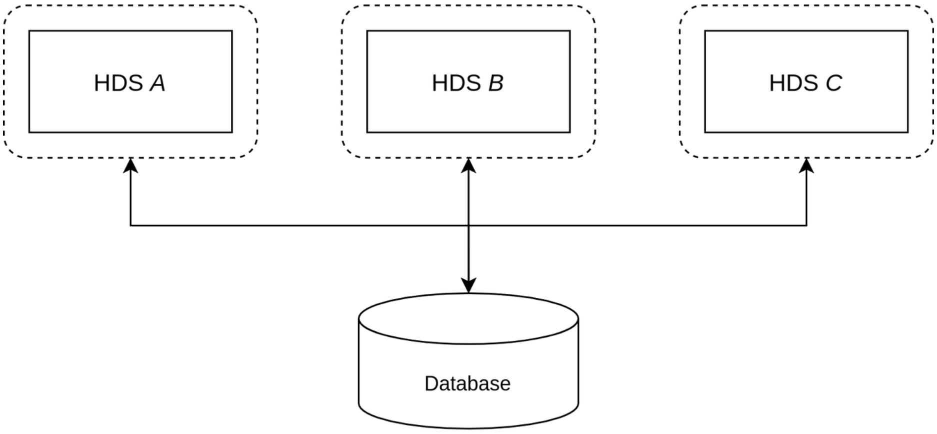

Figure 5.5: Database as the shared cache

This way, a HDS instance would query the database before downloading and extracting the archive again, potentially profiting from the cached entries. The major downside of this approach is the increased network traffic, which is introduced by transferring large amounts of data between the database and the HDS instances. On the other hand, HDS instances could share this data between host boundaries.

Instead of using an external database, another approach would be to make use of the local file system as the shared cache as shown in figure 5.6.

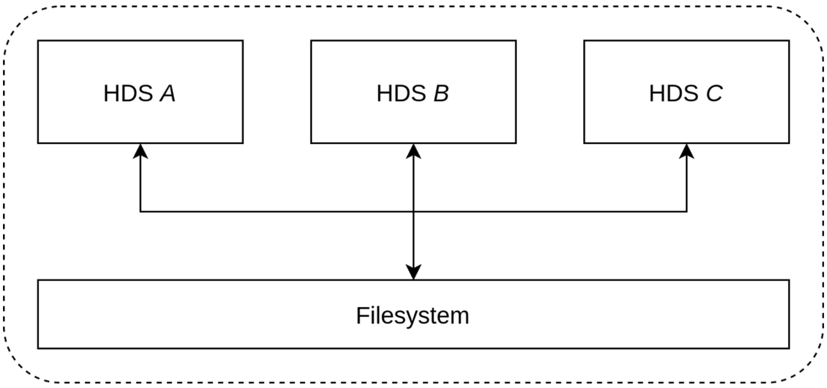

Figure 5.6: File system as the shared cache

First of all, this is a solution that removes the complexity of an additional database. Secondly, there is no additional network traffic introduced in order to persist the data after extraction. The disadvantage is that HDS instances can only share cached data when they have access to the same local file system. Theoretically, this can be bypassed by using a network file system, but this would again introduce high network traffic when persisting the data.

For both solutions, two problems still exist:

- even though cached data can be added, there is no mechanism to remove already cached data in order to free space
- cached data can be modified concurrently, leading to unspecified results

The mechanism to remove already cached data could be either integrated to the HDS itself or to another service that solely keeps track of the cache and removes unused entries by a predefined cache policy (e.g., *last-frequently-used, last-recently used, etc.*). Restricting the concurrent access can be implemented by either using already existing locking mechanisms like table/row locks (database) or lockfiles (file system).

While both approaches come with their advantages and disadvantages, both can be used to allow multiple instances to use the same cache leading to reduced network traffic and increased performance. In the scope of this thesis, the caching was realized using a shared file system, whereas the mechanism of removing cached entries was ignored due to its low priority. The implementation of the file system as the shared cache is described in detail in section 6.3.

5.4 Compatibility with the Open Data Service (ODS)

The HDS aims to add support of FTP data Sources to the ODS. In the scope of this thesis, the HDS is implemented as a standalone microservice that is entirely independent of the ODS. The same applies to the proof-of-concept user interface, which was implemented to demonstrate the functionality of the HDS for the user.

The ODS currently consists of several microservices as shown in figure 5.7. Here is a short overview about the purpose of each component:

- *Datasource* - Fetch the data from the data sources
- *Pipeline* - Transform the fetched data according to ETL
- *Query* - Persist the data and make it accessible
- *Notification* - Send notifications on events
- *Scheduler* - Orchestrate tasks and schedule pipeline executions
- *Web-Client* - User interface for creating pipelines and data sources

Regarding the HDS, the *Datasource* service is of special interest since it is responsible for (periodically) fetching the data from the external Open Data source. Contrary to the HDS, the *Datasource* service does not provide any functionality to explore the data source itself and create a configuration *based on* this exploration but simply expects a single URL, which must be given by the user. Such

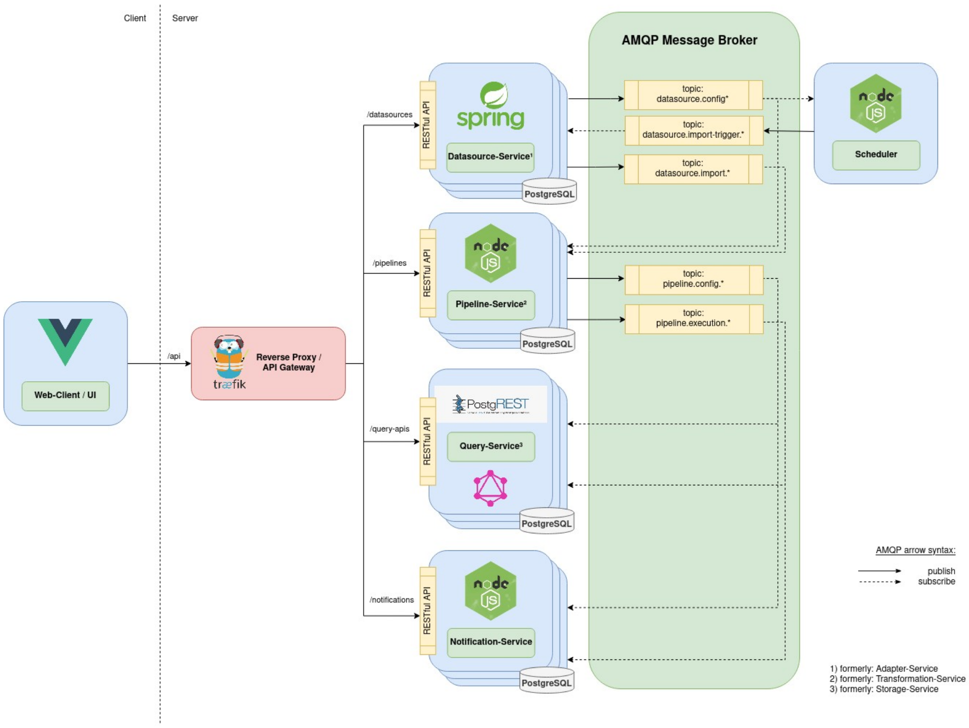

Figure 5.7: Architecture of the ODS.
Reference:
https://github.com/jvalue/open-data-service/blob/main/doc/service_arch.png

a configuration contains all required information about the data source and is essential for both services. The HDS defines such a configuration with some modifications compared to the *Datasource* service, as shown in figure 5.8.

First of all, the metadata and id properties are omitted due to simplicity, whereas the trigger property is the same for both configurations. The primary differences are the protocol/format and connection/entries properties, which specify how to connect to the data source and which data should be fetched. In contrast to the protocol property, the connection property stores the URL of the FTP server, its port and the user/password. The entries property extends the format property by the support of configuring multiple file paths via regular expressions instead of just a single URL. This is due to the fact that the path of files is often not known beforehand due to components in the filename or directory structure that are due to change over time, e.g., dates, indices, and so on. As a result, a single item of the entries list can match an arbitrary number of files which ideally should apply to the same schema. It is further described in section 6.2 how these regular expressions are generated and resolved.

Another difference is that the HDS itself does not store any data source configurations or triggers the execution to fetch new data, since the focus of this thesis

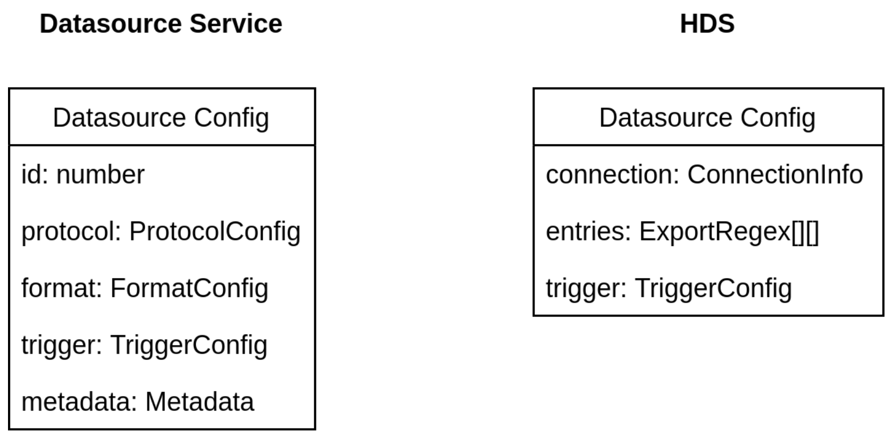

Figure 5.8: Data source configuration of the *Datasource* service and the HDS

is the accessibility of FTP data sources. Besides that, the HDS would mostly fit into the ODS ecosystem. Chapter 9 gives a short outlook about how this integration can be accomplished.

5.4.1 Adaption to the Pipeline Mechanism

Once a data source configuration is created, it can be used to configure a pipeline. A pipeline represents the process of fetching the data that is specified by a particular data source configuration, applying an optional data transformation, and persisting the result through the *Query* service. The various services are notified via the *Message Broker* once new data is available.

The core concept of the HDS works in a similar manner, but it provides some additional challenges. As mentioned in the previous chapter, the data source configuration of the HDS can contain *multiple* files which should be fetched upon its execution due to its configuration via regular expressions. As a result, the content processed in a pipeline is no longer a single JSON/CSV/XML resource. This introduces a problem when the retrieved data should be queried via the *Query* service. In the example of the repository, the latest entry of such a pipeline is retrievable via the link

$$\text{http://}\underbrace{\text{localhost:9000/storage}}_{\text{URL of } \textit{Query} \text{ service}}/\underbrace{2}_{\text{Pipeline ID}}?\underbrace{\text{order=id.desc\&limit=1}}_{\text{Query parameter}}$$

which will return the JSON/CSV/XML content. In order to illustrate the new challenges, a short example is introduced. The assumption is a configuration with three ExportRegex entries, which resolve to nine files that the HDS/ODS should export. When retrieving the data via the *Query* service, the user should be able to distinguish between the single files. Therefore, it is required to specify the corresponding ExportRegex entry and the index of the file of interest within this entryA possible solution could extend the existing link structure with two additional query parameters, e.g.:

$$\underbrace{\text{entry=2}}_{\text{Index of the ExportRegex entry}} \quad \& \quad \underbrace{\text{index=1}}_{\text{Index of the file within this entry}}$$

Accessing a file that does not exist via invalid entry and index values could simply result in an error response by the *Query* service. Otherwise, the file content of the corresponding file will be returned.

6 Implementation

6.1 The Hierarchical Open Data Service (HDS)

The HDS is implemented as a microservice in Typescript[1] and available on GitHub[2]. Its coarse structure is shown in figure 6.1.

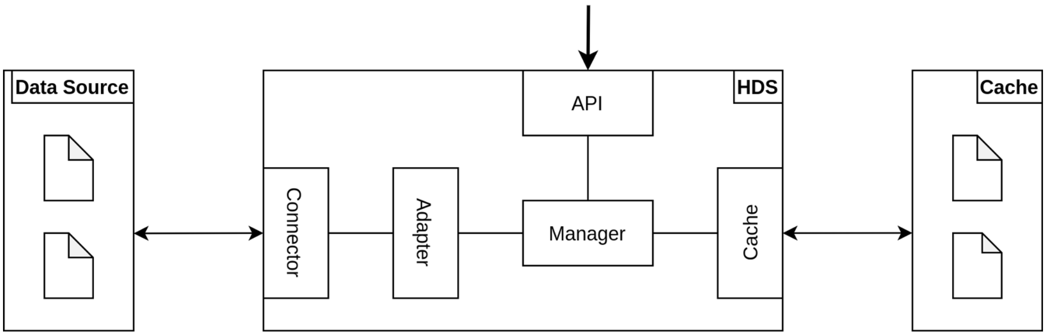

Figure 6.1: Architecture of the HDS

The HDS is accessible via its RESTful API that exposes the functionality of the HDS and forwards incoming requests to the corresponding functions of the *Manager* module, which is the central component of the HDS. It implements the business logic and uses the *Cache* and *Adapter* in order to access the remote data source or locally stored data respectively. The *Adapter* abstracts the access to remote data sources and relies on a corresponding *Connector* which establishes the actual connections.

In the following sections, the word *node* will be used as a placeholder for FTP specific nodes such as files, directories, archives and symbolic links. Finally, here is a short overview about the most important software packages being used by the HDS:

[1] https://www.typescriptlang.org/
[2] https://github.com/jvalue/hierarchical-datasources

- **basic-ftp** (MIT) - FTP client
- **decompress** (MIT) - Extracting (compressed) archives
- **proper-lockfile** (MIT) - File locking utility

6.1.1 Application Programming Interface

The HDS provides a RESTful API for stateless communication between the client and the HDS service. The API neither provides an authentication mechanism for restricted access nor encryption for secure communication. The content type for all API endpoints is JSON. According to the REST principles, the API aims to model data sources and their content as entities. The GitHub repository also contains an OpenAPI v3 specification[3] of the API.

The *data source* entity provides two endpoints for listing and adding FTP data sources.

- GET /datasources
  Get allimported data sources
  The response body contains a list of ConnectionInfo items (see figure 6.2).
- POST /datasources
  Add a new data source
  The request body contains the connection information (see figure 6.2).

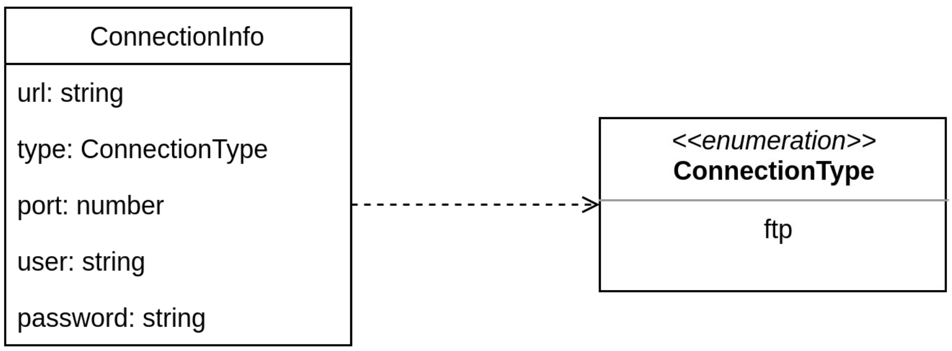

Figure 6.2: Definition of a data source connection

The *node* entity provides a single GET endpoint to retrieve the content of the node and a single POST endpoint to perform actions on this node. It is important to note that the node URL is sufficient for the HDS to reconstruct the path components of the path attribute. Therefore, the URL is split upon the known archive borders (see section 6.1.5).

[3]https://github.com/jvalue/hierarchical-datasources/blob/main/backend/static/swagger.yml

- GET /datasources/{dsUrl}/{nodeUrl}
  Get the content of the node
  The placeholder {dsUrl} and {nodeUrl} contain the URI encoded URL of the data source/node. This endpoint returns the content of the node (see figure 5.1), independent of the type of the node.

- POST /datasources/{dsUrl}/{nodeUrl}
  Perform an action on the node
  The placeholder {dsUrl} and {nodeUrl} contain the URI encoded URL of the data source/node. The request body contains a JSON object with an *identifier* property (string) that specifies the action to perform. The response content is dependent on the action.

Supported actions are download and extract to download a single file or extract an archive. The former transfers the file content as a *blob*, the response body of the latter is empty. Following a symbolic link or inspecting an extracted archive can be reduced to the GET endpoint by requesting it with the link destination or the URL of the archive. This implementation violates the design principles of REST since these actions are not modelled with the Create Read Update Delete (CRUD) operations. Instead, this implementation applies to an RPC based design. Nevertheless, this approach was chosen due to its simplicity and extendability.

Section D of the appendix shows some exemplary API calls with the content of the opendata.dwd.de data source. The usual procedure is to first add the data source to explore (example D.1), then request the content of the root node (example D.3) and navigate through the data source until, for example, a file was found that should be downloaded (example D.4). Furthermore, the API provides a single POST endpoint for exporting a data source configuration (see section 6.2).

- POST /export
  Get the matching nodes of an export configuration
  The request body contains the export configuration (see figure 6.4). The response body contains a list of Node objects (see figure 5.1), which are the matching nodes of the export configuration (example D.5).

Finally, the API provides another *events* entity that is mainly used for development and testing purposes. These endpoints are disabled when the HDS is running in production mode and their usage is further described in section 8.3.

- GET /events
  Get the registered events
  The response body contains the list of registered events.

- DELETE /events
  Delete all registered events

Deletes all registered events.

The following events are recorded:

- *archiveLockIsHeld* - The archive is currently locked
- *archiveLocked* - The archive lock was acquired
- *archiveReleased* - The archive lock was released
- *archiveUpdated* - The cached archive was updated
- *archiveCached* - The archive was cached and is still up to date
- *archiveExtracted* - The archive was extracted

6.1.2 Connection to Data Sources

The first step that has to be taken when exploring a data source is establishing a connection to the data source. Therefore, several parameters must be known:

- the protocol that is used to access the data source
- the URL and port at which the data source is accessible
- the username and password for accessing the data source

The HDS only supports FTP data sources (see figure 6.2) which require a username/password combination. In general, authentication can also make use of another mechanism like an authentication token, which is often the case for RESTful APIs. The HDS provides an endpoint POST /datasources (see 6.1.1) that expects a JSON object containing this information in the request body. Based on this connection data, a connection to the data source is established. If the connection to the data source fails, e.g., due to an unreachable URL or invalid port number, the error is returned to the client. On success, an internal mapping stores the connection objects for this data source. An exemplary API request is shown in example D.1.

By default, up to five connections are established to each data source for parallel access. The number of available connections could be even increased by sequentially opening new connections until the first one is rejected by the server. All these connections are established in the *passive mode* by default. While the HDS is running, these connections are kept alive, which means that once the connection is closed by the server, they are automatically reconnected the next time this connection object is used. In the case of the FTP, this might happen after a certain timeout defined by the FTP server itself. Different clients of the HDS share the same connections for the same data source. This is because many FTP servers restrict the number of connections for specific IP ranges. Since the data sources are public, this approach was chosen to prevent constantly establishing and closing client-specific connections, which would reduce performance.

6.1.3 Exploration of Data Sources

The exploration of the data source is implemented by the single API endpoint GET /datasources/{dsUrl}/{nodeUrl} (see section 6.1.1). Sending a request to this endpoint will return the content of the corresponding node in JSON format. Once a connection to the data source is established, this endpoint can be used to explore the data source in a structured way. Without previous knowledge, the first request starts at the root node "/" (encoded %2F) of the data source. The response contains the properties of the root node itself and the list of its children nodes. The same API endpoint can then be requested again with one of the now known child URLs, which has the isLeaf property set to false. Requesting a leafnode again will simply return the already known content. This procedure can be repeated in order to fully explore the whole data source.

In order to navigate back from a child node to its parent node, either the URL of the parent node must be stored or its URL is retrieved by the *dirname* of the URL of the child node. Of course, the application can also maintain a stack of parent nodes. It is important to note that symbolic links can introduce cycles in this exploration when they are followed (see section 6.1.4). The data source can also be explored fully automatically without any user interaction, for example, when searching for a specific file. Therefore, this search would also start at the root node - assuming there is no previous knowledge about specific URLs - and use the API endpoint to explore the data source using a depth-first or breadth-first search. The search can also follow symbolic links when a set of already visited nodes is maintained.

6.1.4 Symbolic Links

Symbolic links provide a useful way to have a static reference to files or directories that are due to change. For example, the latest measurement file of a series of .csv files could be referenced by a symbolic link with the name latest_measurement. This link then can be used to access the latest measurement data without actually knowing the name of the file which contains this data. In this scenario, the symbolic link acts like an intermediate node for accessing the referenced node.

Since symbolic links are files themselves, it is a special node with an additional link property which distincts them from regular files. Consequently, there is a difference between the content of the link it points to and the node itself. In order to retrieve the content of the linked node, the nodeUrl parameter of the API endpoints (see section 6.1.1) must be replaced with the value of the link property. Symbolic links are also restored when an archive is extracted. The link property is joined with the *dirname* of the symbolic link when the destination is a relative path to retrieve the correct destination. If the destination is an absolute path, the link property will be simply set to this path.

6.1.5 Archive Extraction

As described in section 5.2, the inspection of archives is a central problem this thesis is trying to solve. While archives provide advantages, such as reduced file size for the data source provider, they add additional overhead for the user when trying to access the contained data.

Archives themselves are regular files and can be downloaded from the data source using the POST /datasources/{dsUrl}/{nodeUrl} endpoint (see 6.1.1). This will download the archive as it is from the data source, providing no additional benefit to the user who still has to store, extract and inspect the archive locally. Instead, the same endpoint can be requested with the extract action which will download the archive from the data source to the cache of the HDS and extract it. The extraction of the archive is performed recursively, because an archive can contain multiple other archives or compressed files. This procedure simplifies the process when accessing a node from the cache since it can be safely assumed that each node is already accessible. The HDS supports the extraction of the following archive types:

- .zip
- .tar.gz / .tgz
- .tar.bz2
- .tar

After extracting the archive, the modification timestamp of the extracted root directory is updated with the original modification timestamp from the data source. This is because the downloaded archive has its modification timestamp set to the point of time when the download process started. The modification timestamp is essential to find out if an archive was updated on the data source. This process is described in detail in section 6.3.3.

Besides archives, there are also often solely compressed files. A very popular compression tool is *gzip*[4] which is developed for the GNU project. In contrast to archives which can store multiple files and directories, *gzip* will only compress single files. In order to treat both archives and solely compressed files similarly, decompressing with *gzip* compressed files has to be adapted. First, the compressed file is downloaded and extracted to a temporary file. Afterwards, a

[4] https://www.gnu.org/software/gzip/

directory with the name of the compressed file is created and the temporary file is moved into this directory. In the last step, the extracted file is renamed to the original filename without the compressed .gz extension and the modification timestamp of the directory is updated similar to regular archives. This way, solely compressed files are handled the same way as regular archives, reducing complexity and edge cases in the implementation. The HDS only supports this procedure for with *gzip* compressed files.

6.2 Export Configuration of the ODS Pipeline

The final result of the exploration of the data source is a set of files selected by the user that contain the required data for the third-party application. The goal is to periodically retrieve those files by the ODS and provide them to the third-party application. The configuration which specifies this result set should be only created once and should contain all relevant information for connecting to the data source and retrieving the desired files.

As described in section 3.2, the concept of microservices is based on a clear separation of responsibility and functionality. Thus, the ODS should be fully independent of any FTP related functionality of the HDS. Since the configuration must be stored by the ODS anyway, this should be all the information required in order to retrieve the specified files. As a result, the ODS can send this configuration to the HDS and download each file of the returned result set from the HDS using its RESTful API. This way, the HDS acts as a proxy for retrieving the files from the FTP server such that the ODS can make use of its existing HTTP functionality. In the upcoming sections, it will be described first how regular expressions are used to enable dynamically resolving the result set of the configuration, and afterwards how the configuration is structured and the final result set is determined.

6.2.1 Support of Regular Expressions

During the exploration process, the user selects the files of interest which are statically determined by their URL. Unfortunately, these paths often contain components that are due to change. The opendata.dwd.de data source, for example, provides .zip archives like {...}_20210322_{...}.zip with the corresponding date (3rd March 2021) in the filename. Consequently, a dashboard application that uses the daily measurement data relies on a way to dynamically determine the location of the latest measurement instead of using statically specified paths.

A common approach to solve this kind of problem is the usage of regular expressions. Regular expressions provide a powerful mechanism to statically specify a pattern for strings which then is dynamically applied at runtime to see if the string matches this pattern. In this context, the strings are the URLs of the desired files of the FTP data source. Whereas a static file path is unique, regular expressions can be matched arbitrary times, creating a result set. This result set should be further adjustable in order to select certain important files instead of a whole collection of data. In general, the result set should be

- sortable by applying a predefined order, converting the set into a list
- sliceable by selecting a range of the original list with an offset and size

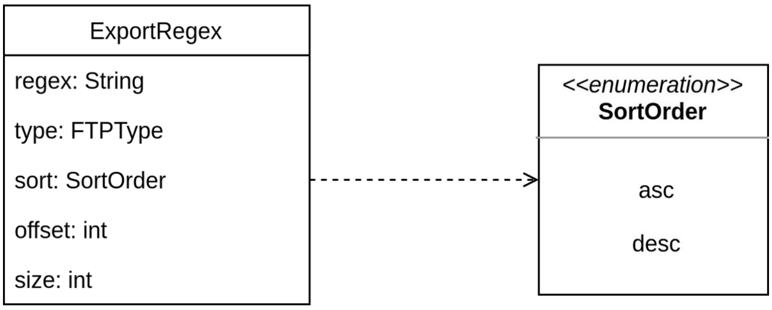

Figure 6.3: Definition of a regular expression for the export configuration

The regular expression regex is defined according to the JavaScript format[5]. The regex must be specified as a single string without the leading and ending slash and is applied **case-insensitive**. For example, the regular expression data/(.*).csv would match all .csv files in the data directory.

As shown in figure 6.3, the ExportRegex also contains a type attribute that specifies for which type of nodes the regular expression should match, e.g., only files by setting its value to file. Supported values of the sort order are asc or desc for alphanumeric ascending or descending order. The offset and size parameters must be positive integers including zero. When the size property is set to zero, all matches are taken into account.

6.2.2 Structure of a Configuration File

The previous section described the structure of a single regular expression. These are the fundamental components of the overall export configuration that can be sent to the HDS to dynamically determine the list of matching files. Therefore, the configuration must contain the following data:

- a trigger configuration
- the connection data of the data source (see figure 6.2)

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

- a list of regular expressions that specify the files of interest

The final goal is that the ODS periodically retrieves the files from the HDS and persists those files for the third-party application. Because of that, the trigger configuration specifies *when* the ODS is supposed to download those files. This is due to the adaption to the current implementation of the ODS, as described in section 5.4. Hence, the interval parameter specifies the periodic interval in seconds (if periodic is true) and the firstExecution parameter defines the date and time of the first execution. The final structure of the configuration is shown in figure 6.4.

Figure 6.4: Definition of the export configuration

It is noticeable that the entries parameter is a list of ExportRegex lists. This is due to the fact that the *overall* regular expression must be split upon the archive borders, since the content of archives is not directly accessible. Thus, it is only required to have multiple ExportRegex items defined when the desired files are located in at least one archive. The type attribute of the last ExportRegex must always be set to file. For the sake of simplicity, the term ExportEntry will be used interchangeably for a list of ExportRegex items. The next section will describe how the list of matching files is determined based on such an export configuration.

6.2.3 Resolving a Configuration

Once a request is sent to the POST /export endpoint, it is checked if the request body contains a valid export configuration. If this is the case, a connection to the data source is established if there is not already an existing connection. At this point, the setup is completed and the list of ExportEntry items is iterated. For each ExportEntry, the following algorithm is executed:

1. Initialize the result list for the *i* -th ExportEntry as the empty list
2. Iterate over the corresponding ExportRegex items and determine the matching files
   (a) Split the regex with the path separator "/" and apply the regular expression component by component

   - The next component is only applied relative to the matches of the previous one

(b) The result of the previous step is a set of URLs that match the regex of the ExportRegex

   i. Sort the result set using the sort order specified by sort

   ii. Slice the result list using the size and offset parameters

(c) The result contains the URLs with which the next ExportRegex item starts

   - If the type is set to archive, the result contains archives which must be extracted before continuing

(d) Apply steps (a) - (c) until the last ExportRegex is resolved and add each matching file to the result list

The final result is a list of Node lists, similar to the list of ExportRegex lists of the configuration. For example, the fourth Node list of the result contains all files that matched the fourth ExportEntry item of the export configuration. Since all these nodes are downloadable files, the ODS can simply request the corresponding API endpoint for each node URL in order to download and store the file.

To demonstrate this procedure, listing C.1 contains an exemplary export configuration for which the trigger property is dismissed due to simplicity. Therefore, a single ExportEntry will be used with two ExportRegex entries. The specified FTP data source is the custom FTP server that is also used for the automated testing described in section 8.3. Its content is displayed in figure 15. Finally, figure 16 shows in detail how the configuration is evaluated using the above-mentioned procedure. For easier understanding, the intermediate steps with the matching nodes are listed as well.

6.3 File System as the Distributed Cache

6.3.1 Hierarchical Structure

The file system cache is, similar to the data sources themselves, hierarchically structured. The root directory of the cache is defined by the environment variable HDS_CACHE. For each connected data source, a subdirectory named by the data source URL is created in the cache root directory. This directory is referenced as the data source root directory. Relative to this directory, the URL of a node is equal to its path on the file system. As a consequence, no additional mapping or state keeping is required by the cache about the location of single nodes. This makes the solution flexible, error-prone, and fulfills the imposed state restrictions for the HDS.

When an archive is requested to be extracted, the archive must be downloaded first. In this example, it is assumed that the path of the archive within the data source is /directory/archive.zip. The destination within the cache is the joined path of the cache root directory, the data source root directory and the path of the archive, e.g., ${HDS_CACHE}/{dsUrl}/directory/archive.zip. Each intermediate directory, which does not already exist, is created. After extracting the archive, the URL of a node of that extracted archive can be simply computed from its path on the file system by cutting off the cache prefix. This cache prefix consists of the cache root directory and the data source root directory.

6.3.2 Concurrent Access

As described in section 5.3, it is beneficial for multiple instances of the HDS to share the same cache in order to improve performance, reduce network traffic and disk space. Unfortunately, these advantages introduce additional overhead since various HDS instances can now access and modify the same files. This can result in race conditions or lost update problems between two or more concurrent HDS instances, when for example two HDS instances are downloading the same archive at the same time in order to extract it afterwards. Due to the implementation as a microservice and the horizontal scaling, HDS instances do not share communication data that would enable the services to mutually lock the access to a certain file.

Therefore, a solution is required that is solely based on the already shared file system cache. Similar to other applications that concurrently access the same file system, the mechanism of lockfiles is used. These lockfiles indicate that a file is currently locked by another application and should not be modified. Usually, these lockfiles are located in the same directory as the file they lock with a suffix indicating that this is a dedicated lockfile (e.g. file.lock). This practice is infeasible for the HDS since it is not guaranteed that there is no such file provided by the data source itself. Thus, these lockfiles must be separated from the actual data of the data source, thus outside of the data source root directories. The path of the lockfile must be fully determined by the URL of the data source and the URL of the file it should lock.

For this, a special directory .locks in the cache root directory is created that will be referenced as the cache locks directory. This directory solely stores the lockfiles in the same hierarchical structure. For example, the lockfile of the archive /directory/archive.zip of the data source datasource.de will be located at ${HDS_CACHE}/.locks/datasource.de/directory/archive.zip. In case of recursive archives, only the root archive must be locked since the extraction is performed recursively after the lock had been acquired. The library *proper-lockfile*[6] is used for the lockfile mechanism. The implemented locking mechanism is based on the modification timestamp of the corresponding file and thus also works to restrict concurrent access within the same process. It was tested (see section 8.3.1) on a *ext4* file system but also supports network-based file systems according to its documentation.

6.3.3 Updating Cached Archives

The previous section described how concurrent access for the same cached file is synchronized using the mechanism of lockfiles. This will become important when an archive is requested to be extracted. What seems like a straight-forward operation at first requires some additional mechanism to prevent concurrency problems when multiple HDS instances are using the same cache and receive multiple of these requests at the same time.

The general procedure is depicted in figure 1. The local and remote modification timestamps are used to detect if the archive was modified on the data source in the meantime. If the remote modification timestamp is more current than the local one, the archive changed and must be updated. Consequently, there is no need to update the archive when the archive already exists in the cache and is still up to date. For any other case, the archive must be *locked* by acquiring its corresponding lockfile. After the lockfile was acquired successfully, the modification timestamps are checked again since another HDS instance, which held the lockfile before, could have already updated the archive. If this is not the case, the archive must be downloaded and extracted. After the extraction, the modification timestamp is updated as described in section 6.1.5. Finally, the archive is *unlocked* by releasing the lockfile.

Without this locking mechanism, two HDS instances could potentially write to the same file when downloading and extracting the same archive. This could result in an unpredictable state of the file. Furthermore, the repeated timestamp comparison prevents lost updates. Another advantage of this update mechanism is that it reduces the network traffic of the HDS since archives will only be updated when changed on the data source.

It is important to note that this mechanism only prevents concurrent write access, thus reading the content of the archive is allowed while it is locked. This is due to the fact that the archive is actually downloaded and extracted using a temporary location and is only moved to the actual path once this process is finished. Theoretically, there is a time frame in which one service can read the content of the archive while another one replaces the extracted archive. Since this can only happen when reading the content is slower than downloading, locking,

[6] https://www.npmjs.com/package/proper-lockfile

extracting and moving the extracted archive, this problem was neglected in the scope of this thesis.

6.4 User Interface

The web-based user interface was built using the JavaScript frameworks Vue.js[7] 3 and Bootstrap[8] 5. It is implemented in the class style syntax with property decorators[9] and uses the Vuex[10] store for state sharing and the communication between the single components. It consists of four general views as shown in figure 6.5.

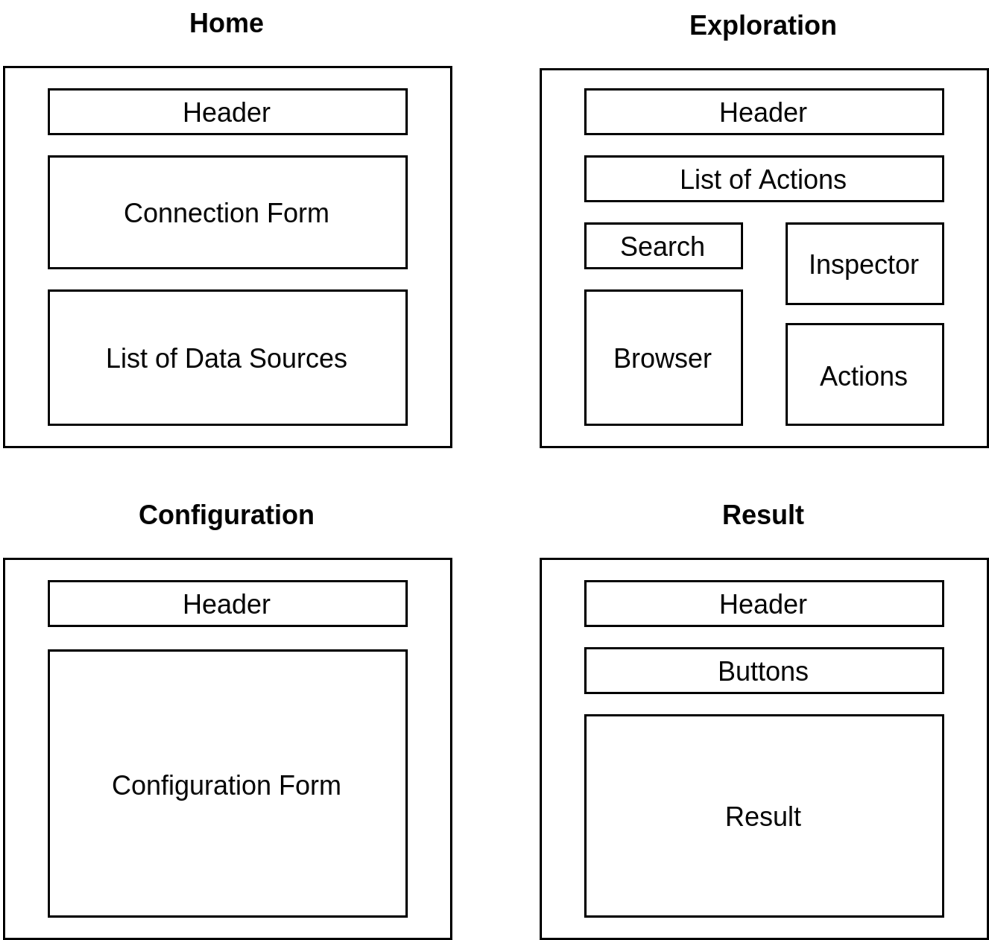

Figure 6.5: Architecture of the user interface

[7] https://vuejs.org/
[8] https://getbootstrap.com/
[9] https://www.npmjs.com/package/vue-property-decorator
[10] https://vuex.vuejs.org/

Each of these views serves one single purpose in the sequential workflow:

1. **Home** - The user can add data sources.
2. **Exploration** - The user can explore the data source in a file browser like manner and add files to the selection to export.
3. **Configuration** - The user can create the export configuration based on the added files from the previous step.
4. **Result** - The user can run the final configuration in order to check if the result meets his expectations.

The user can navigate between these views using the navigation buttons which are included in the header bar. The user interface is not implemented as a Single Page Application (SPA) but uses the the Vue Router[11] to serve the different views as single pages.

[11]https://router.vuejs.org/

7 Demonstration

In this chapter, the usage of the HDS via the user interface will be demonstrated. Therefore, an exemplary use case of exploring and configuring a FTP data source via its user interface is shown. This example was executed on a single host system where the HDS was running on *localhost:8080*, and the user interface was served by a simple HTTP server on *localhost:8081*. For that, the docker-compose.yml file was used to start the two Docker containers by running the command docker-compose up in the project root directory. The screenshots which are displayed in figures 2 - 9 are recorded with a display size of 1920x1200 px and a device pixel ratio of 3.

At first, the URL of the user interface was opened in the browser, displaying the *Home* view of the user interface. At this step, the user can add and remove data sources. In this example, the opendata.dwd.de data source was added as shown in figure 2. Once a data source is added, adding the same URL again is disabled. When trying to add a data source that does not exist, an error message is displayed after the attempt to connect finally fails. After clicking the *Explore* button, the user interface will switch to the *Exploration* view, in which the user can explore the data source.

The *Exploration* view consists of the *Browser*, the *Inspector* and the *Actions* component (see figure 3). The *Browser* is the central component of this view and works similar to a regular file browser. It is used to navigate through the data source and supports pagination within the current directory. The *Browser* also provides a search bar, a button for navigating back in the history, and a refresh button for reloading the current directory contents. The search bar can be used to filter the displayed entries, which is especially useful to find files in a large directory. It can also be used to filter all added files (via @added) or all extracted archives (via @extracted). The *Inspector* displays the properties of the selected file. The *Actions* section provides a list of applicable actions for the currently selected node, such as downloading a file or extracting an archive. These actions are executed in the background and are displayed in a separate list that is only visible when at least one action is still running (see figure 9). By clicking the *Add* button, the selected file is added to the export selection. This is also indicated by the blue highlighted number of added nodes in the top bar. A previously added file can also be removed again, consequently reducing the number of added files by one.

In this example, two files were added. First, the file /weather/alerts/content.log.bz2 was downloaded and added afterwards. Afterwards, the archive jahreswerte_KL_00044_akt.zip in the directory /test/CDC/observations_germany/climate/annual/kl/recent was extracted and inspected. This archive contains 16 files (seven .html files and nine .txt files). The file Metadaten_Geographie_00044.txt was successfully downloaded and added afterwards. Following a symbolic link was tested in the directory /weather/alerts/cap/COMMUNEUNION_EVENT_STAT, which stores several symbolic links to the latest .zip archive. It should be noted that the loading screen is displayed when a response of the HDS exceeds a specific timeout. A good example is the directory /test/weather/weather_reports/synoptic/germany/geojson, which contains about 59000 items and therefore takes some time to get listed (see figure 8).

After the exploration of the data source is finished and all files of interest are identified and added to the export selection, the export configuration has to be created (see figure 4). Therefore, the user navigates to the *Configuration* view. The *Configuration* view lists all added files from the previous step. Once such an entry is selected, a configuration form is displayed that enables the user to create such a configuration as described in section 6.2. The user can save the configuration by clicking on the *Add* button or remove the whole entry from the export configuration by clicking the *Remove* button. The regular expressions provide a help message which is displayed in a new modal. This help message explains how to use the single forms in detail with some short examples (see figure 7). An invalid definition of a regular expression or the other parameters is indicated by red colour. In this example, the entry for the file /weather/alerts/content.log.bz2 was added unmodified. Thus, only the single file should be retrieved when the configuration is evaluated. The other entry is modified to match the second to fifth .txt files in the archive when the alphanumeric descending sort order is applied (see figure 5). Once all entries have been configured, the user can move to the last step and evaluate the resulting configuration.

The *Result* view is the last view of the exploration process. It provides two buttons to view the final JSON configuration and comfortably copy it to the clipboard. In contrast, the major task of this step is to execute the configuration and check if the result matches the expectation. When clicking the *Try it!* button, the configuration is sent to the HDS and the matching files are resolved. In the meantime, the user interface displays the loading screen until a response is retrieved. On success, the list of the matching files is displayed, so that the user can check if the result matches the expectations. Otherwise, a simple error message is displayed. If the result does not match the expectations, the user can navigate back to the *Configuration* view and modify the configuration again. When executing the exemplary configuration, a result list of four files is retrieved which is depicted in figure 6 and matches the expected outcome.

8 Evaluation

8.1 Functionality of the HDS

The HDS enables the user to explore FTP data sources in a structured way, based on the inherent file system hierarchy. In section 5.1, the definition of the data structures is described with respect to its extendability and specific FTP implementation. The implementation details are outlined in section 6.1, especially regarding the extraction of archives and symbolic links. It is shown that archived files can be accessed through the HDS by downloading and extracting the archive via the HDS without major effort of the user.

Therefore, objectives 4.1.1, 4.2.1 and 4.2.2 are fulfilled.

The HDS implements a simple RESTful API that provides all of the above-mentioned functionality. The API was described in detail in section 6.1.1, especially with respect to the violations of the RESTful design. The proof-of-concept user interface uses the API in order to enable the user to explore and configure FTP data sources. Of course, the API can also be used for automating the exploration of FTP data sources.

Therefore, objective 4.1.3 is only partly fulfilled.

Regarding the configuration of data sources, a more complex configuration mechanism based on regular expressions is presented in section 6.2. It is discussed in section 5.4 how this configuration fits into the existing ODS ecosystem and what the potential drawbacks are. Furthermore, a detailed example of how the configuration is resolved is given in section 6.2.3. Finally, the HDS is implemented as a standalone microservice which is categorized into the existing ODS architecture in section 5.4. The source code of the HDS is linted using *eslint*[1] and contains helpful comments to ensure its code quality.

Therefore, objectives 4.1.2 and 4.1.4 are fulfilled.

[1] https://github.com/jvalue/hierarchical-datasources/blob/main/backend/.eslintrc.js

8.2 User Interface

The proof-of-concept user interface enables the user to explore and configure FTP data sources easily. It is implemented using *VueJS* and *Bootstrap* as both of these frameworks provide flexible and robust solutions for building frontend applications. Similar to the backend implementation, the source code is linted using *eslint*[2] in order to apply to common programming guidelines. The responsive layout of the user interface stacks the standard horizontal layout and was realized using the *Bootstrap* grid system to provide a comfortable user experience on mobile devices. Figures 10 - 13 show the responsive layout for all four views on a display with a resolution of 767x1200 px and a device pixel ratio of 3.

Therefore, objectives 4.3.1 and 4.3.2 are fulfilled.

The user is guided through the process of connecting to, exploring and configuring a data source by the sequential workflow of the user interface. This workflow is described in detail in chapter 7. The user can navigate between the four views *Home*, *Exploration*, *Configuration* and *Result* with simple navigation buttons. The *Exploration* view enables the user to navigate through the data source with the *Browser* component. In addition, the user can select single files in order to display additional information about them or even download the file to view its content.

Therefore, objectives 4.3.3 and 4.3.4 are fulfilled.

Finally, the user can add single files to the export selection and further specify the data source configuration in the *Configuration* view. Multiple files can be specified in the data source configuration by using regular expressions in the configuration step. However, the selection of, for example, all files of a single directory is not supported in the *Exploration* view.

Therefore, objective 4.3.5 is only partly fulfilled.

8.3 Automated Tests with a Custom FTP Server

This section focuses on automated tests to evaluate the implemented mechanisms described in sections 6.2.3, 6.3.2, and 6.3.3. For this, the data source must provide special kind of data (e.g., recursive archives), and this data must also be modifiable, for example, when the update mechanism is tested. Thus, it is not feasible to perform those tests on any publicly available data source. In order to provide a flexible test setup, these tests are executed within a docker-compose setup. It consists of a FTP server[3] that provides the custom data directory (Fig-

[2] https://github.com/jvalue/hierarchical-datasources/blob/main/frontend/.eslintrc.js
[3] https://github.com/stilliard/docker-pure-ftpd

ure 15) and two HDS instances. All these services, including the tests themselves, are executed in separate Docker containers. The integration tests are implemented using the *jest*[4] framework and can be run with the Makefile target make it. This target starts all required Docker containers and executes the tests. There is a detailed description of all executed tests in figure 14.

8.3.1 Concurrency

A crucial aspect of the implementation is the shared caching and its locking mechanism (see section 6.3.2). The corresponding tests are located in the concurrency .test.ts file of the integration tests directory. The contained tests are of the following pattern:

1. A request to extract *the same* archive is sent to each HDS instance *n* times
2. Once all requests finished, the events for this archive are retrieved from both HDS instances
3. The actual tests are performed on those retrieved events

The archive that is requested to be extracted should be of a noticeable size, so that the extraction will not be finished before all initial requests have been sent. The following properties are tested and depend on this assumption.

- The number of archiveLocked and archiveReleased events must be equal for each instance
  Reason: Each lock must be acquired and released in order to prevent deadlocks
- The number of archiveLockIsHeld events of both services must be at least *2n - 1*
  Reason: Only the first request immediately acquires the lock, whereas all other requests fail to acquire the lock at least once
- The number of archiveCached events of both services must be equal to *2n - 1*
  Reason: After the first request releases the lock, all other requests make use of the cached entry

This test is performed on both archives bash-5.1-rc1.tar.gz (10.4 MB) and data/exiftool.tar.gz (4.9 MB) with *n = 5*. Of course, additional archives can be used and the number of concurrent requests could be increased as well.

[4] https://jestjs.io/

8.3.2 Recursively Structured Archives

The behavior of extracting recursively structured archives is tested in the test cases of the file recursiveArchives.test.ts. These tests extract the recursive archives /data/code/lib.zip and /data/code/archiveA.zip (see figure 15) and check afterwards if the files within the inner archives are accessible, loadable, and contain the correct information. In addition, it is checked that symbolic links at various depths of the recursive archive are extracted properly, and relative and absolute paths are properly constructed.

8.3.3 Update Mechanism of the File System Cache

The update mechanism of previously extracted archives is tested with some comparatively straightforward test cases that are contained in the update.test.ts file. These tests check three use cases:

1. An archive is extracted again after it had been updated on the data source
   Expected outcome: The event archiveUpdated must occur exactly once

2. An archive is extracted twice without being modified in between
   Expected outcome: The event archiveUpdated must not occur and the event archiveCached must occur exactly once

3. An archive is extracted again after its modification time had been set to a timestamp previous to the first extraction
   Expected outcome: The event archiveUpdated must not occur and the event archiveCached must occur exactly once

9 Conclusion

In summary, the outweighing majority of objectives were met. The implementation demonstrates the benefits when dealing with FTP data sources, especially regarding archived data. The HDS furthermore supports complex configurations based on regular expressions without major effort for the user. Consequently, this software provides additional value to developers who are working with FTP based data sources.

Therefore, it would be beneficial to integrate the HDS into the ODS. A reasonable solution for the long term would be to *merge* the HDS with the existing *Datasource* service to combine similar functionality into one single service. In this step, the support of parameterizable data sources could be integrated as well (Wächtler, 2021).

Due to the limited time of this thesis, there are some additional improvements for the HDS which already have been identified. First of all, an advanced solution for removing cached archives (see section 5.3) must be developed. Furthermore, clickable path components can be integrated into the user interface for a better navigation experience, symbolic links can be supported by the export configuration, and an additional regular expression search on the overall data source can be added in order to simplify the process of finding files based on their name.

Whereas these changes improve or extend the current functionality for FTP data sources, another benefit would be the support of other hierarchical data sources like RESTful APIs. Supporting this kind of API would increase the value of the software by covering an additional variety of use cases and could be implemented in the scope of an upcoming thesis.

Appendices

A Conceptual Designs

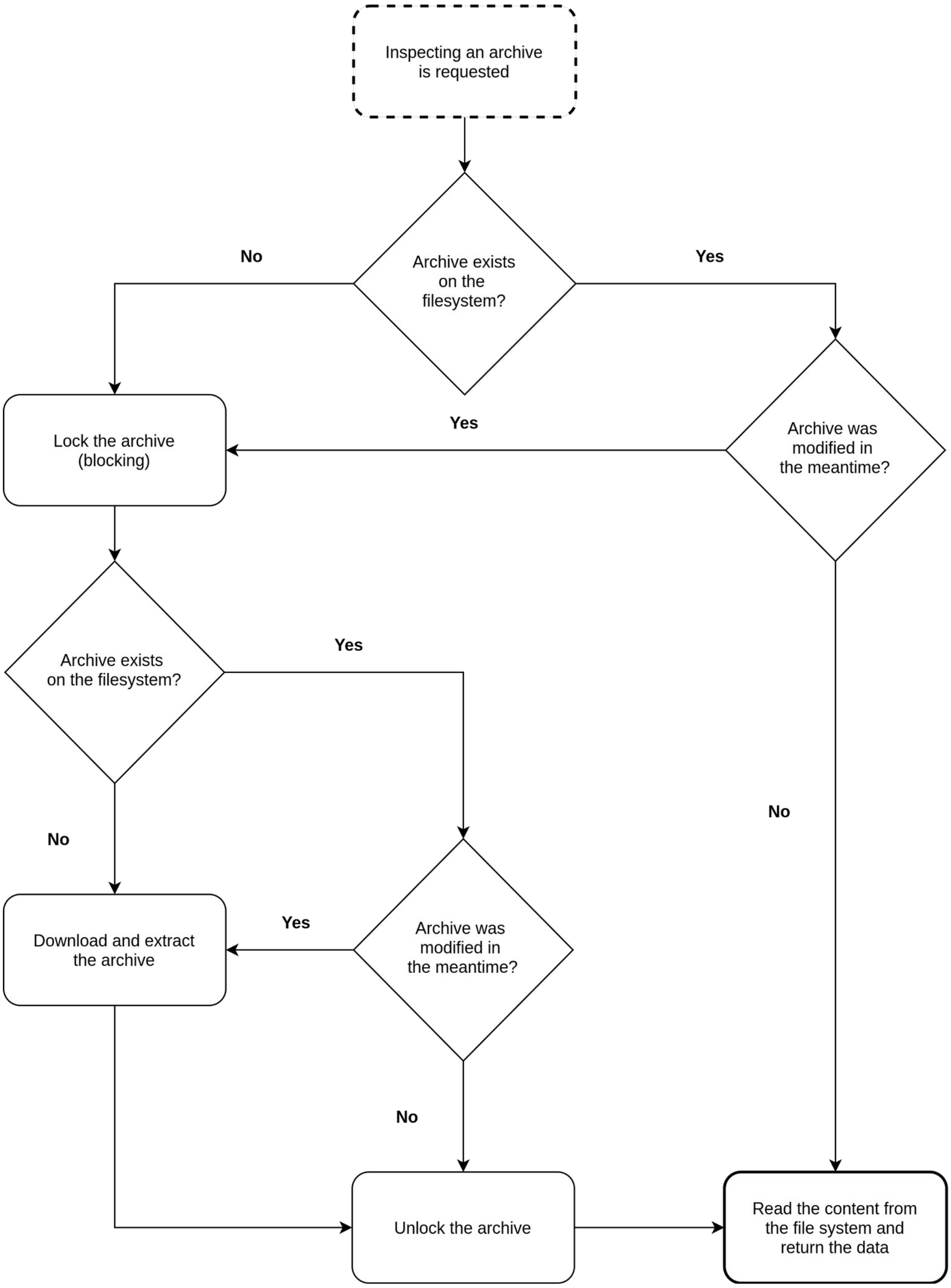

Figure 1: Flowchart of the caching procedure

B User Interface

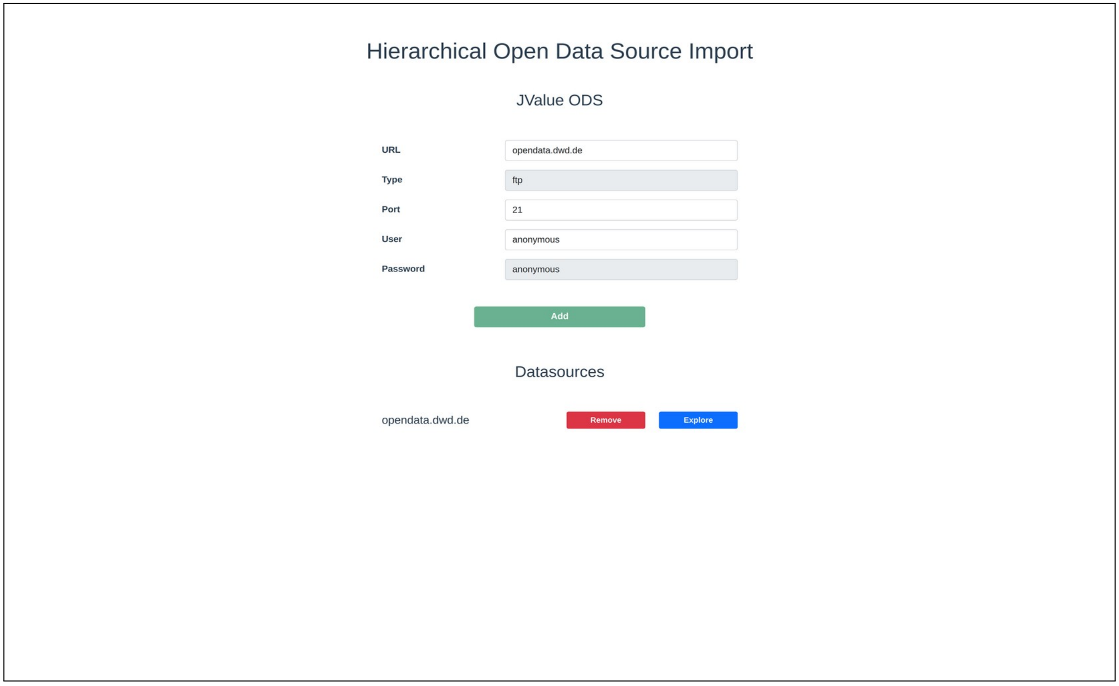

Figure 2: The *Home* view of the user interface

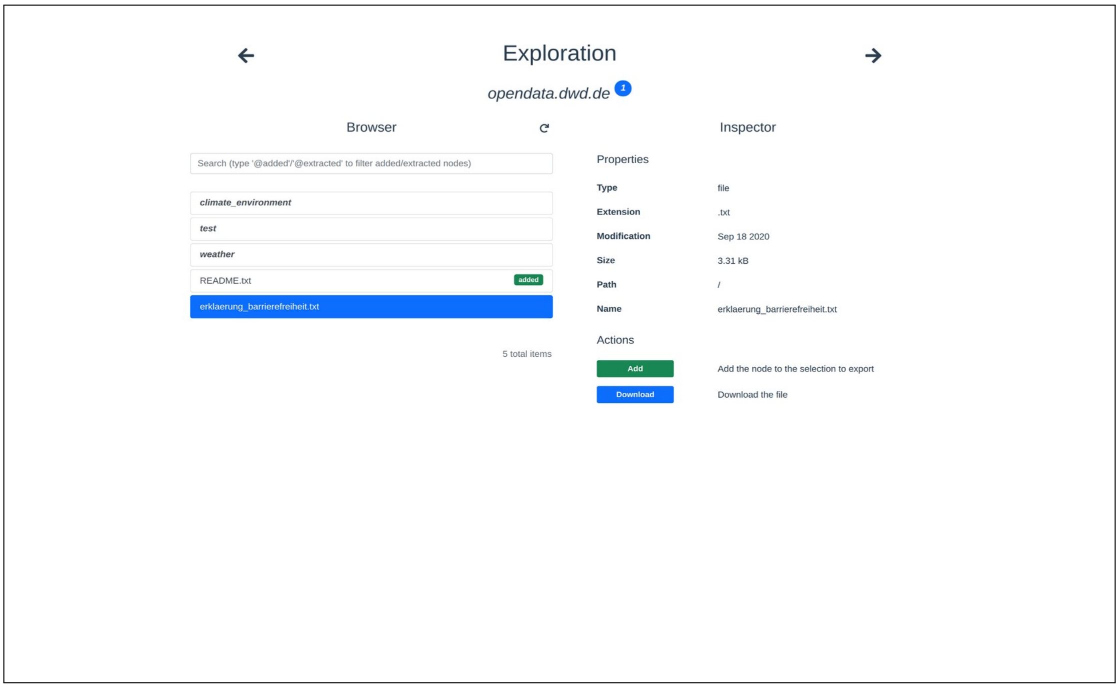

Figure 3: The *Exploration* view of the user interface

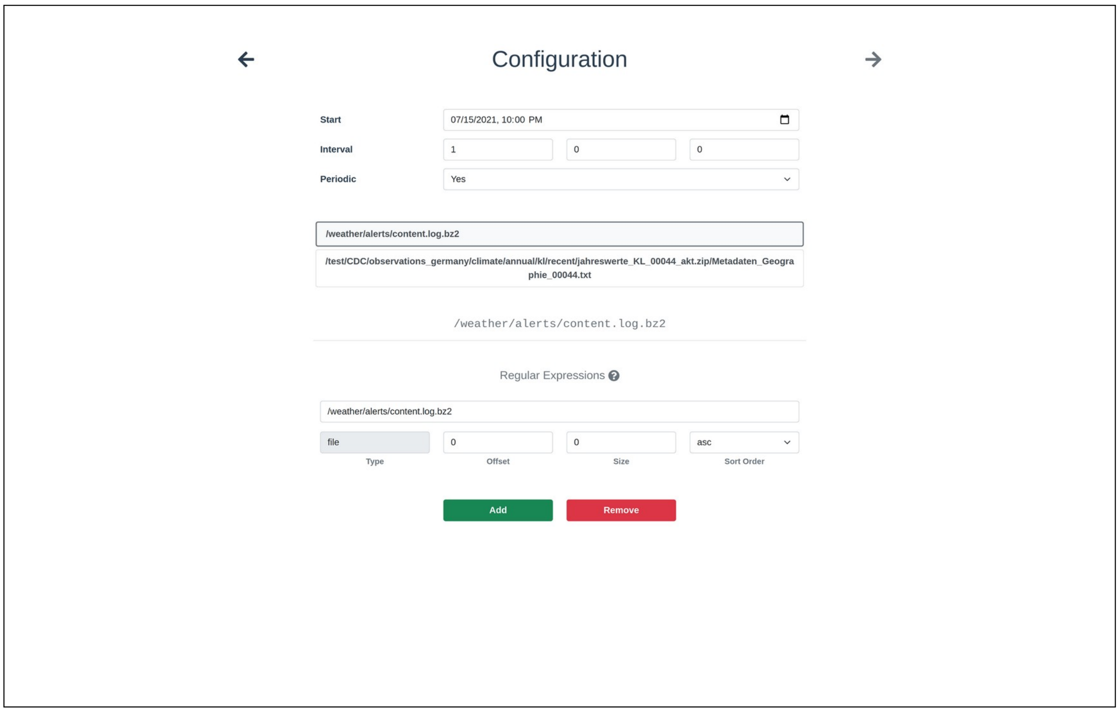

Figure 4: The *Configuration* view of the user interface (1)

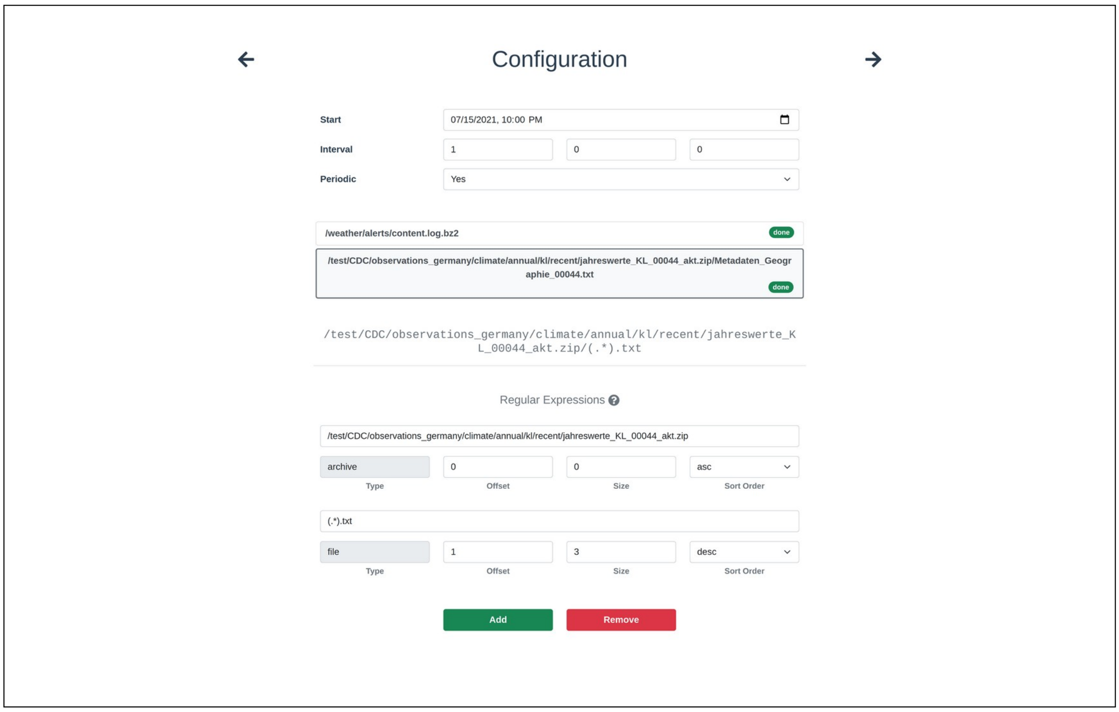

Figure 5: The *Configuration* view of the user interface (2)

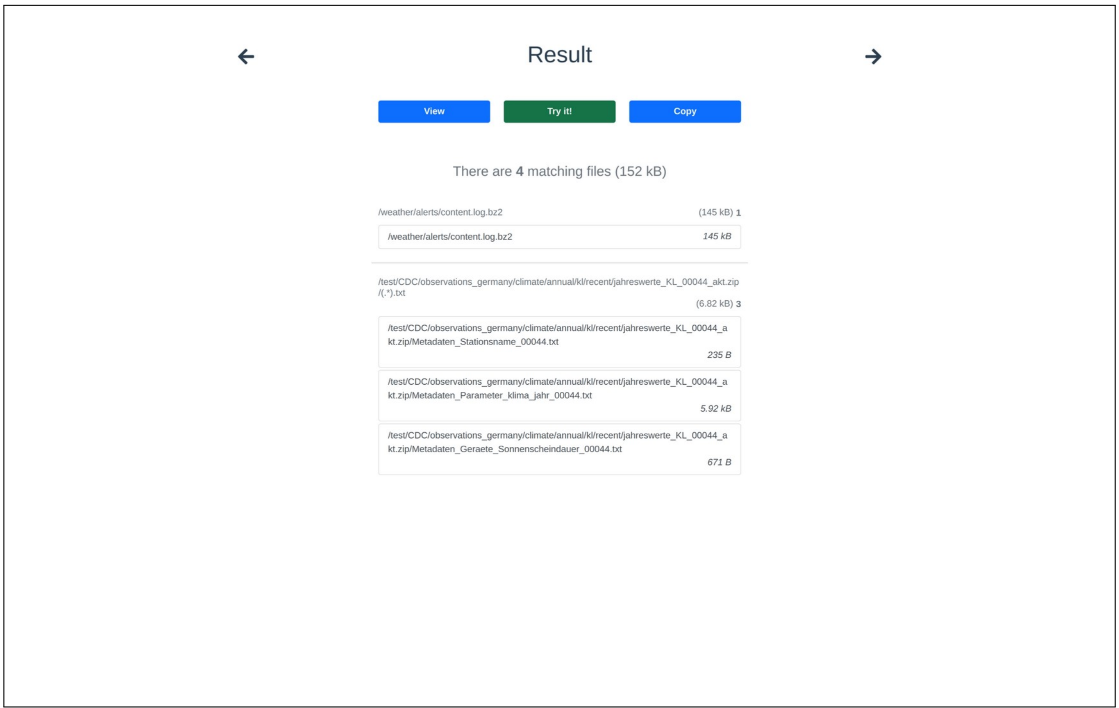

Figure 6: The *Result* view of the user interface

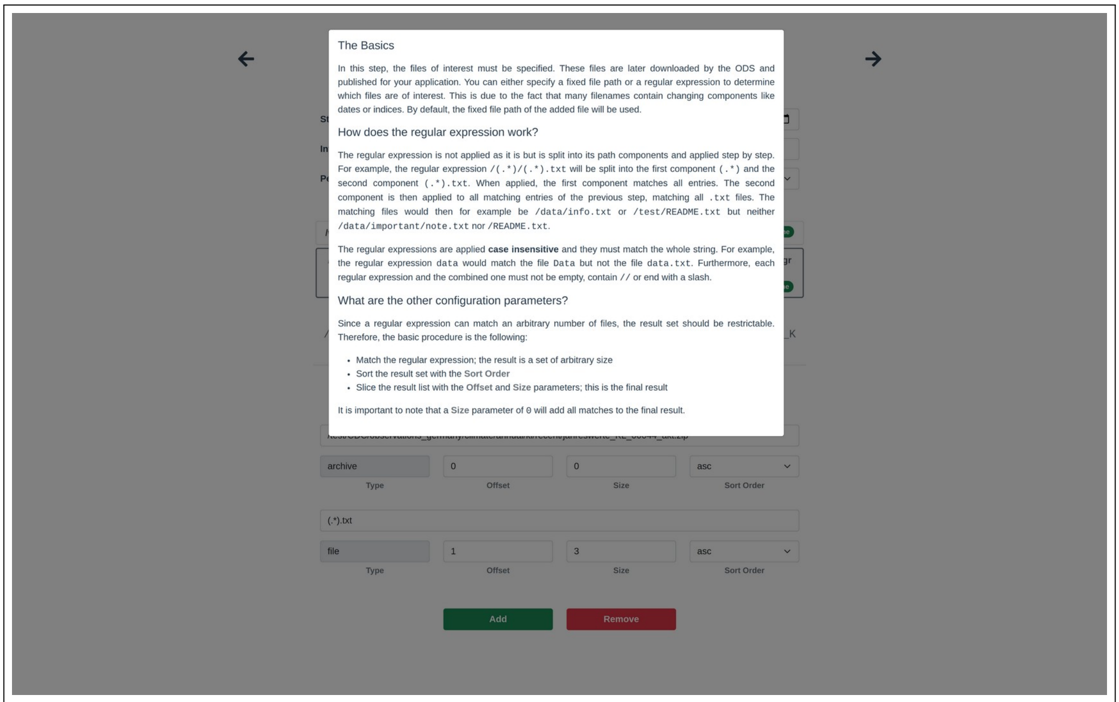

Figure 7: The help message of the *Configuration* view

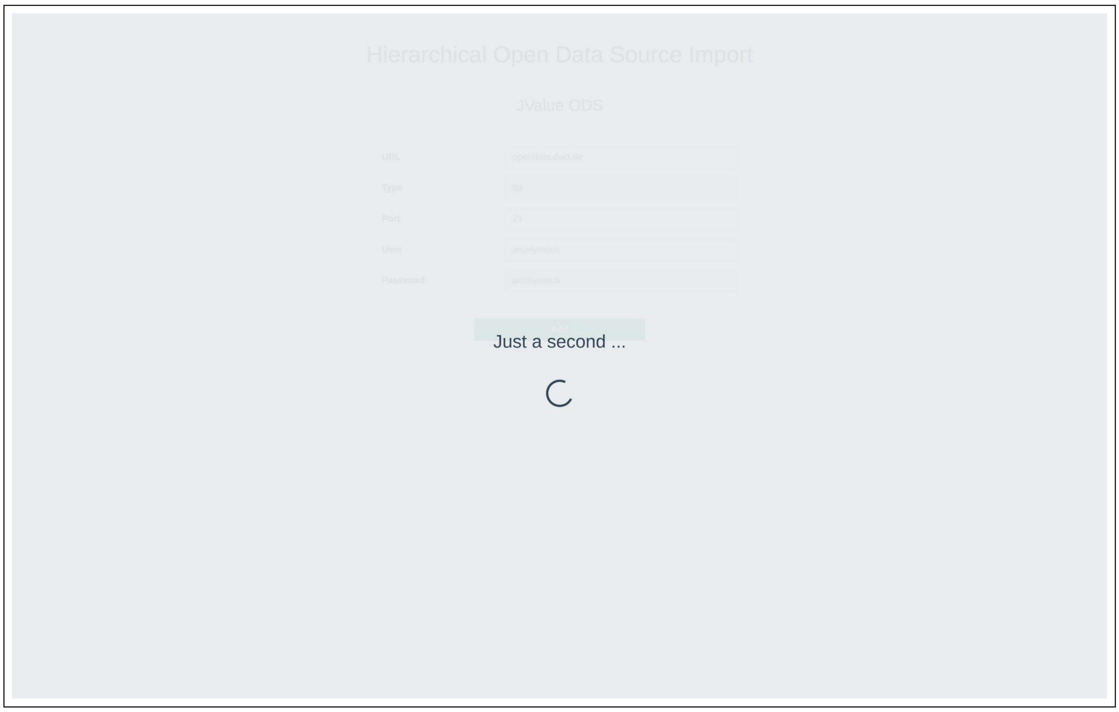

Figure 8: The loading screen of the user interface

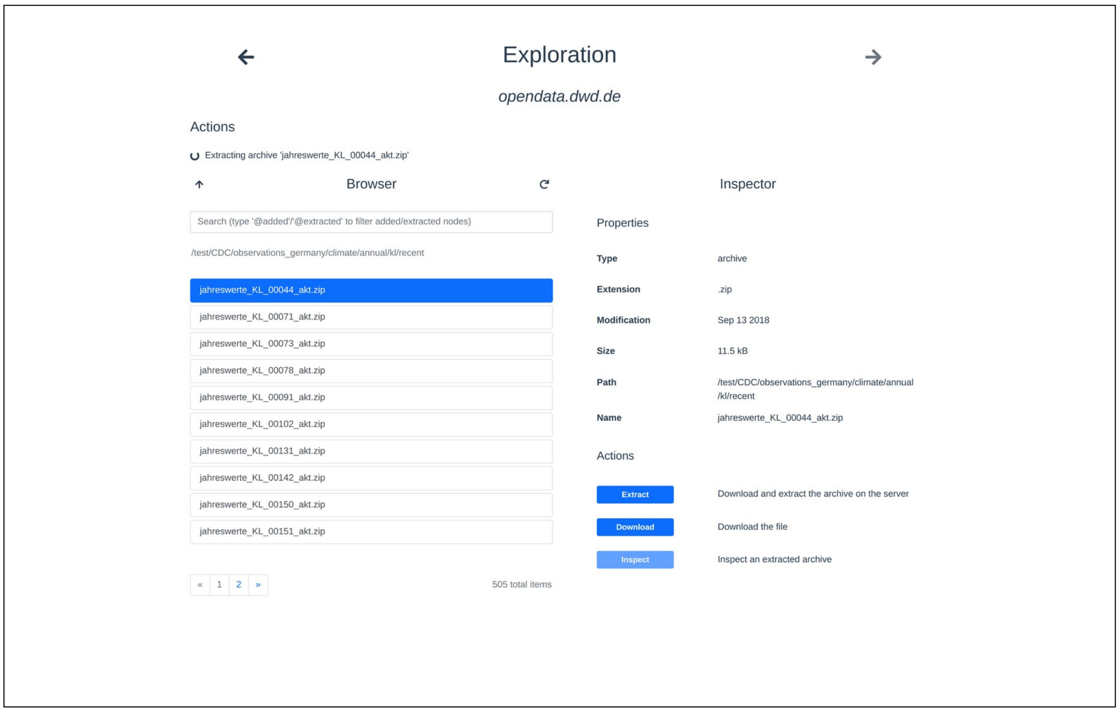

Figure 9: Display of running actions in the *Exploration* view

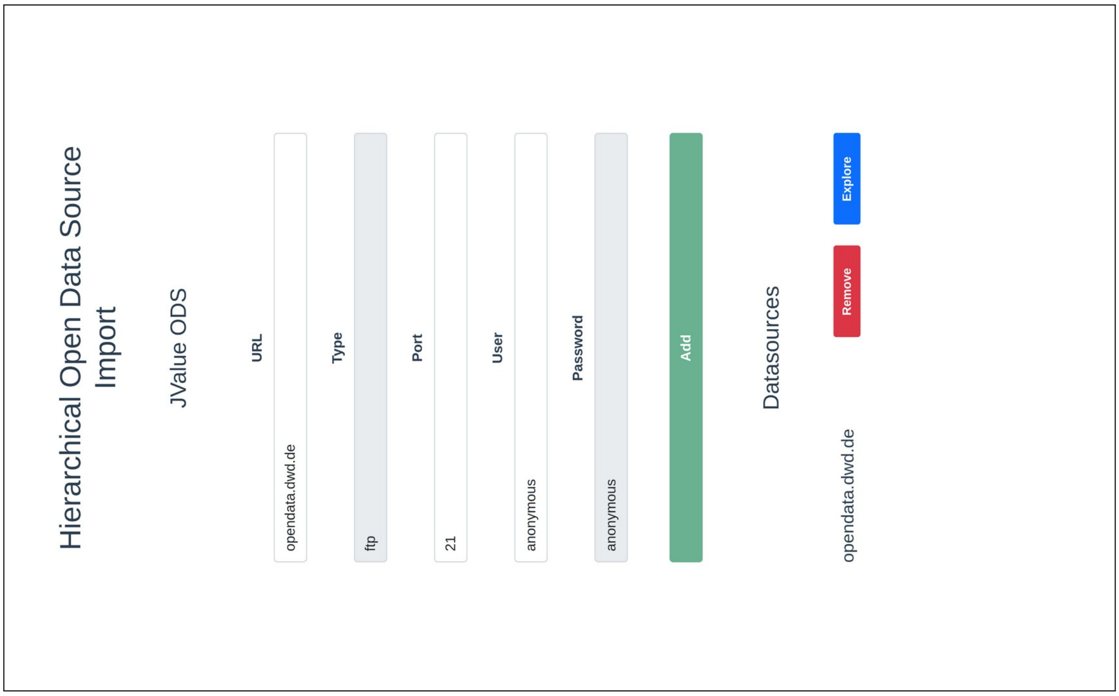

Figure 10: Responsive layout of the *Home* view

Figure 11: Responsive layout of the *Exploration* view

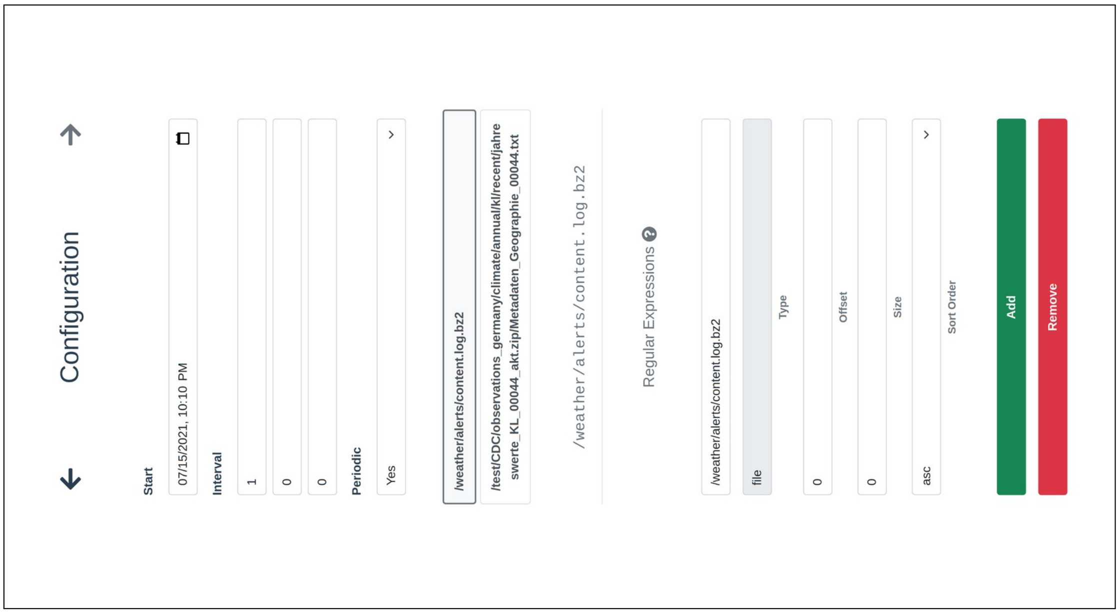

Figure 12: Responsive layout of the *Configuration* view

Figure 13: Responsive layout of the *Result* view

C Miscellaneous

1. Basic setup
    - Test if the API is accessible
    - Test not existing API endpoint ’/idontexist’
2. Data Sources
    - Test if the API is accessible
    - Test not existing API endpoint ’/idontexist’
3. Node
    - Check content of the root
    - Check the content of a .tar.gz archive
    - Extract an empty .zip archive
    - Extract an .zip archive and check its content
4. Recursive Archives
    - Test archive ’/data/code/lib.zip’
    - Test archive ’/data/code/archiveA.zip’
5. Update archives
    - Extract archive again after it was updated
    - Extract an archive twice
    - Extract an archive with past modified timestamp
6. Concurrency
    - Extract the archive ’/bash-5.1-rc1.tar.gz’ concurrently
    - Extract the archive ’/data/exiftool.tar.gz’ concurrently
7. Exports
    - Simple export configuration with a single file
    - Export configuration with ignore case
    - Export configuration with sort order and slicing
    - Export configuration with archived files

Figure 14: List of all integration tests

```
/
├── archive.zip
├── bash-5.1-rc1.tar.gz
├── readme.md
├── README.md
├── README.txt
├── README.TXT
└── data
    ├── audio.tar.gz
    ├── audio.tgz
    ├── data
    ├── data_20200502_raw.csv
    ├── data_20200601_raw.csv
    ├── data_20210502_raw.csv
    ├── empty.zip
    ├── exiftool.tar.gz
    ├── images.zip
    │   ├── image_1.jpg
    │   ├── image_2.png
    │   └── image_3.jpeg
    └── code
        ├── lib.zip
        │   └── lib/
        │       ├── share.zip
        │       │   └── share/
        │       │       └── binary
        │       └── usr.zip
        │           └── usr/
        │               └── binary
        └── archiveA.zip
            └── archiveA/
                └── archive1.zip
                    └── archive1/
                        ├── test.txt
                        ├── wow
                        ├── wow1
                        └── archive2.zip
                            └── archive2/
                                └── file42.txt
```

Figure 15: Structure of the test data

URL	Port	Description
ftp.wwpdb.org	21	• *Protein Data Bank* • 3-D structure of biological macromolecules
ftp.cdc.gov	21	• *The National Center for Health Statistics (NCHS)* • Health statistics information • Survey data
ofacftp.treas.gov	21	• *Office of Foreign Assets Control (OFAC)* • List of imposed sanctions by the U.S • Specially Designated Nationals And Blocked Persons Lists (SDNs)
opendata.dwd.de	21	• *Deutscher Wetterdienst* • German weather and climate data
ftp.census.gov	21	• *United States Census Bureau* • American Community Survey (ACS) data files
ftp.esrf.eu	21	• *EUMETSTAT* • Global and regional marine/atmosphere data

Table 1: Exemplary list of public FTP servers

```
{
  "trigger": {
    ...
  },
  "connection": {
    "type": "ftp",
    "url": "localhost",
    "port": 21,
    "user": "user",
    "password": "password"
  },
  "entries": [
    [
      {
        "offset": 0,
        "size": 0,
        "sort": "asc",
        "type": "archive",
        "regex": "/data/images.zip"
      },
      {
        "offset": 1,
        "size": 1,
        "sort": "desc",
        "type": "files",
        "regex": "image_(.*)"
      }
    ]
  ]
}
```


Listing C.1: Example of an export configuration

1. ExportRegex for regular expression */data/images.zip* on result list []
   (a) Apply the regular expression for each path component
      i. Regular Expression: data
         - Nodes of [/]:
           [archive.zip, bash-5.1-rc1.tar.gz, ..., data]
         - Result: [/data]
      ii. Regular Expression: images.zip
         - Nodes of [/data]:
           [audio.tar.gz, audio.tgz, ..., code]
         - Result: [/data/images.zip]
   (b) Slice the result list
      i. Apply type archive: [/data/images.zip]
      ii. Apply sort asc: [/data/images.zip]
      iii. Apply offset 0: [/data/images.zip]
      iv. Apply size 0: [/data/images.zip]
   (c) Result of first ExportRegex: [/data/images.zip]
      - Extract the archive before continuing
2. ExportRegex for regular expression image_(.*) on result list [/data/images.zip]
   (a) Apply the regular expression for each path component
      i. Regular Expression: image_(.*)
         - Nodes of [/data/images.zip]:
           [image_1.jpg, image_2.png, image_3.jpeg]
         - Result:
           [/data/images.zip/image_1.jpg,
           /data/images.zip/image_2.png,
           /data/images.zip/image_3.jpeg]
   (b) Slice the result list
      i. Apply type file:
         [/data/images.zip/image_1.jpg, ...,
         /data/images.zip/image_3.jpeg]
      ii. Apply sort desc:
         [/data/images.zip/image_3.jpeg, ...,
         /data/images.zip/image_1.jpg]
      iii. Apply offset 1:
         [/data/images.zip/image_2.png,
         /data/images.zip/image_1.jpg]
      iv. Apply size 1: [/data/images.zip/image_2.png]
   (c) Result of second ExportRegex: [/data/images.zip/image_2.png]
      - Final result

Figure 16: Example of resolving an export configuration

D API

```
/*
 * POST /datasources
 *
 * Add a new data source.
 */

// Request body
{
    "type": "ftp",
    "url": "opendata.dwd.de",
    "port": 21,
    "user": "anonymous",
    "password": "anonymous"
}
```

Listing D.1: API example: Add a new data source

```
/*
 * GET /datasources
 *
 * Get all imported data sources.
 */

// Response body
[
    {
        "type": "ftp",
        "url": "opendata.dwd.de",
        "port": 21,
        "user": "anonymous",
        "password": "anonymous"
    }
]
```

Listing D.2: API example: Get all imported data sources

```
/*
 * GET /datasources/opendata.dwd.de/%2F
 *
 * Get the content of a node.
 */

// Response body
{
    "url": "/",
    "name": "/",
    "properties": {
        "path": [
            {
                "path": "/",
                "type": "directory"
            }
        ],
        "type": "directory"
    },
    "isLeaf": false,
    "actions": [],
    "children": [
        {
            "url":"/README.txt",
            "name":"README.txt",
            ...
        }
        ...
    ]
}
```

Listing D.3: API example: Get the content of a node

```
/*
 * POST /datasources/opendata.dwd.de/%2FREADME.txt/
 *
 * Download a file.
 */

// Request body
{
    "identifier": "download"
}

// Response body
Im Rahmen seines gesetzlichen Auftrags stellt der DWD ...

...

Ihre Daten werden nicht an Dritte weitergegeben.
```

Listing D.4: API example: Download a file

```
/*
 * POST /export
 *
 * Get the matching nodes of an export configuration.
 */

// Request body
{
  "trigger": {
    "periodic": true,
    "firstExecution": "2021-07-15T10:15",
    "interval": 86400
  },
  "connection": {
    "type": "ftp",
    "url": "opendata.dwd.de",
    "port": 21,
    "user": "anonymous",
    "password": "anonymous"
  },
  "entries": [
    [
      {
        "offset": 0,
        "size": 0,
        "sort": "asc",
        "type": "file",
        "regex": "/(.*).txt"
      }
    ]
  ]
}

// Response body
[
  [
    {
      "url": "/erklaerung_barrierefreiheit.txt",
      "name": "erklaerung_barrierefreiheit.txt",
      "properties": {
        "type": "file",
        "extension": ".txt",
```

```
        "modification": "Sep 18 2020",
        "size": 3312,
        "path": [
          {
            "path": "/erklaerung_barrierefreiheit.txt",
            "type": "file"
          }
        ]
      },
      ...
    },
    {
      "url": "/README.txt",
      "name": "README.txt",
      "properties": {
        "type": "file",
        "extension": ".txt",
        "modification": "Jul 25 2017",
        "size": 528,
        "path": [
          {
            "path": "/README.txt",
            "type": "file"
          }
        ]
      },
      ...
    }
  ]
]
```


Listing D.5: API example: Get the matching nodes of an export configuration

References

Braunschweig, K., Eberius, J., Thiele, M. & Lehner, W. (2012). The state of open data.

European Commission. (n.d.). *What is open data*. Retrieved June 21, 2021, from https://data.europa.eu/elearning/en/module1/#/id/co-01

Fielding, R. T. (2008). *Rest apis must be hypertext-driven*. Retrieved July 22, 2021, from https : / / roy. gbiv . com / untangled / 2008 / rest - apis - must - be - hypertext-driven

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation). University of Californa, Irvine.

Gleason, M. (2005). *The file transfer protocol (ftp) and your firewall / network address translation (nat) router / load-balancing router*. Retrieved June 16, 2021, from https : / / www . ncftp . com / ncftpd / doc / misc / ftp _ and _ firewalls.html

Hirsch, F., Kemp, J. & Ilkka, J. (2007). *Mobile web services - architecture and implementation*. John Wiley & Sons.

Housley, R. & Yee, P. E. (2000). *Encryption using KEA and SKIPJACK* (RFC No. 2773). RFC Editor. RFC Editor. https://doi.org/10.17487/RFC2773

Lafon, Y., Mendelsohn, N., Hadley, M., Karmarkar, A., Nielsen, H. F., Moreau, J.-J. & Gudgin, M. (2007). *SOAP version 1.2 part 1: Messaging framework (second edition)* (W3C Recommendation) [https://www.w3.org/TR/2007/REC-soap12-part1-20070427/]. W3C.

Lewis, J. & Fowler, M. (2014). *Microservices - a definition of this new architectural term*. Retrieved June 21, 2021, from https://martinfowler.com/articles/microservices.html

Lunt, S. J. (1997). FTP Security Extensions. https://doi.org/10.17487/RFC2228

Mumbaikar, S., Padiya, P. et al. (2013). Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, *3* (5), 1–4.

Newman, S. (2015). *Building microservices: Designing fine-grained systems* (1st). O'Reilly Media.

NGINX Inc. (2016). *Nginx announces results of 2016 future of application development and delivery survey*. Retrieved June 21, 2021, from https://www.

nginx.com/press/nginx-announces-results-of-2016-future-of-application-development-and-delivery-survey/

Open Knowledge Foundation. (n.d.). *Open definition*. Retrieved June 21, 2021, from http://opendefinition.org/

Postel, J. & Reynolds, J. (1985). File Transfer Protocol. https://doi.org/10.17487/RFC0959

Publications Office of the European Union. (2020). *The benefits and value of open data*. Retrieved June 21, 2021, from https://data.europa.eu/en/highlights/benefits-and-value-open-data

Reinsel, D., Gantz, J. & Rydning, J. (2017). *Data age 2025: The evolution of data to life-critical* (tech. rep.). International Data Corporation (IDC).

Schwarz, G. (2019). *Migration the JValue ODS to Microservices* (Master's thesis). Friedrich-Alexander Universität Erlangen-Nürnberg.

Wächtler, J. (2021). *Design and Implementation of Parameterizable Data Import for the JValue ODS* (Bachelor's Thesis). Friedrich-Alexander Universität Erlangen-Nürnberg.

Xia, L., Chao-sheng, F., Ding, Y. & Can, W. (2010). Design of secure ftp system. *2010 International Conference on Communications, Circuits and Systems (ICCCAS)*, 270–273. https://doi.org/10.1109/ICCCAS.2010.5582002